JuMP

The JuMP core developers and contributors

October 28, 2021

Contents

Part I

Introduction

Chapter 1

Introduction

Welcome to the documentation for JuMP!

1.1 What is JuMP?

JuMP is a domain-specific modeling language for mathematical optimization embedded in Julia. It currently supports a number of open-source and commercial solvers for a variety of problem classes, including linear, mixed-integer, second-order conic, semidefinite, and nonlinear programming.

> **Tip**
>
> If you aren't sure if you should use JuMP, read Should I use JuMP?.

1.2 Resources for getting started

There are few ways to get started with JuMP:

- Read the Installation Guide.
- Read the introductory tutorials Getting started with Julia and Getting started with JuMP.
- Browse some of our modeling tutorials, including classics such as The diet problem, or the Maximum likelihood estimation problem using nonlinear programming.

> **Tip**
>
> Need help? Join the community forum to search for answers to commonly asked questions.
>
> Before asking a question, make sure to read the post make it easier to help you, which contains a number of tips on how to ask a good question.

1.3 How the documentation is structured

Having a high-level overview of how this documentation is structured will help you know where to look for certain things.

- **Tutorials** contain worked examples of solving problems with JuMP. Start here if you are new to JuMP, or you have a particular problem class you want to model.

- The **Manual** contains short code-snippets that explain how to achieve specific tasks in JuMP. Look here if you want to know how to achieve a particular task, such as how to Delete a variable or how to Modify an objective coefficient.
- The **API Reference** contains a complete list of the functions you can use in JuMP. Look here if you want to know how to use a particular function.
- The **Background information** section contains background reading material to provide context to JuMP. Look here if you want an understanding of what JuMP is and why we created it, rather than how to use it.
- The **Developer docs** section contains information for people contributing to JuMP development or writing JuMP extensions. Don't worry about this section if you are just using JuMP to formulate and solve problems as a user.
- The **MathOptInterface** section is a self-contained copy of the documentation for MathOptInterface. Look here for functions and constants beginning with `MOI.`, as well as for general information on how MathOptInterface works.

1.4 Citing JuMP

If you find JuMP useful in your work, we kindly request that you cite the following paper (pdf):

```
@article{DunningHuchetteLubin2017,
author = {Iain Dunning and Joey Huchette and Miles Lubin},
title = {JuMP: A Modeling Language for Mathematical Optimization},
journal = {SIAM Review},
volume = {59},
number = {2},
pages = {295-320},
year = {2017},
doi = {10.1137/15M1020575},
}
```

For an earlier work where we presented a prototype implementation of JuMP, see here:

```
@article{LubinDunningIJOC,
author = {Miles Lubin and Iain Dunning},
title = {Computing in Operations Research Using Julia},
journal = {INFORMS Journal on Computing},
volume = {27},
number = {2},
pages = {238-248},
year = {2015},
doi = {10.1287/ijoc.2014.0623},
}
```

A preprint of this paper is freely available.

1.5 NumFOCUS

JuMP is a Sponsored Project of NumFOCUS, a 501(c)(3) nonprofit charity in the United States. NumFOCUS provides JuMP with fiscal, legal, and administrative support to help ensure the health and sustainability of the project. Visit numfocus.org for more information.

You can support JuMP by donating.

Figure 1.1: NumFOCUS logo

Donations to JuMP are managed by NumFOCUS. For donors in the United States, your gift is tax-deductible to the extent provided by law. As with any donation, you should consult with your tax adviser about your particular tax situation.

JuMP's largest expense is the annual JuMP-dev workshop. Donations will help us provide travel support for JuMP-dev attendees and take advantage of other opportunities that arise to support JuMP development.

Chapter 2

Installation Guide

This guide explains how to install Julia and JuMP. If you have installation troubles, read the Common installation issues section below.

2.1 Install Julia

JuMP is a package for Julia. To use JuMP, first download and install Julia.

> **Tip**
>
> If you are new to Julia, read our Getting started with Julia tutorial.

Which version should I pick?

You can install the "Current stable release" or the "Long-term support (LTS) release".

- The "Current stable release" is the latest release of Julia. It has access to newer features, and is likely faster.
- The "Long-term support release" is an older version of Julia that has continued to receive bug and security fixes. However, it may not have the latest features or performance improvements.

For most users, you should install the "Current stable release", and whenever Julia releases a new version of the current stable release, you should update your version of Julia. Note that any code you write on one version of the current stable release will continue to work on all subsequent releases.

For users in restricted software environments (e.g., your enterprise IT controls what software you can install), you may be better off installing the long-term support release because you will not have to update Julia as frequently.

2.2 Install JuMP

From Julia, JuMP is installed using the built-in package manager:

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

Tip

We recommend you create a Pkg environment for each project you use JuMP for, instead of adding lots of packages to the global environment. The Pkg manager documentation has more information on this topic.

When we release a new version of JuMP, you can update with:

```
import Pkg
Pkg.update("JuMP")
```

2.3 Install a solver

JuMP depends on solvers to solve optimization problems. Therefore, you will need to install one before you can solve problems with JuMP.

Install a solver using the Julia package manager, replacing `"Clp"` by the Julia package name as appropriate.

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

Once installed, you can use Clp as a solver with JuMP as follows, using `set_optimizer_attributes` to set solver-specific options:

```
using JuMP
using Clp
model = Model(Clp.Optimizer)
set_optimizer_attributes(model, "LogLevel" => 1, "PrimalTolerance" => 1e-7)
```

Note

Most packages follow the `ModuleName.Optimizer` naming convention, but exceptions may exist. See the README of the Julia package's Github repository for more details on how to use a particular solver, including any solver-specific options.

Supported solvers

Most solvers are not written in Julia, and some require commercial licenses to use, so installation is often more complex.

- If a solver has `Manual` in the `Installation` column, the solver requires a manual installation step, such as downloading and installing a binary, or obtaining a commercial license. Consult the README of the relevant Julia package for more information.
- If the solver has `Manual`^{M} in the `Installation` column, the solver requires an installation of MATLAB.
- If the `Installation` column is missing an entry, installing the Julia package will download and install any relevant solver binaries automatically, and you shouldn't need to do anything other than `Pkg.add`.

Solvers with a missing entry in the `Julia Package` column are written in Julia. The link in the `Solver` column is the corresponding Julia package.

Where:

Solver	Julia Package	Installation	License	Supports
Alpine.jl			Triad NS	(MI)NLP
Artelys Knitro	KNITRO.jl	Manual	Comm.	(MI)LP, (MI)SOCP, (MI)NLP
BARON	BARON.jl	Manual	Comm.	(MI)NLP
Bonmin	AmplNLWriter.jl		EPL	(MI)NLP
Cbc	Cbc.jl		EPL	(MI)LP
CDCS	CDCS.jl	Manual[M]	GPL	LP, SOCP, SDP
CDD	CDDLib.jl		GPL	LP
Clp	Clp.jl		EPL	LP
COSMO.jl			Apache	LP, QP, SOCP, SDP
Couenne	AmplNLWriter.jl		EPL	(MI)NLP
CPLEX	CPLEX.jl	Manual	Comm.	(MI)LP, (MI)SOCP
CSDP	CSDP.jl		EPL	LP, SDP
EAGO.jl			MIT	NLP
ECOS	ECOS.jl		GPL	LP, SOCP
FICO Xpress	Xpress.jl	Manual	Comm.	(MI)LP, (MI)SOCP
GLPK	GLPK.jl		GPL	(MI)LP
Gurobi	Gurobi.jl	Manual	Comm.	(MI)LP, (MI)SOCP
HiGHS	HiGHS.jl		MIT	LP
Hypatia.jl			MIT	LP, SOCP, SDP
Ipopt	Ipopt.jl		EPL	LP, QP, NLP
Juniper.jl			MIT	(MI)SOCP, (MI)NLP
MadNLP.jl			MIT	LP, QP, NLP
MOSEK	MosekTools.jl	Manual	Comm.	(MI)LP, (MI)SOCP, SDP
NLopt	NLopt.jl		GPL	LP, QP, NLP
OSQP	OSQP.jl		Apache	LP, QP
PATH	PATHSolver.jl		MIT	MCP
Pavito.jl			MPL-2	(MI)NLP
ProxSDP.jl			MIT	LP, SOCP, SDP
SCIP	SCIP.jl		ZIB	(MI)LP, (MI)NLP
SCS	SCS.jl		MIT	LP, SOCP, SDP
SDPA	SDPA.jl, SDPAFamily.jl		GPL	LP, SDP
SDPNAL	SDPNAL.jl	Manual[M]	CC BY-SA	LP, SDP
SDPT3	SDPT3.jl	Manual[M]	GPL	LP, SOCP, SDP
SeDuMi	SeDuMi.jl	Manual[M]	GPL	LP, SOCP, SDP
Tulip.jl			MPL-2	LP

- LP = Linear programming
- QP = Quadratic programming
- SOCP = Second-order conic programming (including problems with convex quadratic constraints and/or objective)
- MCP = Mixed-complementarity programming
- NLP = Nonlinear programming
- SDP = Semidefinite programming
- (MI)XXX = Mixed-integer equivalent of problem type XXX

Note

Developed a solver or solver wrapper? This table is open for new contributions! Start by making a pull request to edit the installation.md file.

Note

Developing a solver or solver wrapper? See Models and the MathOptInterface docs for more details on how JuMP interacts with solvers. Please get in touch via the Developer Chatroom with any questions about connecting new solvers with JuMP.

