Updated Last
5 Months Ago

Started In
March 2013

Gurobi.jl

Gurobi.jl is a wrapper for the Gurobi Optimizer.

It has two components:

- a thin wrapper around the complete C API
- an interface to MathOptInterface

Affiliation

This wrapper is maintained by the JuMP community with help from Gurobi.

Getting help

If you are a commercial customer, please contact Gurobi directly through the Gurobi Help Center.

Otherwise, you should ask a question on the JuMP community forum. with the `gurobi` tag, or post in Gurobi's Community Forum

If you have a reproducible example of a bug, please open a GitHub issue.

License

`Gurobi.jl` is licensed under the MIT License.

The underlying solver is a closed-source commercial product for which you must obtain a license.

Free Gurobi licenses are available for academics and students.

Installation

To use Gurobi, you need a license, which you can obtain from from gurobi.com.

Once you have a license, follow Gurobi's instructions to retrieve and set up a Gurobi license.

The instructions depend on the type of license that you have obtained.

As one exception, if you have used the default installation of Gurobi.jl and the instructions call for `grbgetkey`, do:

```
import Pkg
Pkg.add("Gurobi_jll")
import Gurobi_jll
# Replace the contents xxxxx with your actual key
key = "xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx"
run(`$(Gurobi_jll.grbgetkey()) $key`)
```

Default installation

Install Gurobi as follows:

```
import Pkg
Pkg.add("Gurobi")
```

In addition to installing the Gurobi.jl package, this will also download and install the Gurobi binaries from Gurobi_jll.jl. You do not need to install Gurobi separately.

Manual installation

To opt-out of using the `Gurobi_jll` binaries, set the `GUROBI_HOME` environment variable to point to your local installation and set the `GUROBI_JL_USE_GUROBI_JLL` environment variable to `"false"`, then run `Pkg.add` and `Pkg.build`:

```
# On Windows, this might be
ENV["GUROBI_HOME"] = "C:\\Program Files\\gurobi1100\\win64"
# ... or perhaps ...
ENV["GUROBI_HOME"] = "C:\\gurobi1100\\win64"
# On Mac, this might be
ENV["GUROBI_HOME"] = "/Library/gurobi1100/macos_universal2"

# Opt-out of using Gurobi_jll
ENV["GUROBI_JL_USE_GUROBI_JLL"] = "false"

import Pkg
Pkg.add("Gurobi")
Pkg.build("Gurobi")
```

To change the location of a manual install, change the value of `GUROBI_HOME`, re-run `Pkg.build("Gurobi")`, and then re-start Julia for the change to take effect.

Use with JuMP

To use Gurobi with JuMP, use `Gurobi.Optimizer`:

```
using JuMP, Gurobi
model = Model(Gurobi.Optimizer)
set_attribute(model, "TimeLimit", 100)
set_attribute(model, "Presolve", 0)
```

MathOptInterface API

The Gurobi optimizer supports the following constraints and attributes.

List of supported objective functions:

- `MOI.ObjectiveFunction{MOI.ScalarAffineFunction{Float64}}`
- `MOI.ObjectiveFunction{MOI.ScalarQuadraticFunction{Float64}}`
- `MOI.ObjectiveFunction{MOI.VariableIndex}`
- `MOI.ObjectiveFunction{MOI.VectorAffineFunction{Float64}}`

List of supported variable types:

- `MOI.Reals`

List of supported constraint types:

- `MOI.ScalarAffineFunction{Float64}` in `MOI.EqualTo{Float64}`
- `MOI.ScalarAffineFunction{Float64}` in `MOI.GreaterThan{Float64}`
- `MOI.ScalarAffineFunction{Float64}` in `MOI.LessThan{Float64}`
- `MOI.ScalarQuadraticFunction{Float64}` in `MOI.EqualTo{Float64}`
- `MOI.ScalarQuadraticFunction{Float64}` in `MOI.GreaterThan{Float64}`
- `MOI.ScalarQuadraticFunction{Float64}` in `MOI.LessThan{Float64}`
- `MOI.VariableIndex` in `MOI.EqualTo{Float64}`
- `MOI.VariableIndex` in `MOI.GreaterThan{Float64}`
- `MOI.VariableIndex` in `MOI.Integer`
- `MOI.VariableIndex` in `MOI.Interval{Float64}`
- `MOI.VariableIndex` in `MOI.LessThan{Float64}`
- `MOI.VariableIndex` in `MOI.Semicontinuous{Float64}`
- `MOI.VariableIndex` in `MOI.Semiinteger{Float64}`
- `MOI.VariableIndex` in `MOI.ZeroOne`
- `MOI.VectorOfVariables` in `MOI.SOS1{Float64}`
- `MOI.VectorOfVariables` in `MOI.SOS2{Float64}`
- `MOI.VectorOfVariables` in `MOI.SecondOrderCone`

- `MOI.VectorAffineFunction` in `MOI.Indicator`

List of supported model attributes:

- `MOI.HeuristicCallback()`
- `MOI.LazyConstraintCallback()`
- `MOI.Name()`
- `MOI.ObjectiveSense()`
- `MOI.UserCutCallback()`

Options

See the Gurobi Documentation for a list and description of allowable parameters.

