

Gurobi Implementation Roadmap

From Model to Solution



A Complete Guide for Technical Users

I. Prerequisites & Preparation	3
System Requirements	3
Before You Begin	3
License Acquisition	3
Choosing Your Installation Method	3
II. Installation	4
Installation via amply (Recommended)	4
Step 1: Install amply	4
Step 2: Install Gurobi Module	4
Step 3: Activate License	4
III. Verification & Testing	5
IV. Working with Gurobi	7
Understanding Gurobi Options	7
Setting Parameters	7
Algorithm Selection	7
Advanced Gurobi Features	8
Infeasibility Analysis (IIS)	8
Performance Tuning	9
V. Practical Examples	10
End-to-End Examples	10
Example 1: Linear Programming — Production Planning	10
Example 2: Mixed-Integer Programming — Facility Location	12
Example 3: Large-Scale Problem — Network Optimization	14
VI. Troubleshooting & Best Practices	15
Common Issues & Solutions	15
Installation Failures	15
License Activation Problems	15
Memory and Performance Issues	15
Best Practices	15
Model Organization and File Structure	15
Version Control	15
Documentation Standards	15
Performance Benchmarking	16
VII. Resources & Next Steps	17
Official Documentation	17
Community & Support	17
Advanced Topics to Explore	17
Additional Resources	17

I. Prerequisites & Preparation

System Requirements

Before beginning installation, ensure your system meets the following requirements:

- **Operating System:** Windows 10/11, Linux, or macOS
- **Python Version:** Python 3.7 or higher (Python 3.9+ recommended)
- **Development Environment:** Jupyter Notebook, or Visual Studio Code

Before You Begin

License Acquisition

You will need a valid AMPL license to use Gurobi. License types include:

- **Free Trial License:** Available for evaluation purposes
- **Community Edition License:** For educational institutions and research
- **Commercial License:** Full-featured license for commercial use

Your license will be provided as a UUID (Universally Unique Identifier) that looks like:

```
12345678-1234-1234-1234-123456789abc
```

Choosing Your Installation Method

AMPL and Gurobi can be installed using the following method:

- **amplpy with Python Modules (Recommended):** Best for Python users who want seamless integration with Python workflows. Quick installation via pip with automatic dependency management.
- **For other language APIs (R, MATLAB, C++, Java),** see <https://dev.ampl.com/ampl/install.html>

Note: Both methods provide full Gurobi functionality. The choice depends on your preferred development environment.

II. Installation

Installation via amplpy (Recommended)

This method provides the fastest path to getting started with AMPL and Gurobi in Python environments.

Step 1: Install amplpy

Open your terminal or command prompt and run:

```
python -m pip install amplpy --upgrade
```

Common Issues:

- **Permission denied:** Add --user flag or use sudo on Linux/macOS
- **Proxy issues:** Configure pip proxy with --proxy http://user:pass@server:port

Step 2: Install Gurobi Module

Once amplpy is installed, add the Gurobi solver module:

```
python -m amplpy.modules install gurobi
```

What Gets Installed:

- The latest version of Gurobi solver optimized for your platform
- Required libraries and dependencies
- Integration files for seamless AMPLpy operation

Installation Verification:

Modules are installed in your Python environment's site-packages directory. To verify your installed modules run:

```
python -m amplpy.modules installed
```

Step 3: Activate License

Activate your AMPL license using your UUID:

```
python -m amplpy.modules activate <license-uuid>
```

Replace <license-uuid> with your actual license UUID (without angle brackets).

License Verification:

- Upon successful activation, you'll see a confirmation message
- The license file will be stored in your AMPL configuration directory
- Your license type and limitations will be displayed

Verification Checkpoint:

- Check installation by running: `python -c "import amplpy; print(amplpy.__version__)"`
- You should see the amplpy version number printed
- To test the AMPL license, you can run in your terminal: `python -m amplpy.modules run ampl -vvq`

III. Verification & Testing

Test the complete workflow with a simple optimization problem:

Simple Linear Programming Example:

Create model file (test_model.mod)

```
# Define variables
var x >= 0;
var y >= 0;

# Objective function
maximize profit: 3*x + 2*y;

# Constraints
subject to labor: 2*x + y <= 100;
subject to materials: x + y <= 80;
```