2.4 AMPL-based solvers

Use AmplNLWriter to access solvers that support the nl format.

Some solvers, such as Bonmin and Couenne can be installed via the Julia package manager. Others need to be manually installed.

Consult the AMPL documentation for a complete list of supported solvers.

2.5 GAMS-based solvers

Use GAMS.jl to access solvers available through GAMS. Such solvers include: AlphaECP, Antigone, BARON, CONOPT, Couenne, LocalSolver, PATHNLP, SHOT, SNOPT, SoPlex. See a complete list here.

Note

GAMS.jl requires an installation of the commercial software GAMS for which a free community license exists.

2.6 NEOS-based solvers

Use NEOSServer.jl to access solvers available through the NEOS Server.

2.7 Previously supported solvers

The following solvers were compatible with JuMP up to release 0.18 but are not yet compatible with the latest version because they do not implement the new MathOptInterface API:

- Pajarito

Please join the Developer Chatroom if you have interest in reviving a previously supported solver.

2.8 Common installation issues

Tip

When in doubt, run `import Pkg; Pkg.update()` to see if updating your packages fixes the issue. Remember you will need to exit Julia and start a new session for the changes to take effect.

Check the version of your packages

Each package is versioned with a three-part number of the form `vX.Y.Z`. You can check which versions you have installed with `import Pkg; Pkg.status()`.

This should almost always be the most-recent release. You can check the releases of a package by going to the relevant Github page, and navigating to the "releases" page. For example, the list of JuMP releases is available at: https://github.com/jump-dev/JuMP.jl/releases.

If you post on the community forum, please include the output of `Pkg.status()`!

Unsatisfiable requirements detected

Did you get an error like `Unsatisfiable requirements detected for package JuMP`? The Pkg documentation has a section on how to understand and manage these conflicts.

Installing new packages can make JuMP downgrade to an earlier version

Another common complaint is that after adding a new package, code that previously worked no longer works.

This usually happens because the new package is not compatible with the latest version of JuMP. Therefore, the package manager rolls-back JuMP to an earlier version! Here's an example.

First, we add JuMP:

```
(jump_example) pkg> add JuMP
  Resolving package versions...
Updating `~/jump_example/Project.toml`
  [4076af6c] + JuMP v0.21.5
Updating `~/jump_example/Manifest.toml`
  ... lines omitted ...
```

The `+ JuMP v0.21.5` line indicates that JuMP has been added at version `0.21.5`. However, watch what happens when we add JuMPeR:

```
(jump_example) pkg> add JuMPeR
  Resolving package versions...
Updating `~/jump_example/Project.toml`
  [4076af6c] ↓ JuMP v0.21.5 ⇒ v0.18.6
  [707a9f91] + JuMPeR v0.6.0
Updating `~/jump_example/Manifest.toml`
  ... lines omitted ...
```

JuMPeR gets added at version `0.6.0` (`+ JuMPeR v0.6.0`), but JuMP gets downgraded from `0.21.5` to `0.18.6` (`↓ JuMP v0.21.5 ⇒ v0.18.6`)! The reason for this is that JuMPeR doesn't support a version of JuMP newer than `0.18.6`.

> **Tip**
>
> Pay careful attention to the output of the package manager when adding new packages, especially when you see a package being downgraded!

Part II

Tutorials

Chapter 3

Getting started

3.1 Getting started with Julia

Since JuMP is embedded in Julia, knowing some basic Julia is important for learning JuMP.

> **Tip**
>
> This tutorial is designed to provide a minimalist crash course in the basics of Julia. You can find resources that provide a more comprehensive introduction to Julia here.

Where to get help

- Read the documentation
  - JuMP https://jump.dev/JuMP.jl/stable/
  - Julia https://docs.julialang.org/en/v1/
- Ask (or browse) the Julia community forum: https://discourse.julialang.org
  - If the question is JuMP-related, ask in the Optimization (Mathematical) section, or tag your question with "jump"

To access the built-in help at the REPL, type ?, followed by the name of the function to lookup:

```
help?> help
search: help schedule Channel hasfield check_belongs_to_model @threadcall AbstractChannel
↪  searchsortedlast

 Welcome to Julia 1.6.2. The full manual is available at

 https://docs.julialang.org

 as well as many great tutorials and learning resources:

 https://julialang.org/learning/

 For help on a specific function or macro, type ? followed by its name, e.g. ?cos, or ?@time, and
 ↪  press enter. Type ; to enter shell mode, ] to enter package mode.
```

Installing Julia

To install Julia, download the latest stable release, then follow the platform specific install instructions.

> **Tip**
>
> Unless you know otherwise, you probably want the 64-bit version.

Next, you need an IDE to develop in. VS Code is a popular choice, so follow these install instructions.

Numbers and arithmetic

Since we want to solve optimization problems, we're going to be using a lot of math. Luckily, Julia is great for math, with all the usual operators:

```
@show 1 + 1
@show 1 - 2
@show 2 * 2
@show 4 / 5
@show 3^2
```

```
1 + 1 = 2
1 - 2 = -1
2 * 2 = 4
4 / 5 = 0.8
3 ^ 2 = 9
```

> **Info**
>
> The `@` in front of something indicates that it is a macro, which is just a special type of function. In this case, `@show` prints the expression as typed (e.g., `1 - 2`), as well as the evaluation of the expression (`-1`).

Did you notice how Julia didn't print `.0` after some of the numbers? Julia is a dynamic language, which means you never have to explicitly declare the type of a variable. However, in the background, Julia is giving each variable a type. Check the type of something using the `typeof` function:

```
@show typeof(1)
@show typeof(1.0)
```

```
typeof(1) = Int64
typeof(1.0) = Float64
```

Here `1` is an `Int64`, which is an integer with 64 bits of precision, and `1.0` is a `Float64`, which is a floating point number with 64-bits of precision.

> **Tip**
>
> If you aren't familiar with floating point numbers, make sure to read the Floating point numbers section.

We create complex numbers using `im`:

```
x = 2 + 1im
@show real(x)
@show imag(x)
@show typeof(x)
@show x * (1 - 2im)
```

```
real(x) = 2
imag(x) = 1
typeof(x) = Complex{Int64}
x * (1 - 2im) = 4 - 3im
```

> **Info**
>
> The curly brackets surround what we call the parameters of a type. You can read `Complex{Int64}` as "a complex number, where the real and imaginary parts are represented by `Int64`." If we call `typeof(1.0 + 2.0im)` it will be `Complex{Float64}`, which a complex number with the parts represented by `Float64`.

There are also some cool things like an irrational representation of π.

```
π
```

```
π
 = 3.1415926535897...
```

> **Tip**
>
> To make π (and most other Greek letters), type `\pi` and then press `[TAB]`.

```
typeof(π)
```

```
Irrationalπ{:}
```

However, if we do math with irrational numbers, they get converted to `Float64`:

```
typeof(2π / 3)
```

```
Float64
```

Floating point numbers

> **Warning**
>
> If you aren't familiar with floating point numbers, make sure to read this section carefully.

A `Float64` is a floating point approximation of a real number using 64-bits of information.

Because it is an approximation, things we know hold true in mathematics don't hold true in a computer! For example:

```
0.1 * 3 == 0.3
```

```
false
```

```
sin(2π / 3) == √3 / 2
```

```
false
```

> **Tip**
>
> Get √ by typing `\sqrt` then press `[TAB]`.

Let's see what the differences are:

```
0.1 * 3 - 0.3
```

```
5.551115123125783e-17
```

```
sin(2π / 3) - √3 / 2
```

```
1.1102230246251565e-16
```

They are small, but not zero!

One way of explaining this difference is to consider how we would write `1 / 3` and `2 / 3` using only four digits after the decimal point. We would write `1 / 3` as `0.3333`, and `2 / 3` as `0.6667`. So, despite the fact that `2 * (1 / 3) == 2 / 3`, `2 * 0.3333 == 0.6666 != 0.6667`.

Let's try that again using ≈ (`\approx + [TAB]`) instead of ==:

```
0.1 * 3 ≈ 0.3
```

```
true
```

```
sin(2π / 3) ≈ √3 / 2
```

```
true
```

≈ is just a clever way of calling the `isapprox` function:

```
isapprox(sin(2π / 3), √3 / 2; atol = 1e-8)
```

```
true
```

> **Warning**
>
> Floating point is the reason solvers use tolerances when they solve optimization models. A common mistake you're likely to make is checking whether a binary variable is 0 using `value(z) == 0`. Always remember to use something like `isapprox` when comparing floating point numbers.

Note that `isapprox` will always return `false` if one of the number being compared is `0` and `atol` is zero (its default value).

```
1e-300 ≈ 0.0
```

```
false
```

so always set a nonzero value of `atol` if one of the arguments can be zero.

```
isapprox(1e-9, 0.0, atol = 1e-8)
```

```
true
```

> **Tip**
>
> Gurobi has a good series of articles on the implications of floating point in optimization if you want to read more.

If you aren't careful, floating point arithmetic can throw up all manner of issues. For example:

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

```
true
```

It even turns out that floating point numbers aren't associative!

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

```
false
```

It's important to note that this issue isn't Julia-specific. It happens in every programming language (try it out in Python).