C API

The C API can be accessed via `Gurobi.GRBxx` functions, where the names and arguments are identical to the C API.

See the Gurobi documentation for details.

As general rules when converting from Julia to C:

- When Gurobi requires the column index of a variable `x`, use `Gurobi.c_column(model, x)`
- When Gurobi requires a `Ptr{T}` that holds one element, like `double *`, use a `Ref{T}()`.
- When Gurobi requires a `Ptr{T}` that holds multiple elements, use a `Vector{T}`.
- When Gurobi requires a `double`, use `Cdouble`
- When Gurobi requires an `int`, use `Cint`
- When Gurobi requires a `NULL`, use `C_NULL`

For example:

```
julia> import MathOptInterface as MOI

julia> using Gurobi

julia> model = Gurobi.Optimizer();

julia> x = MOI.add_variable(model)
MOI.VariableIndex(1)

julia> x_col = Gurobi.c_column(model, x)
0

julia> GRBupdatemodel(model)
```

```
0

julia> pValue = Ref{Cdouble}(NaN)
Base.RefValue{Float64}(NaN)

julia> GRBgetdblattrelement(model, "LB", x_col, pValue)
0

julia> pValue[]
-1.0e100

julia> GRBsetdblattrelement(model, "LB", x_col, 1.5)
0

julia> GRBupdatemodel(model)
0

julia> GRBgetdblattrelement(model, "LB", x_col, pValue)
0

julia> pValue[]
1.5
```

The C API from JuMP

You can call the C API from JuMP if you use `direct_model`. This is most useful for adding `GRBaddgenXXX` constraints. Here are some examples:

```
using JuMP, Gurobi
column(x::VariableRef) = Gurobi.c_column(backend(owner_model(x)), index(x))
model = direct_model(Gurobi.Optimizer())
@variable(model, x)
@variable(model, y)
p = [3.0, 0.0, 0.0, 7.0, 3.0]
GRBaddgenconstrPoly(backend(model), C_NULL, column(x), column(y), 5, p, "")
optimize!(model)
```

```
using JuMP, Gurobi
column(x::VariableRef) = Gurobi.c_column(backend(owner_model(x)), index(x))
model = direct_model(Gurobi.Optimizer())
@variable(model, x[i in 1:2])
@variable(model, y[1:2])
GRBaddgenconstrPow(backend(model), "x1^0.7", column(x[1]), column(y[1]), 0.7, "")
GRBaddgenconstrPow(backend(model), "x2^3", column(x[2]), column(y[2]), 3.0, "")
@objective(model, Min, y[1] + y[2])
optimize!(model)
```

Reusing the same Gurobi environment for multiple solves

When using this package via other packages such as JuMP.jl, the default behavior is to obtain a new Gurobi license token every time a model is created. If you are using Gurobi in a setting where the number of concurrent Gurobi uses is limited (for example, "Single-Use" or "Floating-Use" licenses), you might instead prefer to obtain a single license token that is shared by all models that your program solves.

You can do this by passing a `Gurobi.Env()` object as the first parameter to `Gurobi.Optimizer`. For example:

```
using JuMP, Gurobi
const GRB_ENV = Gurobi.Env()

model_1 = Model(() -> Gurobi.Optimizer(GRB_ENV))

# The solvers can have different options too
model_2 = direct_model(Gurobi.Optimizer(GRB_ENV))
set_attribute(model_2, "OutputFlag", 0)
```

If you create a module with a `Gurobi.Env` as a module-level constant, use an `__init__` function to ensure that a new environment is created each time the module is loaded:

```
module MyModule

import Gurobi

const GRB_ENV_REF = Ref{Gurobi.Env}()

function __init__()
    global GRB_ENV_REF
    GRB_ENV_REF[] = Gurobi.Env()
    return
end

# Note the need for GRB_ENV_REF[] not GRB_ENV_REF
create_optimizer() = Gurobi.Optimizer(GRB_ENV_REF[])

end
```

Accessing Gurobi-specific attributes

Get and set Gurobi-specific variable, constraint, and model attributes as follows:

```
using JuMP, Gurobi
model = direct_model(Gurobi.Optimizer())
```

```
@variable(model, x >= 0)
@constraint(model, c, 2x >= 1)
@objective(model, Min, x)
grb = backend(model)
MOI.set(grb, Gurobi.ConstraintAttribute("Lazy"), index(c), 2)
optimize!(model)
MOI.get(grb, Gurobi.VariableAttribute("LB"), index(x))  # Returns 0.0
MOI.get(grb, Gurobi.ModelAttribute("NumConstrs")) # Returns 1
```

A complete list of supported Gurobi attributes can be found in their online documentation.