Access and solve through Python

```
from amplpy import AMPL

# Create AMPL instance
ampl = AMPL()

# Load model
ampl.read("test_model.mod")

# Set solver and solve
ampl.option["solver"] = "gurobi"
ampl.solve()

# Check solve status
if ampl.solve_result == "solved":
    # Get objective value
    profit = ampl.get_objective("profit")
    print(f"Optimal profit: {profit.value()}")

    # Get variable values
    x_val = ampl.get_variable("x").value()
    y_val = ampl.get_variable("y").value()
    print(f"x = {x_val}, y = {y_val}")
else:
    print(f"Solve failed: {ampl.solve_result}")
```

Expected Output:

Gurobi should solve the problem and display:

```
Optimal profit: 180.0
x = 20.0, y = 60.0
```

Interpretation:

- **Solver Status:** optimal solution (problem solved successfully)

Number	String	Interpretation
0-99	solved	Optimal solution found
100-199	solved?	Optimal solution indicated, but error likely
200-299	infeasible	Constraints cannot be satisfied
300-399	unbounded	Objective can be improved without limit
400-499	limit	Stopped by a limit that you set (such as on iterations)
500-599	failure	Stopped by an error condition in the solver routines

- **Objective Value:** 180 (maximum profit achieved)
- **Decision Variables:** $x=20$, $y=60$ (optimal production quantities)

Troubleshooting If Solve Fails:

- **License error:** Verify license activation (check license UUID), for additional troubleshooting refer to <https://dev.ampl.com/help/dynamic-license-troubleshooting.html>
- **Solver not found:** Ensure Gurobi is in PATH and properly installed through the amplpy module
- **Syntax errors:** Check for typos in the model formulation

IV. Working with Gurobi

Understanding Gurobi Options

Gurobi provides hundreds of parameters to control solver behavior. Understanding and using these options is key to achieving optimal performance. Some optimization specialists dedicate significant time to mastering Gurobi's full parameter space. This guide covers the most impactful options for typical users—focusing on the 80/20 of practical usage.

For comprehensive Gurobi documentation, see:

<https://docs.gurobi.com/projects/optimizer/en/current/reference/parameters.html>

For the complete list of Gurobi options available through AMPL, see:

<https://dev.ampl.com/solvers/gurobi/options.html>

Major Categories of Options:

- **Algorithm Selection:** Choose between primal simplex, dual simplex, barrier, concurrent, deterministic concurrent, or PDHG methods
- **MIP Parameters:** Control cuts, heuristics, branching strategies, and node selection for mixed-integer programs
- **Tolerances:** Set feasibility, optimality, and integrality tolerances
- **Resource Limits:** Time limits, iteration limits, memory limits
- **Output Control:** Display settings, logging options, solution file generation

Commonly Used Options:

- `mipgap`: Relative MIP gap tolerance (default 0.0001%)
- `timelimit`: Maximum solution time in seconds
- `threads`: Number of parallel threads (0 = automatic, up to 32)
- `outlev`: Control solver output (0=off, 1=on)
- `presolve`: Presolve level (-1=automatic, 0=off, 1=conservative, 2=aggressive)
- `mipfocus`: MIP solution strategy (0=balance, 1=feasibility, 2=optimality, 3=bound)

Setting Parameters

Gurobi options can be set in multiple ways depending on your workflow.

Via Python/AMPLpy

```
from amply import AMPL
ampl = AMPL()
ampl.option["solver"] = "gurobi"
ampl.option["gurobi_options"] = "mipgap=0.01 timelimit=300"
ampl.solve()
```

Algorithm Selection

Choosing the right algorithm can dramatically impact solution time. Gurobi offers several optimization algorithms for different problem types.

Linear Programming Algorithms:

- Primal Simplex (*method=0*)
- Dual Simplex (*method=1*). Default for LP and MIP node relaxations
- Barrier (*method=2*): Best for large, sparse problems; uses interior-point method
- Concurrent (*method=3*): Runs multiple algorithms in parallel, returns the first to finish
- Deterministic Concurrent (*method=4*): Like concurrent, but produces reproducible results
- PDHG (*method=5*): Primal-Dual Hybrid Gradient; good for very large LPs where approximate solutions are acceptable

Mixed-Integer Programming Strategies:

- Balance (*mipfocus=0*): Default; balances finding feasible solutions and proving optimality
- Emphasis on Feasibility (*mipfocus=1*): Finds good feasible solutions quickly
- Emphasis on Optimality (*mipfocus=2*): Proves optimality rigorously (may take longer)
- Emphasis on Moving Best Bound (*mipfocus=3*): Improves the lower/upper bound quickly

Advanced Gurobi Features

Infeasibility Analysis (IIS)