Vectors, matrices and arrays

Similar to Matlab, Julia has native support for vectors, matrices and tensors; all of which are represented by arrays of different dimensions. Vectors are constructed by comma-separated elements surrounded by square brackets:

```
b = [5, 6]
```

```
2-element Vector{Int64}:
 5
 6
```

Matrices can by constructed with spaces separating the columns, and semicolons separating the rows:

```
A = [1.0 2.0; 3.0 4.0]
```

```
2×2 Matrix{Float64}:
 1.0  2.0
 3.0  4.0
```

We can do linear algebra:

```
x = A \ b
```

```
2-element Vector{Float64}:
 -3.9999999999999987
  4.499999999999999
```

> **Info**
>
> Here is floating point at work again! `x` is approximately `[-4, 4.5]`.

```
A * x
```

```
2-element Vector{Float64}:
 5.0
 6.0
```

```
A * x ≈ b
```

```
true
```

Note that when multiplying vectors and matrices, dimensions matter. For example, you can't multiply a vector by a vector:

```
    b * b
```

```
MethodError: no method matching *(::Vector{Int64}, ::Vector{Int64})
Closest candidates are:
  *(::Any, ::Any, !Matched::Any, !Matched::Any...) at operators.jl:560
  *(!Matched::StridedMatrix{T}, ::StridedVector{S}) where {T<:Union{Float32, Float64, ComplexF32,
     ComplexF64}, S<:Real} at /buildworker/worker/package_linux64/build/usr/share/julia/stdlib/v1.6/
     LinearAlgebra/src/matmul.jl:44
  *(::StridedVecOrMat{T} where T, !Matched::LinearAlgebra.Adjoint{var"#s814", var"#s813"} where {var
     "#s814", var"#s813"<:LinearAlgebra.LQPackedQ}) at /buildworker/worker/package_linux64/build/usr/
     share/julia/stdlib/v1.6/LinearAlgebra/src/lq.jl:254
  ...
```

But multiplying transposes works:

```
b' * b
```

```
61
```

```
b * b'
```

```
2×2 Matrix{Int64}:
 25  30
 30  36
```

Other common types

Strings

Double quotes are used for strings:

```
typeof("This is Julia")
```

```
String
```

Unicode is fine in strings:

```
typeof("π is about 3.1415")
```

```
String
```

Use `println` to print a string:

```
println("Hello, World!")
```

```
Hello, World!
```

Use `$()` to interpolate values into a string:

```
x = 123
println("The value of x is: $(x)")
```

```
The value of x is: 123
```

Symbols

Julia `Symbols` are a data structure from the compiler that represent Julia identifiers (i.e., variable names).

```
println("The value of x is: $(eval(:x))")
```

```
The value of x is: 123
```

> **Tip**
>
> We used `eval` here to demonstrate how Julia links `Symbols` to variables. However, avoid calling `eval` in your code. It is usually a sign that your code is doing something that could be more easily achieved a different way. The Community Forum is a good place to ask for advice on alternative approaches.

```
typeof(:x)
```

```
Symbol
```

You can think of a `Symbol` as a `String` that takes up less memory, and that can't be modified.

Convert between `String` and `Symbol` using their constructors:

```
String(:abc)
```

```
"abc"
```

```
Symbol("abc")
```

```
:abc
```

> **Tip**
>
> `Symbol`s are often (ab)used to stand in for a `String` or an `Enum`, when one of the former is likely a better choice. The JuMP Style guide recommends reserving `Symbol`s for identifiers. See @enum vs. Symbol for more.

Tuples

Julia makes extensive use of a simple data structure called Tuples. Tuples are immutable collections of values. For example:

```
t = ("hello", 1.2, :foo)
```

```
("hello", 1.2, :foo)
```

```
typeof(t)
```

```
Tuple{String, Float64, Symbol}
```

Tuples can be accessed by index, similar to arrays:

```
t[2]
```

```
1.2
```

And they can be "unpacked" like so:

```
a, b, c = t
b
```

```
1.2
```

The values can also be given names, which is a convenient way of making light-weight data structures.

```
t = (word = "hello", num = 1.2, sym = :foo)
```

```
(word = "hello", num = 1.2, sym = :foo)
```

Values can be accessed using dot syntax:

```
t.word
```

```
"hello"
```

Dictionaries

Similar to Python, Julia has native support for dictionaries. Dictionaries provide a very generic way of mapping keys to values. For example, a map of integers to strings:

```
d1 = Dict(1 => "A", 2 => "B", 4 => "D")
```

```
Dict{Int64, String} with 3 entries:
  4 => "D"
  2 => "B"
  1 => "A"
```

> **Info**
>
> Type-stuff again: `Dict{Int64,String}` is a dictionary with `Int64` keys and `String` values.

Looking up a values uses the bracket syntax:

```
d1[2]
```

```
"B"
```

Dictionaries support non-integer keys and can mix data types:

```
Dict("A" => 1, "B" => 2.5, "D" => 2 - 3im)
```

```
Dict{String, Number} with 3 entries:
  "B" => 2.5
  "A" => 1
  "D" => 2-3im
```

> **Info**
>
> Julia types form a hierarchy. Here the value type of the dictionary is `Number`, which is a generalization of `Int64`, `Float64`, and `Complex{Int}`. Leaf nodes in this hierarchy are called "concrete" types, and all others are called "Abstract". In general, having variables with abstract types like `Number` can lead to slower code, so you should try to make sure every element in a dictionary or vector is the same type. For example, in this case we could represent every element as a `Complex{Float64}`:

```
Dict("A" => 1.0 + 0.0im, "B" => 2.5 + 0.0im, "D" => 2.0 - 3.0im)
```

```
Dict{String, ComplexF64} with 3 entries:
  "B" => 2.5+0.0im
  "A" => 1.0+0.0im
  "D" => 2.0-3.0im
```

Dictionaries can be nested:

```
d2 = Dict("A" => 1, "B" => 2, "D" => Dict(:foo => 3, :bar => 4))
```

```
Dict{String, Any} with 3 entries:
  "B" => 2
  "A" => 1
  "D" => Dict(:bar=>4, :foo=>3)
```

```
d2["B"]
```

```
2
```

```
d2["D"][:foo]
```

```
3
```

Structs

You can define custom datastructures with `struct`:

```
struct MyStruct
    x::Int
    y::String
    z::Dict{Int,Int}
end

a = MyStruct(1, "a", Dict(2 => 3))
```

```
Main.MyStruct(1, "a", Dict(2 => 3))
```

```
a.x
```

```
1
```

By default, these are not mutable

```
    a.x = 2
```

```
setfield! immutable struct of type MyStruct cannot be changed
```

However, you can declare a `mutable struct` which is mutable:

```
mutable struct MyStructMutable
    x::Int
    y::String
    z::Dict{Int,Int}
end

a = MyStructMutable(1, "a", Dict(2 => 3))
a.x
```

```
1
```

```
a.x = 2

a
```

```
Main.MyStructMutable(2, "a", Dict(2 => 3))
```

Loops

Julia has native support for for-each style loops with the syntax `for <value> in <collection> end`:

```
for i in 1:5
    println(i)
end
```

```
1
2
3
4
5
```

> **Info**
>
> Ranges are constructed as `start:stop`, or `start:step:stop`.

```
for i in 1.2:1.1:5.6
    println(i)
end
```

```
1.2
2.3
3.4
4.5
5.6
```

This for-each loop also works with dictionaries:

```
for (key, value) in Dict("A" => 1, "B" => 2.5, "D" => 2 - 3im)
    println("$(key): $(value)")
end
```

```
B: 2.5
A: 1
D: 2 - 3im
```

Note that in contrast to vector languages like Matlab and R, loops do not result in a significant performance degradation in Julia.

Control Flow

Julia control flow is similar to Matlab, using the keywords `if-elseif-else-end`, and the logical operators `||` and && for **or** and **and** respectively:

```
for i in 0:5:15
    if i < 5
        println("$(i) is less than 5")
    elseif i < 10
        println("$(i) is less than 10")
    else
        if i == 10
            println("the value is 10")
        else
            println("$(i) is bigger than 10")
        end
    end
end
```

```
0 is less than 5
5 is less than 10
the value is 10
15 is bigger than 10
```

Comprehensions

Similar to languages like Haskell and Python, Julia supports the use of simple loops in the construction of arrays and dictionaries, called comprehensions.

A list of increasing integers:

```
[i for i in 1:5]
```

```
5-element Vector{Int64}:
 1
 2
 3
 4
 5
```

Matrices can be built by including multiple indices:

```
[i * j for i in 1:5, j in 5:10]
```

```
5×6 Matrix{Int64}:
  5   6   7   8   9  10
 10  12  14  16  18  20
 15  18  21  24  27  30
 20  24  28  32  36  40
 25  30  35  40  45  50
```

Conditional statements can be used to filter out some values:

```
[i for i in 1:10 if i % 2 == 1]
```

```
5-element Vector{Int64}:
 1
 3
 5
 7
 9
```

A similar syntax can be used for building dictionaries:

```
Dict("$(i)" => i for i in 1:10 if i % 2 == 1)
```

```
Dict{String, Int64} with 5 entries:
  "1" => 1
  "5" => 5
  "7" => 7
  "9" => 9
  "3" => 3
```

Functions

A simple function is defined as follows:

```
function print_hello()
    return println("hello")
end
print_hello()
```

```
hello
```

Arguments can be added to a function:

```
function print_it(x)
    return println(x)
end
print_it("hello")
print_it(1.234)
print_it(:my_id)
```

```
hello
1.234
my_id
```

Optional keyword arguments are also possible:

```
function print_it(x; prefix = "value:")
    return println("$(prefix) $(x)")
end
print_it(1.234)
print_it(1.234, prefix = "val:")
```

```
value: 1.234
val: 1.234
```