Callbacks

Here is an example using Gurobi's solver-specific callbacks.

```
using JuMP, Gurobi, Test

model = direct_model(Gurobi.Optimizer())
@variable(model, 0 <= x <= 2.5, Int)
@variable(model, 0 <= y <= 2.5, Int)
@objective(model, Max, y)
cb_calls = Cint[]
function my_callback_function(cb_data, cb_where::Cint)
    # You can reference variables outside the function as normal
    push!(cb_calls, cb_where)
    # You can select where the callback is run
    if cb_where != GRB_CB_MIPSOL && cb_where != GRB_CB_MIPNODE
        return
    end
    # You can query a callback attribute using GRBcbget
    if cb_where == GRB_CB_MIPNODE
        resultP = Ref{Cint}()
        GRBcbget(cb_data, cb_where, GRB_CB_MIPNODE_STATUS, resultP)
        if resultP[] != GRB_OPTIMAL
            return  # Solution is something other than optimal.
        end
    end
    # Before querying `callback_value`, you must call:
    Gurobi.load_callback_variable_primal(cb_data, cb_where)
    x_val = callback_value(cb_data, x)
    y_val = callback_value(cb_data, y)
    # You can submit solver-independent MathOptInterface attributes such as
    # lazy constraints, user-cuts, and heuristic solutions.
    if y_val - x_val > 1 + 1e-6
        con = @build_constraint(y - x <= 1)
        MOI.submit(model, MOI.LazyConstraint(cb_data), con)
    elseif y_val + x_val > 3 + 1e-6
        con = @build_constraint(y + x <= 3)
        MOI.submit(model, MOI.LazyConstraint(cb_data), con)
    end
```

```
        if rand() < 0.1
            # You can terminate the callback as follows:
            GRBterminate(backend(model))
        end
        return
    end
    # You _must_ set this parameter if using lazy constraints.
    MOI.set(model, MOI.RawOptimizerAttribute("LazyConstraints"), 1)
    MOI.set(model, Gurobi.CallbackFunction(), my_callback_function)
    optimize!(model)
    @test termination_status(model) == MOI.OPTIMAL
    @test primal_status(model) == MOI.FEASIBLE_POINT
    @test value(x) == 1
    @test value(y) == 2
```

See the Gurobi documentation for other information that can be queried with `GRBcbget`.

Common Performance Pitfall with JuMP

Gurobi's API works differently than most solvers. Any changes to the model are not applied immediately, but instead go sit in a internal buffer (making any modifications appear to be instantaneous) waiting for a call to `GRBupdatemodel` (where the work is done).

This leads to a common performance pitfall that has the following message as its main symptom:

```
Warning: excessive time spent in model updates. Consider calling update less frequently.
```

This often means the JuMP program was structured in such a way that Gurobi.jl ends up calling `GRBupdatemodel` in each iteration of a loop.

Usually, it is possible (and easy) to restructure the JuMP program in a way it stays ssolver-agnostic and has a close-to-ideal performance with Gurobi.

To guide such restructuring it is good to keep in mind the following bits of information:

1. `GRBupdatemodel` is only called if changes were done since last `GRBupdatemodel` (that is, if the internal buffer is not empty).
2. `GRBupdatemodel` is called when `JuMP.optimize!` is called, but this often is not the source of the problem.
3. `GRBupdatemodel` *may* be called when *any* model attribute is queried, *even if* that specific attribute was not changed. This often the source of the problem.

The worst-case scenario is, therefore, a loop of modify-query-modify-query, even if what is being modified and what is being queried are two completely distinct things.

As an example, instead of:

```
model = Model(Gurobi.Optimizer)
@variable(model, x[1:100] >= 0)
for i in 1:100
    set_upper_bound(x[i], i)
    # `GRBupdatemodel` called on each iteration of this loop.
    println(lower_bound(x[i]))
end
```

do

```
model = Model(Gurobi.Optimizer)
@variable(model, x[1:100] >= 0)
# All modifications are done before any queries.
for i in 1:100
    set_upper_bound(x[i], i)
end
for i in 1:100
    # Only the first `lower_bound` query may trigger an `GRBupdatemodel`.
    println(lower_bound(x[i]))
end
```

Common errors

Using Gurobi v9.0 and you got an error like `Q not PSD`?

You need to set the NonConvex parameter:

```
model = Model(Gurobi.Optimizer)
set_optimizer_attribute(model, "NonConvex", 2)
```

Gurobi Error 1009: Version number is XX.X, license is for version XX.X

Make sure that your license is correct for your Gurobi version. See the Gurobi documentation for details.

Once you are sure that the license and Gurobi versions match, re-install Gurobi.jl by running:

```
import Pkg
Pkg.build("Gurobi")
```

Required Packages

- BenchmarkTools