When a model is infeasible, Gurobi can identify an Irreducible Inconsistent Subsystem (IIS) — a minimal set of constraints that cannot be satisfied simultaneously. To return consistent IIS results, it is recommended to set AMPL's presolve (distinct from Gurobi presolve) to 0, through `AMPL.option["presolve"] = 0`

Detecting Infeasibility:

```
AMPL = AMPL()
AMPL.read("infeasible_model.mod")
AMPL.option["solver"] = "gurobi"
AMPL.option["gurobi_options"] = "iis=1"

AMPL.solve()

if AMPL.solve_result == "infeasible":
    var_iis, con_iis = AMPL.get_iis()
    print(var_iis, con_iis)
```

Interpreting IIS Output:

- Gurobi will report which constraints are in the IIS
- Review these constraints to identify modeling errors or conflicting requirements
- Common causes: typos, incorrect bounds, mutually exclusive requirements

Resolving Infeasibilities:

- Relax one or more constraints in the IIS
- Adjust bounds or right-hand sides
- Add slack/surplus variables with penalties in objective

Solution Pool (Multiple Solutions):

Generate and store multiple near-optimal solutions:

```
AMPL.option["gurobi_options"] = "sol:poollimit=10 sol:poolmode=2\nsol:stuck=solutions"
```

This creates alternative solutions that may be valuable for scenario analysis.

Lazy Constraints:

For problems with many constraints where only a subset is active at optimality, lazy constraints can improve performance by adding constraints dynamically during the branch-and-bound process. This requires using AMPL's `ampls` (AMPL solver libraries) library or callback functions.

Performance Tuning

Memory Management:

- `nodefilestart`: Gigabytes of memory to use for MIP tree nodes before writing to disk (default: no limit)
- `nodefiledir`: Directory where node files are written (default: current directory)

Parallel Processing:

```
ampl.option["gurobi_options"] = "threads=8"
```

- `threads=0`: Automatic (uses all available cores, up to 32)
- `threads=N`: Use N threads

Tuning Tool:

Gurobi includes an automatic parameter tuning tool that experiments with different parameter settings:

```
ampl.option["gurobi_options"] = "tunebase=results tunetimelim=3600"
```

The tuner will report recommended parameter settings for your specific problem, writing results to `.prm` files derived from the tunebase name.

Warm Starting

Provide initial solution values to help Gurobi start the branch-and-bound process:

```
from amplpy import AMPL
ampl = AMPL()
ampl.read("facility.mod")
# ... set data ...

# Set initial values for binary variables
# (from previous solve or heuristic)
ampl.get_variable('open').set_values({
    'F1': 1,
    'F2': 0,
    'F3': 1
})

# Solve with warm start
ampl.option["solver"] = "gurobi"
ampl.solve()
```

V. Practical Examples

End-to-End Examples

Example 1: Linear Programming — Production Planning

Problem Description:

A factory produces two products, A and B. Each product requires labor and materials. The goal is to maximize profit while respecting resource constraints.

Model File (production.mod):

```
# Sets and Parameters
set PRODUCTS;
param profit {PRODUCTS};
param labor_required {PRODUCTS};
param material_required {PRODUCTS};
param labor_available;
param material_available;

# Decision Variables
var produce {PRODUCTS} >= 0;

# Objective Function
maximize total_profit:
    sum {p in PRODUCTS} profit[p] * produce[p];

# Constraints
subject to labor_limit:
    sum {p in PRODUCTS} labor_required[p] * produce[p] <= labor_available;

subject to material_limit:
    sum {p in PRODUCTS} material_required[p] * produce[p] <=
material_available;
```

Python Solution:

```
from amplpy import AMPL
import pandas as pd

# Create AMPL instance
ampl = AMPL()

# Load model
ampl.read("production.mod")

# Prepare data
products_df = pd.DataFrame({
    'profit': [50, 40],
    'labor_required': [2, 1],
    'material_required': [1, 2]
}, index=[1, 2])
products_df.index.name = 'PRODUCTS'
```

```

# Send data to AMPL
ampl.set_data(products_df, 'PRODUCTS')
ampl.get_parameter('labor_available').set(100)
ampl.get_parameter('material_available').set(80)

# Configure solver
ampl.option["solver"] = "gurobi"
ampl.option["gurobi_options"] = "outlev=1"

# Solve
ampl.solve()

# Check solve status
if ampl.solve_result == "solved":
    # Get objective value
    obj = ampl.get_objective('total_profit')
    print(f"Maximum profit: ${obj.value():.2f}")

    # Get variable values as DataFrame
    production = ampl.get_variable('produce').get_values().to_pandas()
    print("\nOptimal production plan:")
    print(production)

    # Check constraint slack
    labor_slack = ampl.get_constraint('labor_limit').slack()
    material_slack = ampl.get_constraint('material_limit').slack()
    print(f"\nUnused labor: {labor_slack:.2f} hours")
    print(f"Unused materials: {material_slack:.2f} units")
else:
    print(f"Optimization failed: {ampl.solve_result}")