The keyword `return` is used to specify the return values of a function:

```
function mult(x; y = 2.0)
    return x * y
end

mult(4.0)
```

```
8.0
```

```
mult(4.0, y = 5.0)
```

```
20.0
```

Anonymous functions

The syntax `input -> output` creates an anonymous function. These are most useful when passed to other functions. For example:

```
f = x -> x^2
f(2)
```

```
4
```

```
map(x -> x^2, 1:4)
```

```
4-element Vector{Int64}:
  1
  4
  9
 16
```

Type parameters

We can constrain the inputs to a function using type parameters, which are `::` followed by the type of the input we want. For example:

```
function foo(x::Int)
    return x^2
end

function foo(x::Float64)
    return exp(x)
end

function foo(x::Number)
    return x + 1
```

```
end

@show foo(2)
@show foo(2.0)
@show foo(1 + 1im)
```

```
foo(2) = 4
foo(2.0) = 7.38905609893065
foo(1 + 1im) = 2 + 1im
```

But what happens if we call `foo` with something we haven't defined it for?

```
    foo([1, 2, 3])
```

```
MethodError: no method matching foo(::Vector{Int64})
Closest candidates are:
  foo(!Matched::Int64) at getting_started_with_julia.md:637
  foo(!Matched::Float64) at getting_started_with_julia.md:641
  foo(!Matched::Number) at getting_started_with_julia.md:645
```

We get a dreaded `MethodError`! A `MethodError` means that you passed a function something that didn't match the type that it was expecting. In this case, the error message says that it doesn't know how to handle an `Vector{Int64}`, but it does know how to handle `Float64`, `Int64`, and `Number`.

> **Tip**
>
> Read the "Closest candidates" part of the error message carefully to get a hint as to what was expected.

Broadcasting

In the example above, we didn't define what to do if `f` was passed a `Vector`. Luckily, Julia provides a convenient syntax for mapping `f` element-wise over arrays! Just add a `.` between the name of the function and the opening `(`. This works for any function, including functions with multiple arguments. For example:

```
f.([1, 2, 3])
```

```
3-element Vector{Int64}:
 1
 4
 9
```

> **Tip**
>
> Get a `MethodError` when calling a function that takes a `Vector`, `Matrix`, or `Array`? Try broadcasting it!

Mutable vs immutable objects

Some types in Julia are mutable, which means you can change the values inside them. A good example is an array. You can modify the contents of an array without having to make a new array.

In contrast, types like `Float64` are immutable. You can't modify the contents of a `Float64`.

This is something to be aware of when passing types into functions. For example:

```
function mutability_example(mutable_type::Vector{Int}, immutable_type::Int)
    mutable_type[1] += 1
    immutable_type += 1
    return
end

mutable_type = [1, 2, 3]
immutable_type = 1

mutability_example(mutable_type, immutable_type)

println("mutable_type: $(mutable_type)")
println("immutable_type: $(immutable_type)")
```

```
mutable_type: [2, 2, 3]
immutable_type: 1
```

Because `Vector{Int}` is a mutable type, modifying the variable inside the function changed the value outside of the function. In contrast, the change to `immutable_type` didn't modify the value outside the function.

You can check mutability with the `isimmutable` function:

```
isimmutable([1, 2, 3])
```

```
false
```

```
isimmutable(1)
```

```
true
```

The package manager

Installing packages

No matter how wonderful Julia's base language is, at some point you will want to use an extension package. Some of these are built-in, for example random number generation is available in the `Random` package in the standard library. These packages are loaded with the commands `using` and `import`.

```
using Random  # The equivalent of Python's `from Random import *`
import Random  # The equivalent of Python's `import Random`

Random.seed!(33)

[rand() for i in 1:10]
```

```
10-element Vector{Float64}:
 0.8245577112736127
 0.2928364052074266
 0.8765793121770682
 0.41615145984974955
 0.7113242552761618
 0.7762718106176869
 0.407423649552187
 0.15761624576044575
 0.8889767003637221
 0.017829104289712516
```

The Package Manager is used to install packages that are not part of Julia's standard library.

For example the following can be used to install JuMP,

```
using Pkg
Pkg.add("JuMP")
```

For a complete list of registered Julia packages see the package listing at JuliaHub.

From time to you may wish to use a Julia package that is not registered. In this case a git repository URL can be used to install the package.

```
using Pkg
Pkg.add("https://github.com/user-name/MyPackage.jl.git")
```

Package environments

By default, `Pkg.add` will add packages to Julia's global environment. However, Julia also has built-in support for virtual environments.

Activate a virtual environment with:

```
import Pkg; Pkg.activate("/path/to/environment")
```

You can see what packages are installed in the current environment with `Pkg.status()`.

> **Tip**
>
> We strongly recommend you create a Pkg environment for each project that you create in Julia, and add only the packages that you need, instead of adding lots of packages to the global environment. The Pkg manager documentation has more information on this topic.

> **Tip**
>
> This tutorial was generated using Literate.jl. View the source `.jl` file on GitHub.

3.2 Getting started with JuMP

This tutorial is aimed at providing a quick introduction to writing and solving optimization models with JuMP.

If you're new to Julia, you should start with Getting started with Julia.

What is JuMP?

JuMP ("Julia for Mathematical Programming") is an open-source modeling language that is embedded in Julia. It allows users to formulate various classes of optimization problems (linear, mixed-integer, quadratic, conic quadratic, semidefinite, and nonlinear) with easy-to-read code. These problems can then be solved using state-of-the-art open-source and commercial solvers.

JuMP also makes advanced optimization techniques easily accessible from a high-level language.

What is a solver?

A solver is a software package that incorporates algorithms for finding solutions to one or more classes of problem.

For example, GLPK is a solver for linear programming (LP) and mixed integer programming (MIP) problems. It incorporates algorithms such as the simplex method and the interior-point method.

The Supported-solvers table lists the open-source and commercial solvers that JuMP currently supports.

What is MathOptInterface?

Each solver has its own concepts and data structures for representing optimization models and obtaining results.

MathOptInterface (MOI) is an abstraction layer that JuMP uses to convert from the problem written in JuMP to the solver-specific data structures for each solver.

MOI can be used directly, or through a higher-level modeling interface like JuMP.

> **Note**
>
> JuMP re-exports the MathOptInterface.jl package via the `MOI` constant. When you see code like `OPTIMAL`, this is a constant from the MathOptInterface package.

Installation

JuMP is a package for Julia. From Julia, JuMP is installed by using the built-in package manager.

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

You also need to include a Julia package which provides an appropriate solver. One such solver is `GLPK.Optimizer`, which is provided by the GLPK.jl package.

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

See Installation Guide for a list of other solvers you can use.

An example

Let's to solve the following linear programming problem using JuMP and GLPK. We will first look at the complete code to solve the problem and then go through it step by step.

Here's the problem:

$$\begin{aligned} \min \quad & 12x + 20y \\ \text{s.t.} \quad & 6x + 8y \geq 100 \\ & 7x + 12y \geq 120 \\ & x \geq 0 \\ & y \in [0, 3] \end{aligned}$$

And here's the code to solve this problem:

```
using JuMP
using GLPK
model = Model(GLPK.Optimizer)
@variable(model, x >= 0)
@variable(model, 0 <= y <= 3)
@objective(model, Min, 12x + 20y)
@constraint(model, c1, 6x + 8y >= 100)
@constraint(model, c2, 7x + 12y >= 120)
print(model)
optimize!(model)
@show termination_status(model)
@show primal_status(model)
@show dual_status(model)
@show objective_value(model)
@show value(x)
@show value(y)
@show shadow_price(c1)
@show shadow_price(c2)
```

```
Min 12 x + 20 y
Subject to
 c1 : 6 x + 8 y ≥ 100.0
 c2 : 7 x + 12 y ≥ 120.0
 x ≥ 0.0
 y ≥ 0.0
 y ≤ 3.0
termination_status(model) = MathOptInterface.OPTIMAL
primal_status(model) = MathOptInterface.FEASIBLE_POINT
dual_status(model) = MathOptInterface.FEASIBLE_POINT
objective_value(model) = 204.99999999999997
value(x) = 15.000000000000005
value(y) = 1.249999999999996
shadow_price(c1) = -0.24999999999999922
shadow_price(c2) = -1.5000000000000007
```

Step-by-step

Once JuMP is installed, to use JuMP in your programs, we just need to write:

```
using JuMP
```

We also need to include a Julia package which provides an appropriate solver. We want to use `GLPK.Optimizer` here which is provided by the `GLPK.jl` package.

```
using GLPK
```

JuMP builds problems incrementally in a `Model` object. Create a model by passing an optimizer to the `Model` function:

```
model = Model(GLPK.Optimizer)
```

```
A JuMP Model
Feasibility problem with:
Variables: 0
Model mode: AUTOMATIC
CachingOptimizer state: EMPTY_OPTIMIZER
Solver name: GLPK
```

Variables are modeled using `@variable`:

```
@variable(model, x >= 0)
```

$$x$$

They can have lower and upper bounds.

```
@variable(model, 0 <= y <= 30)
```

$$y$$

The objective is set using `@objective`:

```
@objective(model, Min, 12x + 20y)
```

$$12x + 20y$$

Constraints are modeled using `@constraint`. Here, `c1` and `c2` are the names of our constraint.

```
@constraint(model, c1, 6x + 8y >= 100)
```

$\texttt{c1} : 6x + 8y \geq 100.0$

```
@constraint(model, c2, 7x + 12y >= 120)
```

c2 : $7x + 12y \geq 120.0$

Call `print` to display the model:

```
print(model)
```

```
Min 12 x + 20 y
Subject to
 c1 : 6 x + 8 y ≥ 100.0
 c2 : 7 x + 12 y ≥ 120.0
 x ≥ 0.0
 y ≥ 0.0
 y ≤ 30.0
```

To solve the optimization problem, call the `optimize!` function.

```
optimize!(model)
```

> **Info**
>
> The `!` after optimize is just part of the name. It's nothing special. Julia has a convention that functions which mutate their arguments should end in `!`. A common example is `push!`.

Now let's see what information we can query about the solution.

`termination_status` tells us why the solver stopped:

```
termination_status(model)
```

```
OPTIMAL::TerminationStatusCode = 1
```

In this case, the solver found an optimal solution.

Check `primal_status` to see if the solver found a primal feasible point:

```
primal_status(model)
```

```
FEASIBLE_POINT::ResultStatusCode = 1
```

and `dual_status` to see if the solver found a dual feasible point:

```
dual_status(model)
```

```
FEASIBLE_POINT::ResultStatusCode = 1
```

Now we know that our solver found an optimal solution, and that it has a primal and a dual solution to query.

Query the objective value using `objective_value`:

```
objective_value(model)
```

```
205.0
```

the primal solution using `value`:

```
value(x)
```

```
14.999999999999993
```

```
value(y)
```

```
1.2500000000000047
```

and the dual solution using `shadow_price`:

```
shadow_price(c1)
```

```
-0.249999999999999
```

```
shadow_price(c2)
```

```
-1.5000000000000007
```

That's it for our simple model. In the rest of this tutorial, we expand on some of the basic JuMP operations.

Model basics

Create a model by passing an optimizer:

```
model = Model(GLPK.Optimizer)
```

```
A JuMP Model
Feasibility problem with:
Variables: 0
Model mode: AUTOMATIC
CachingOptimizer state: EMPTY_OPTIMIZER
Solver name: GLPK
```

Alternatively, call `set_optimizer` at any point before calling `optimize!`:

```
model = Model()
set_optimizer(model, GLPK.Optimizer)
```

For some solvers, you can also use `direct_model`, which offers a more efficient connection to the underlying solver:

```
model = direct_model(GLPK.Optimizer())
```

```
A JuMP Model
Feasibility problem with:
Variables: 0
Model mode: DIRECT
Solver name: GLPK
```

> **Warning**
>
> Some solvers do not support `direct_model`!

Solver Options

Pass options to solvers with `optimizer_with_attributes`:

```
model = Model(optimizer_with_attributes(GLPK.Optimizer, "msg_lev" => 0))
```

```
A JuMP Model
Feasibility problem with:
Variables: 0
Model mode: AUTOMATIC
CachingOptimizer state: EMPTY_OPTIMIZER
Solver name: GLPK
```

> **Note**
>
> These options are solver-specific. To find out the various options available, see the GitHub README of the individual solver packages. The link to each solver's GitHub page is in the Supported solvers table.

You can also pass options with `set_optimizer_attribute`

```
model = Model(GLPK.Optimizer)
set_optimizer_attribute(model, "msg_lev", 0)
```

Solution basics

We saw above how to use `termination_status` and `primal_status` to understand the solution returned by the solver.

However, you should only query solution attributes like `value` and `objective_value` if there is an available solution. Here's a recommended way to check:

```
function solve_infeasible()
    model = Model(GLPK.Optimizer)
    @variable(model, 0 <= x <= 1)
    @variable(model, 0 <= y <= 1)
    @constraint(model, x + y >= 3)
    @objective(model, Max, x + 2y)
    optimize!(model)
    if termination_status(model) != OPTIMAL
        @warn("The model was not solved correctly.")
        return nothing
    end
    return value(x), value(y)
end

solve_infeasible()
```

```
┌
 Warning: The model was not solved correctly.└
 @ Main getting_started_with_JuMP.md:301
```

Variable basics

Let's create a new empty model to explain some of the variable syntax:

```
model = Model()
```

```
A JuMP Model
Feasibility problem with:
Variables: 0
Model mode: AUTOMATIC
CachingOptimizer state: NO_OPTIMIZER
Solver name: No optimizer attached.
```

Variable bounds

All of the variables we have created till now have had a bound. We can also create a free variable.

```
@variable(model, free_x)
```

$$free\_x$$

While creating a variable, instead of using the <= and >= syntax, we can also use the `lower_bound` and `upper_bound` keyword arguments.

```
@variable(model, keyword_x, lower_bound = 1, upper_bound = 2)
```

$$keyword\_x$$

We can query whether a variable has a bound using the `has_lower_bound` and `has_upper_bound` functions. The values of the bound can be obtained using the `lower_bound` and `upper_bound` functions.

```
has_upper_bound(keyword_x)
```

```
true
```

```
upper_bound(keyword_x)
```

```
2.0
```

Note querying the value of a bound that does not exist will result in an error.

```
    lower_bound(free_x)
```

```
Variable free_x does not have a lower bound.
```

Containers

We have already seen how to add a single variable to a model using the `@variable` macro. Now let's look at ways to add multiple variables to a model.

JuMP provides data structures for adding collections of variables to a model. These data structures are referred to as containers and are of three types: `Array`s, `DenseAxisArray`s, and `SparseAxisArray`s.

Arrays JuMP arrays are created when you have integer indices that start at 1:

```
@variable(model, a[1:2, 1:2])
```

```
2×2 Matrix{VariableRef}:
 a[1,1]  a[1,2]
 a[2,1]  a[2,2]
```

Create an n-dimensional variable $x \in R^n$ with bounds $l \leq x \leq u$ ($l, u \in R^n$) as follows:

```
n = 10
l = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
u = [10, 11, 12, 13, 14, 15, 16, 17, 18, 19]

@variable(model, l[i] <= x[i = 1:n] <= u[i])
```

```
10-element Vector{VariableRef}:
 x[1]
 x[2]
 x[3]
 x[4]
 x[5]
 x[6]
 x[7]
 x[8]
 x[9]
 x[10]
```

We can also create variable bounds that depend upon the indices:

```
@variable(model, y[i = 1:2, j = 1:2] >= 2i + j)
```

```
2×2 Matrix{VariableRef}:
 y[1,1]  y[1,2]
 y[2,1]  y[2,2]
```

DenseAxisArrays `DenseAxisArray`s are used when the indices are not one-based integer ranges. The syntax is similar except with an arbitrary vector as an index as opposed to a one-based range:

```
@variable(model, z[i = 2:3, j = 1:2:3] >= 0)
```

```
2-dimensional DenseAxisArray{VariableRef,2,...} with index sets:
    Dimension 1, 2:3
    Dimension 2, 1:2:3
And data, a 2×2 Matrix{VariableRef}:
 z[2,1]  z[2,3]
 z[3,1]  z[3,3]
```

Indices do not have to be integers. They can be any Julia type:

```
@variable(model, w[1:5, ["red", "blue"]] <= 1)
```

```
2-dimensional DenseAxisArray{VariableRef,2,...} with index sets:
    Dimension 1, Base.OneTo(5)
    Dimension 2, ["red", "blue"]
And data, a 5×2 Matrix{VariableRef}:
 w[1,red]  w[1,blue]
 w[2,red]  w[2,blue]
 w[3,red]  w[3,blue]
 w[4,red]  w[4,blue]
 w[5,red]  w[5,blue]
```

SparseAxisArrays `SparseAxisArrays` are created when the indices do not form a rectangular set. For example, this applies when indices have a dependence upon previous indices (called triangular indexing):

```
@variable(model, u[i = 1:2, j = i:3])
```

```
JuMP.Containers.SparseAxisArray{VariableRef, 2, Tuple{Int64, Int64}} with 5 entries:
  [1, 1]  =  u[1,1]
  [1, 2]  =  u[1,2]
  [1, 3]  =  u[1,3]
  [2, 2]  =  u[2,2]
  [2, 3]  =  u[2,3]
```

We can also conditionally create variables by adding a comparison check that depends upon the named indices and is separated from the indices by a semi-colon `;`:

```
@variable(model, v[i = 1:9; mod(i, 3) == 0])
```

```
JuMP.Containers.SparseAxisArray{VariableRef, 1, Tuple{Int64}} with 3 entries:
  [3]  =  v[3]
  [6]  =  v[6]
  [9]  =  v[9]
```

Integrality

JuMP can create binary and integer variables. Binary variables are constrained to the set $\{0, 1\}$, and integer variables are constrained to the set $\mathbb{Z}$.

Integer variables Create an integer variable by passing `Int`:

```
@variable(model, integer_x, Int)
```

$$integer\_x$$

or setting the `integer` keyword to `true`:

```
@variable(model, integer_z, integer = true)
```

$$integer\_z$$

Binary variables Create a binary variable by passing `Bin`:

```
@variable(model, binary_x, Bin)
```

$$binary\_x$$

or setting the `binary` keyword to `true`:

```
@variable(model, binary_z, binary = true)
```

$$binary\_z$$

Constraint basics

We'll need a need a new model to explain some of the constraint basics:

```
model = Model()
@variable(model, x)
@variable(model, y)
@variable(model, z[1:10]);
```

Containers

Just as we had containers for variables, JuMP also provides `Arrays`, `DenseAxisArrays`, and `SparseAxisArrays` for storing collections of constraints. Examples for each container type are given below.