```

Expected Results:

```

Gurobi 13.0.0: optimal solution; objective 2800
2 simplex iterations

```

Maximum profit: \$2800.00

Optimal production plan:

	produce.val
1	40
2	20

Unused labor: 0.00 hours

Unused materials: 0.00 units

Example 2: Mixed-Integer Programming — Facility Location

Problem Description:

Determine which facilities to open and how to allocate customers to facilities to minimize total cost (fixed facility costs plus transportation costs).

Model File (facility.mod):

```
# Sets
set FACILITIES;
set CUSTOMERS;

# Parameters
param fixed_cost {FACILITIES};
param transport_cost {FACILITIES, CUSTOMERS};
param demand {CUSTOMERS};
param capacity {FACILITIES};

# Decision Variables
# Binary: 1 if facility is open, 0 otherwise
var open {FACILITIES} binary;
# Continuous: fraction of customer demand served by facility
var allocate {FACILITIES, CUSTOMERS} >= 0, <= 1;

# Objective Function
minimize total_cost:
    sum {f in FACILITIES} fixed_cost[f] * open[f] +
    sum {f in FACILITIES, c in CUSTOMERS}
        transport_cost[f,c] * demand[c] * allocate[f,c];

# Constraints
subject to customer_demand {c in CUSTOMERS}:
    sum {f in FACILITIES} allocate[f,c] = 1;

subject to facility_capacity {f in FACILITIES}:
    sum {c in CUSTOMERS} demand[c] * allocate[f,c] <= capacity[f] * open[f];
```

Key MIP Features:

- **Binary variables:** Model yes/no facility opening decisions
- **Linking constraints:** Can only allocate to open facilities
- **Branching:** Gurobi explores different combinations of facility openings

Python Solution:

```
from amplpy import AMPL
import pandas as pd

# Create AMPL instance
ampl = AMPL()
ampl.read("facility.mod")

# Prepare facility data
facilities_df = pd.DataFrame({
```

```

    'fixed_cost': [10000, 12000, 15000],
    'capacity': [1000, 1200, 1500]
}, index=['F1', 'F2', 'F3'])
facilities_df.index.name = 'FACILITIES'

# Prepare customer data
customers_df = pd.DataFrame({
    'demand': [200, 300, 250, 400]
}, index=['C1', 'C2', 'C3', 'C4'])
customers_df.index.name = 'CUSTOMERS'

# Prepare transportation costs
transport_df = pd.DataFrame({
    'C1': [5, 7, 9],
    'C2': [8, 4, 6],
    'C3': [6, 7, 4],
    'C4': [9, 5, 3]
}, index=['F1', 'F2', 'F3'])
transport_df.index.name = 'FACILITIES'
transport_df.columns.name = 'CUSTOMERS'

# Send data to AMPL
ampl.set_data(facilities_df, 'FACILITIES')
ampl.set_data(customers_df, 'CUSTOMERS')
ampl.param['transport_cost'] = transport_df

# Configure solver for MIP
ampl.option["solver"] = "gurobi"
ampl.option["gurobi_options"] = " ".join([
    "mipgap=0.01",          # Accept 1% optimality gap
    "mipfocus=1",         # Emphasize finding good solutions quickly
    "timelimit=300"        # 5 minute time limit
])

# Solve
ampl.solve()

# Process results
if ampl.solve_result == "solved":
    print(f"Total cost: ${ampl.get_objective('total_cost').value():.2f}")

    # Which facilities are open?
    open_facilities = ampl.get_variable('open').get_values().to_pandas()
    open_facilities = open_facilities[open_facilities['open.val'] > 0.5]
    print(f"\nOpen facilities: {list(open_facilities.index)}")

    # Customer allocations
    allocations = ampl.get_variable('allocate').get_values().to_pandas()
    allocations = allocations[allocations['allocate.val'] > 0.01]
    print("\nCustomer allocations:")
    print(allocations)
else:

```

```
print(f"Optimization failed: {ampl.solve_result}")
```

Expected Results:

```
Gurobi 13.0.0: optimal solution; objective 18350
```

```
Total cost: $18350.00
```

```
Open facilities: ['F2']
```

```
Customer allocations:
```

```
          allocate.val
index0  index1
F2      C1          1
        C2          1
        C3          1
        C4          1
```

Interpreting MIP Gap:

The MIP gap indicates solution quality:

- **Gap = 0.01 (1%):** Solution is within 1% of proven optimal
- **Gap = 0.00:** Proven optimal solution
- **Larger gaps:** May want to increase time limit or adjust solver parameters

Example 3: Large-Scale Problem — Network Optimization

Performance Considerations:

For problems with millions of variables and constraints, consider:

- **Memory management:** Set appropriate node file start threshold

```
ampl.option["gurobi_options"] = "nodefilestart=8 nodefiledir=/tmp"
```

- **Output control:** Adjust log frequency to avoid I/O bottlenecks

```
ampl.option["gurobi_options"] = "outlev=1 logfreq=30"
```

- **Result management:** Write solutions to files instead of displaying

```
ampl.option["gurobi_options"] = "sol:stub=solution"
```

VI. Troubleshooting & Best Practices

Common Issues & Solutions

Installation Failures

- **Problem:** pip install fails with permission errors
 - **Solution:** Use --user flag or create virtual environment
- **Problem:** Module download fails
 - **Solution:** Check internet connection and proxy settings
- **Problem:** Platform-specific binary not available
 - **Solution:** Use Portal download method for your specific platform

License Activation Problems

- **Problem:** Invalid UUID error
 - **Solution:** Verify UUID is correctly copied (no extra spaces or characters)
- **Problem:** License expired
 - **Solution:** Contact AMPL support to renew or upgrade license
- **Problem:** License limits exceeded
 - **Solution:** Check problem size against license limitations; consider upgrading

Memory and Performance Issues

- **Problem:** Out of memory errors
 - **Solution:** Enable node files by lowering the memory threshold:

```
AMPL.option["gurobi_options"] = "nodefilestart=8"
```

- **Problem:** Slow solve times
 - **Solution:** Use tuning tool or adjust MIP focus parameters:

```
AMPL.option["gurobi_options"] = "tunebase=results tunetimelim=3600"
```

- **Problem:** Numerical instability
 - **Solution:** Scale problem coefficients, increase numericfocus (0–3)

```
AMPL.option["gurobi_options"] = "numericfocus=2"
```

Best Practices

Model Organization and File Structure

- Use descriptive names for sets, parameters, and variables
- Add comments to explain model formulation and assumptions
- Organize related models in project directories

Version Control

- Use Git or similar version control for model files
- Track changes to model formulation over time
- Document major model revisions in commit messages

Documentation Standards

- Maintain a README file describing model purpose and usage
- Document parameter units and valid ranges
- Keep a changelog of model modifications
- Include example data files for testing

Performance Benchmarking

- Record solve times for standard test cases
- Monitor solution quality metrics (gap, objective value)
- Test model scalability with increasing problem sizes
- Compare different solver option configurations

VII. Resources & Next Steps

Official Documentation

- **AMPL Documentation:** <https://ampl.com/resources>
- **AMPL Book (Free PDF):** <https://ampl.com/resources/the-ampl-book>
- **AMPLpy Documentation:** <https://amplpy.readthedocs.io>
- **Gurobi Documentation:** <https://docs.gurobi.com/projects/optimizer/en/current>
- **AMPL Gurobi Documentation:** <https://dev.ampl.com/solvers/gurobi/index.html>

Community & Support

- **AMPL Forum:** Ask questions and share knowledge with the AMPL community: <https://discuss.ampl.com>
- **AMPL Support:** support@ampl.com for technical assistance
- **Example Models:** Browse AMPL model repository for real-world examples: <https://colab.ampl.com>

Advanced Topics to Explore

- **Stochastic Programming:** Model uncertainty with scenarios
- **Robust Optimization:** Optimize under worst-case scenarios
- **Multi-Objective Optimization:** Balance competing objectives
- **Decomposition Methods:** Benders, Dantzig-Wolfe for large-scale problems
- **Custom Callbacks:** Advanced solver control for specialized algorithms

Additional Resources

- **AMPL YouTube Channel:** <https://www.youtube.com/@amplopt>
- **Gurobi Resources:** <https://www.gurobi.com/resources>

End of the Gurobi Implementation Roadmap

For additional support, [contact AMPL Optimization](#)