Arrays

```
@constraint(model, [i = 1:3], i * x <= i + 1)
```

```
3-element Vector{ConstraintRef{Model, MathOptInterface.ConstraintIndex{MathOptInterface.
    ScalarAffineFunction{Float64}, MathOptInterface.LessThan{Float64}}, ScalarShape}}:
 x ≤ 2.0
 2 x ≤ 3.0
 3 x ≤ 4.0
```

DenseAxisArrays

```
@constraint(model, [i = 1:2, j = 2:3], i * x <= j + 1)
```

```
2-dimensional DenseAxisArray{ConstraintRef{Model, MathOptInterface.ConstraintIndex{MathOptInterface.
    ScalarAffineFunction{Float64}, MathOptInterface.LessThan{Float64}}, ScalarShape},2,...} with
    index sets:
   Dimension 1, Base.OneTo(2)
   Dimension 2, 2:3
And data, a 2×2 Matrix{ConstraintRef{Model, MathOptInterface.ConstraintIndex{MathOptInterface.
    ScalarAffineFunction{Float64}, MathOptInterface.LessThan{Float64}}, ScalarShape}}:
 x ≤ 3.0    x ≤ 4.0
 2 x ≤ 3.0  2 x ≤ 4.0
```

SparseAxisArrays

```
@constraint(model, [i = 1:2, j = 1:2; i != j], i * x <= j + 1)
```

```
JuMP.Containers.SparseAxisArray{ConstraintRef{Model, MathOptInterface.ConstraintIndex{
    MathOptInterface.ScalarAffineFunction{Float64}, MathOptInterface.LessThan{Float64}}, ScalarShape
    }, 2, Tuple{Int64, Int64}} with 2 entries:
  [1, 2]  =  x ≤ 3.0
  [2, 1]  =  2 x ≤ 2.0
```

Constraints in a loop

We can add constraints using regular Julia loops:

```
for i in 1:3
    @constraint(model, 6x + 4y >= 5i)
end
```

or use for each loops inside the `@constraint` macro:

```
@constraint(model, [i in 1:3], 6x + 4y >= 5i)
```

```
3-element Vector{ConstraintRef{Model, MathOptInterface.ConstraintIndex{MathOptInterface.
    ScalarAffineFunction{Float64}, MathOptInterface.GreaterThan{Float64}}, ScalarShape}}:
 6 x + 4 y ≥ 5.0
 6 x + 4 y ≥ 10.0
 6 x + 4 y ≥ 15.0
```

We can also create constraints such as $\sum_{i=1}^{10} z_i \leq 1$:

```
@constraint(model, sum(z[i] for i in 1:10) <= 1)
```

$$z_1 + z_2 + z_3 + z_4 + z_5 + z_6 + z_7 + z_8 + z_9 + z_{10} \leq 1.0$$

Objective functions

While the recommended way to set the objective is with the `@objective` macro, the functions `set_objective_sense` and `set_objective_function` provide an equivalent lower-level interface.

```
model = Model(GLPK.Optimizer)
@variable(model, x >= 0)
@variable(model, y >= 0)
set_objective_sense(model, MOI.MIN_SENSE)
set_objective_function(model, x + y)

optimize!(model)
```

```
objective_value(model)
```

```
0.0
```

To query the objective function from a model, we use the `objective_sense`, `objective_function`, and `objective_function_typ` functions.

```
objective_sense(model)
```

```
MIN_SENSE::OptimizationSense = 0
```

```
objective_function(model)
```

$$x + y$$

```
objective_function_type(model)
```

```
AffExpr (alias for GenericAffExpr{Float64, VariableRef})
```

Vectorized syntax

We can also add constraints and an objective to JuMP using vectorized linear algebra. We'll illustrate this by solving an LP in standard form i.e.

$$\begin{aligned} \min \quad & c^T x \\ \text{s.t.} \quad & Ax = b \\ & x \geq 0 \end{aligned}$$

```
vector_model = Model(GLPK.Optimizer)

A = [
    1 1 9 5
    3 5 0 8
    2 0 6 13
```

```
]

b = [7; 3; 5]

c = [1; 3; 5; 2]

@variable(vector_model, x[1:4] >= 0)
@constraint(vector_model, A * x .== b)
@objective(vector_model, Min, c' * x)

optimize!(vector_model)
```

```
objective_value(vector_model)
```

```
4.9230769230769225
```

Tip

This tutorial was generated using Literate.jl. View the source `.jl` file on GitHub.

3.3 Getting started with sets and indexing

Most introductory courses to linear programming will teach you to identify sets over which the decision variables and constraints are indexed. Therefore, it is common to write variables such as x_i for all $i \in I$.

A common stumbling block for new users to JuMP is that JuMP does not provide specialized syntax for constructing and manipulating these sets.

We made this decision because Julia already provides a wealth of data structures for working with sets.

In contrast, because tools like AMPL are stand-alone software packages, they had to define their own syntax for set construction and manipulation. Indeed, the AMPL Book has two entire chapters devoted to sets and indexing (V: Simple Sets and Indexing, and VI: Compound Sets and Indexing).

The purpose of this tutorial is to demonstrate a variety of ways in which you can construct and manipulate sets for optimization models.

If you haven't already, you should first read Getting started with JuMP.

```
using JuMP
```

Unordered sets

Unordered sets are useful to describe non-numeric indices, such as the names of cities or types of products.

The most common way to construct a set is by creating a vector:

```
animals = ["dog", "cat", "chicken", "cow", "pig"]
model = Model()
@variable(model, x[animals])
```

```
1-dimensional DenseAxisArray{VariableRef,1,...} with index sets:
    Dimension 1, ["dog", "cat", "chicken", "cow", "pig"]
And data, a 5-element Vector{VariableRef}:
 x[dog]
 x[cat]
 x[chicken]
 x[cow]
 x[pig]
```

We can also use things like the keys of a dictionary:

```
weight_of_animals = Dict(
    "dog" => 20.0,
    "cat" => 5.0,
    "chicken" => 2.0,
    "cow" => 720.0,
    "pig" => 150.0,
)
animal_keys = keys(weight_of_animals)
model = Model()
@variable(model, x[animal_keys])
```

```
1-dimensional DenseAxisArray{VariableRef,1,...} with index sets:
    Dimension 1, ["cow", "chicken", "cat", "pig", "dog"]
And data, a 5-element Vector{VariableRef}:
 x[cow]
 x[chicken]
 x[cat]
 x[pig]
 x[dog]
```

A third option is to use Julia's Set object.

```
animal_set = Set()
for animal in keys(weight_of_animals)
    push!(animal_set, animal)
end
animal_set
```

```
Set{Any} with 5 elements:
  "cow"
  "chicken"
  "cat"
  "pig"
  "dog"
```

The nice thing about Sets is that they automatically remove duplicates:

```
push!(animal_set, "dog")
animal_set
```

```
Set{Any} with 5 elements:
  "cow"
  "chicken"
  "cat"
  "pig"
  "dog"
```

Note how dog does not appear twice.

```
model = Model()
@variable(model, x[animal_set])
```

```
1-dimensional DenseAxisArray{VariableRef,1,...} with index sets:
    Dimension 1, ["cow", "chicken", "cat", "pig", "dog"]
And data, a 5-element Vector{VariableRef}:
 x[cow]
 x[chicken]
 x[cat]
 x[pig]
 x[dog]
```

Sets of numbers

Sets of numbers are useful to decribe sets that are ordered, such as years or elements in a vector. The easiest way to create sets of numbers is to use Julia's `range` syntax.

These can start at 1:

```
model = Model()
@variable(model, x[1:4])
```

```
4-element Vector{VariableRef}:
 x[1]
 x[2]
 x[3]
 x[4]
```

but they don't have to:

```
model = Model()
@variable(model, x[2012:2021])
```

```
1-dimensional DenseAxisArray{VariableRef,1,...} with index sets:
    Dimension 1, 2012:2021
And data, a 10-element Vector{VariableRef}:
 x[2012]
 x[2013]
 x[2014]
 x[2015]
 x[2016]
 x[2017]
 x[2018]
 x[2019]
 x[2020]
 x[2021]
```

Ranges also have a `start:step:stop` syntax. So the olympic years are:

```
model = Model()
@variable(model, x[1896:4:2020])
```

```
1-dimensional DenseAxisArray{VariableRef,1,...} with index sets:
    Dimension 1, 1896:4:2020
And data, a 32-element Vector{VariableRef}:
 x[1896]
 x[1900]
 x[1904]
 x[1908]
 x[1912]
 x[1916]
 x[1920]
 x[1924]
 x[1928]
 x[1932]

 x[1988]
 x[1992]
 x[1996]
 x[2000]
 x[2004]
 x[2008]
 x[2012]
 x[2016]
 x[2020]
```

Sets of other things

An important observation is that you can have any Julia type as the element of a set. It doesn't have to be a `String` or a `Number`. For example, you can have tuples:

```
sources = ["A", "B", "C"]
sinks = ["D", "E"]
S = [(source, sink) for source in sources, sink in sinks]
model = Model()
@variable(model, x[S])
```

```
1-dimensional DenseAxisArray{VariableRef,1,...} with index sets:
    Dimension 1, [("A", "D") ("A", "E"); ("B", "D") ("B", "E"); ("C", "D") ("C", "E")]
And data, a 6-element Vector{VariableRef}:
 x[("A", "D")]
 x[("B", "D")]
 x[("C", "D")]
 x[("A", "E")]
 x[("B", "E")]
 x[("C", "E")]
```

```
x[("A", "D")]
```

$$x_{("A","D")}$$

For multi-dimensional sets, you can use JuMP's syntax for constructing Containers:

```
model = Model()
@variable(model, x[sources, sinks])
```

```
2-dimensional DenseAxisArray{VariableRef,2,...} with index sets:
    Dimension 1, ["A", "B", "C"]
    Dimension 2, ["D", "E"]
And data, a 3×2 Matrix{VariableRef}:
 x[A,D]  x[A,E]
 x[B,D]  x[B,E]
 x[C,D]  x[C,E]
```

```
x["A", "D"]
```

$$x_{A,D}$$

Info

Note how we indexed `x["A", "D"]` instead of `x[("A", "D")]` as above.

Set operations

Julia has built-in support for set operations such as `union`, `intersect`, and `setdiff`.

Therefore, to create a set of all years in which the summer olympics were held, we can use:

```
baseline = 1896:4:2020
cancelled = [1916, 1940, 1944, 2020]
off_year = [2021]
olympic_years = union(setdiff(baseline, cancelled), off_year)
```

```
29-element Vector{Int64}:
 1896
 1900
 1904
 1908
 1912
 1920
 1924
 1928
 1932
 1936

 1988
 1992
 1996
 2000
 2004
```

```
 2008
 2012
 2016
 2021
```

You can also find the number of elements (i.e., the cardinality) in a set using `length`:

```
length(olympic_years)
```

```
29
```

Set membership operations

To compute membership of sets, use the `in` function.

```
2000 in olympic_years
```

```
true
```

```
2001 in olympic_years
```

```
false
```

Indexing expressions

Use Julia's generator syntax to compute new sets, such as the list of olympic years that are divisible by 3:

```
olympic_3_years = [year for year in olympic_years if mod(year, 3) == 0]
model = Model()
@variable(model, x[olympic_3_years])
```

```
1-dimensional DenseAxisArray{VariableRef,1,...} with index sets:
    Dimension 1, [1896, 1908, 1920, 1932, 1956, 1968, 1980, 1992, 2004, 2016]
And data, a 10-element Vector{VariableRef}:
 x[1896]
 x[1908]
 x[1920]
 x[1932]
 x[1956]
 x[1968]
 x[1980]
 x[1992]
 x[2004]
 x[2016]
```

Alternatively, use JuMP's syntax for constructing Containers:

```
model = Model()
@variable(model, x[year in olympic_years; mod(year, 3) == 0])
```

```
JuMP.Containers.SparseAxisArray{VariableRef, 1, Tuple{Int64}} with 10 entries:
  [1896]  =  x[1896]
  [1908]  =  x[1908]
  [1920]  =  x[1920]
  [1932]  =  x[1932]
  [1956]  =  x[1956]
  [1968]  =  x[1968]
  [1980]  =  x[1980]
  [1992]  =  x[1992]

  [2004]  =  x[2004]
  [2016]  =  x[2016]
```

Compound sets

Consider a transportation problem in which we need to ship goods between cities. We have been provided a list of cities:

```
cities = ["Auckland", "Wellington", "Christchurch", "Dunedin"]
```

```
4-element Vector{String}:
 "Auckland"
 "Wellington"
 "Christchurch"
 "Dunedin"
```

and a distance matrix which records the shipping distance between pairs of cities. If we can't ship between two cities, the distance is 0.

```
distances = [0 643 1071 1426; 0 0 436 790; 0 0 0 360; 1426 0 0 0]
```

```
4×4 Matrix{Int64}:
    0  643  1071  1426
    0    0   436   790
    0    0     0   360
 1426    0     0     0
```

Let's have a look at ways we could write a model with an objective function to minimize the total shipping cost. For simplicity, we'll ignore all constraints.

Fix unused variables

One approach is to fix all variables that we can't use to zero. Most solvers are smart-enough to remove these during a presolve phase, so it has a very small impact on performance:

```
N = length(cities)
model = Model()
@variable(model, x[1:N, 1:N] >= 0)
for i in 1:N, j in 1:N
    if distances[i, j] == 0
        fix(x[i, j], 0.0; force = true)
    end
end
@objective(model, Min, sum(distances[i, j] * x[i, j] for i in 1:N, j in 1:N))
```

$$643x_{1,2} + 1071x_{1,3} + 1426x_{1,4} + 436x_{2,3} + 790x_{2,4} + 360x_{3,4} + 1426x_{4,1}$$

Filtered summation

Another approach is to define filters whenever we want to sum over our decision variables:

```
N = length(cities)
model = Model()
@variable(model, x[1:N, 1:N] >= 0)
@objective(
    model,
    Min,
    sum(
        distances[i, j] * x[i, j] for i in 1:N, j in 1:N if distances[i, j] > 0
    ),
)
```

$$643x_{1,2} + 1071x_{1,3} + 1426x_{1,4} + 436x_{2,3} + 790x_{2,4} + 360x_{3,4} + 1426x_{4,1}$$

Filtered indexing

We could also use JuMP's support for Containers:

```
N = length(cities)
model = Model()
@variable(model, x[i = 1:N, j = 1:N; distances[i, j] > 0])
@objective(model, Min, sum(distances[i...] * x[i] for i in eachindex(x)))
```

$$790x_{2,4} + 643x_{1,2} + 1071x_{1,3} + 1426x_{4,1} + 360x_{3,4} + 1426x_{1,4} + 436x_{2,3}$$

> **Note**
>
> The `i...` is called a "splat". It converts a tuple like `(1, 2)` into two indices like `distances[1, 2]`.

Converting to a different data structure

Another approach, and one that is often the most readable, is to convert the data you have into something that is easier to work with. Originally, we had a vector of strings and a matrix of distances. What we really need is something that maps usable origin-destination pairs to distances. A dictionary is an obvious choice:

```
routes = Dict(
    (a, b) => distances[i, j] for
    (i, a) in enumerate(cities), (j, b) in enumerate(cities) if
    distances[i, j] > 0
)
```

```
Dict{Tuple{String, String}, Int64} with 7 entries:
  ("Auckland", "Wellington")     => 643
  ("Wellington", "Christchurch") => 436
  ("Wellington", "Dunedin")      => 790
  ("Christchurch", "Dunedin")    => 360
  ("Auckland", "Dunedin")        => 1426
  ("Dunedin", "Auckland")        => 1426
  ("Auckland", "Christchurch")   => 1071
```

Then, we can create our model like so:

```
model = Model()
@variable(model, x[keys(routes)])
@objective(model, Min, sum(v * x[k] for (k, v) in routes))
```

$$643x_{("Auckland","Wellington")}+436x_{("Wellington","Christchurch")}+790x_{("Wellington","Dunedin")}+360x_{("Christchurch","Dunedin}$$

This has a number of benefits over the other approaches, including a compacter algebraic model and variables that are named in a more meaningful way.

> **Tip**
>
> If you're struggling to formulate a problem using the available syntax in JuMP, it's probably a sign that you should convert your data into a different form.

Next steps

The purpose of this tutorial was to show how JuMP does not have specialized syntax for set creation and manipulation. Instead, you should use the tools provided by Julia itself.

This is both an opportunity and a challenge, because you are free to pick the syntax and data structures that best suit your problem, but for new users it can be daunting to decide which structure to use.

Read through some of the other JuMP tutorials to get inspiration and ideas for how you can use Julia's syntax and data structures to your advantage.

> **Tip**
>
> This tutorial was generated using Literate.jl. View the source `.jl` file on GitHub.

3.4 Getting started with data and plotting

In this tutorial we will learn how to read tabular data into Julia, and some of the basics of plotting.

If you're new to Julia, start by reading Getting started with Julia and Getting started with JuMP first.

> **Note**
>
> There are multiple ways to read the same kind of data into Julia. This tutorial focuses on DataFrames.jl because it provides the ecosystem to work with most of the required file types in a straightforward manner.

Before we get started, we need this constant to point to where the data files are.

```
const DATA_DIR = joinpath(@__DIR__, "data")
```

```
"/home/runner/work/JuMP.jl/JuMP.jl/docs/latex_build/tutorials/getting_started/data"
```

Where to get help

Read the documentation

- Plots.jl: http://docs.juliaplots.org/latest/
- CSV.jl: http://csv.juliadata.org/stable
- DataFrames.jl: https://dataframes.juliadata.org/stable/

Preliminaries

To get started, we need to install some packages.

DataFrames.jl

The `DataFrames` package provides a set of tools for working with tabular data. It is available through the Julia package manager.

```
using Pkg
Pkg.add("DataFrames")
```

```
import DataFrames
```

!!! info What is a DataFrame? A DataFrame is a data structure like a table or spreadsheet. You can use it for storing and exploring a set of related data values. Think of it as a smarter array for holding tabular data.

Plots.jl

The `Plots` package provides a set of tools for plotting. It is available through the Julia package manager.

```
using Pkg
Pkg.add("Plots")
```

```
import Plots
```

CSV .jl

CSV and other delimited text files can be read by the CSV.jl package.

```
Pkg.add("CSV")
```

```
import CSV
```

DataFrame basics

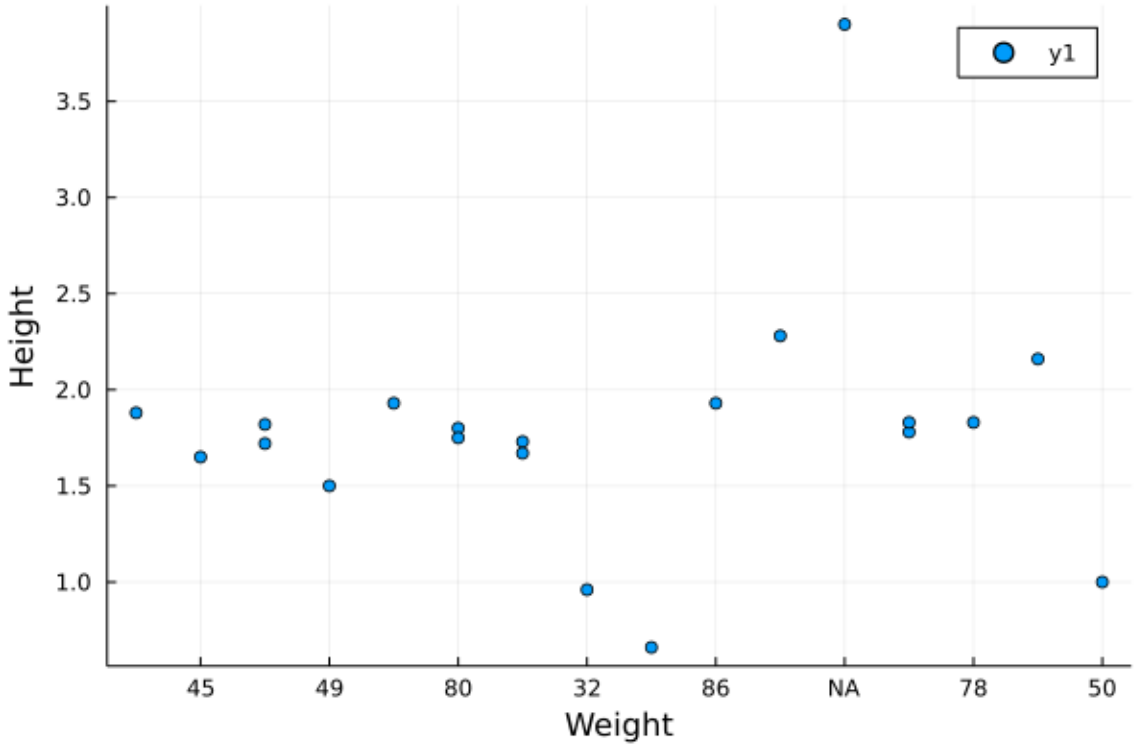
To read a CSV file into a DataFrame, we use the `CSV.read` function.

```
csv_df = CSV.read(joinpath(DATA_DIR, "StarWars.csv"), DataFrames.DataFrame)
```

	Name	Gender	Height	Weight	Eyecolor	Haircolor	Skincolor	Homeland	Born	D
	String	String	Float64	String	String	String	String	String	String	St
1	Anakin Skywalker	male	1.88	84	blue	blond	fair	Tatooine	41.9BBY	4A
2	Padme Amidala	female	1.65	45	brown	brown	light	Naboo	46BBY	19
3	Luke Skywalker	male	1.72	77	blue	blond	fair	Tatooine	19BBY	unk
4	Leia Skywalker	female	1.5	49	brown	brown	light	Alderaan	19BBY	unk
5	Qui-Gon Jinn	male	1.93	88.5	blue	brown	light	unk_planet	92BBY	32
6	Obi-Wan Kenobi	male	1.82	77	bluegray	auburn	fair	Stewjon	57BBY	0E
7	Han Solo	male	1.8	80	brown	brown	light	Corellia	29BBY	unk
8	Sheev Palpatine	male	1.73	75	blue	red	pale	Naboo	82BBY	10
9	R2-D2	male	0.96	32	NA	NA	NA	Naboo	33BBY	unk
10	C-3PO	male	1.67	75	NA	NA	NA	Tatooine	112BBY	3A
11	Yoda	male	0.66	17	brown	brown	green	unk_planet	896BBY	4A
12	Darth Maul	male	1.75	80	yellow	none	red	Dathomir	54BBY	unk
13	Dooku	male	1.93	86	brown	brown	light	Serenno	102BBY	19
14	Chewbacca	male	2.28	112	blue	brown	NA	Kashyyyk	200BBY	25
15	Jabba	male	3.9	NA	yellow	none	tan-green	Tatooine	unk_born	4A
16	Lando Calrissian	male	1.78	79	brown	blank	dark	Socorro	31BBY	unk
17	Boba Fett	male	1.83	78	brown	black	brown	Kamino	31.5BBY	unk
18	Jango Fett	male	1.83	79	brown	black	brown	ConcordDawn	66BBY	22
19	Grievous	male	2.16	159	gold	black	orange	Kalee	unk_born	19
20	Chief Chirpa	male	1.0	50	black	gray	brown	Endor	unk_born	4A

Let's try plotting some of this data

```
Plots.scatter(
    csv_df.Weight,
    csv_df.Height,
    xlabel = "Weight",
    ylabel = "Height",
)
```



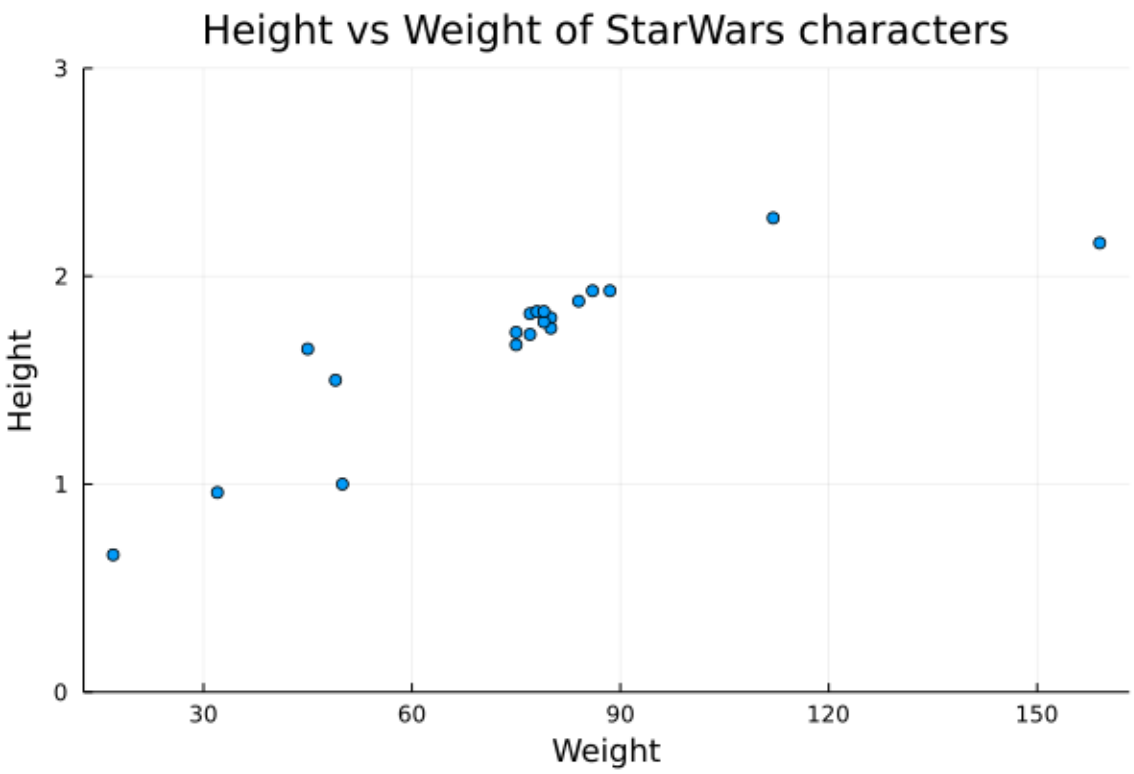
That doesn't look right. What happened? If you look at the dataframe above, it read `Weight` in as a `String` column because there are "NA" fields. Let's correct that, by telling CSV to consider "NA" as `missing`.

```
csv_df = CSV.read(
    joinpath(DATA_DIR, "StarWars.csv"),
    DataFrames.DataFrame,
    missingstring = "NA",
)
```

	Name	Gender	Height	Weight	Eyecolor	Haircolor	Skincolor	Homeland	Born	
	String	String	Float64	Float64?	String?	String?	String?	String	String	S
1	Anakin Skywalker	male	1.88	84.0	blue	blond	fair	Tatooine	41.9BBY	
2	Padme Amidala	female	1.65	45.0	brown	brown	light	Naboo	46BBY	1
3	Luke Skywalker	male	1.72	77.0	blue	blond	fair	Tatooine	19BBY	un
4	Leia Skywalker	female	1.5	49.0	brown	brown	light	Alderaan	19BBY	un
5	Qui-Gon Jinn	male	1.93	88.5	blue	brown	light	unk_planet	92BBY	3
6	Obi-Wan Kenobi	male	1.82	77.0	bluegray	auburn	fair	Stewjon	57BBY	
7	Han Solo	male	1.8	80.0	brown	brown	light	Corellia	29BBY	un
8	Sheev Palpatine	male	1.73	75.0	blue	red	pale	Naboo	82BBY	1
9	R2-D2	male	0.96	32.0	missing	missing	missing	Naboo	33BBY	un
10	C-3PO	male	1.67	75.0	missing	missing	missing	Tatooine	112BBY	
11	Yoda	male	0.66	17.0	brown	brown	green	unk_planet	896BBY	
12	Darth Maul	male	1.75	80.0	yellow	none	red	Dathomir	54BBY	un
13	Dooku	male	1.93	86.0	brown	brown	light	Serenno	102BBY	1
14	Chewbacca	male	2.28	112.0	blue	brown	missing	Kashyyyk	200BBY	2
15	Jabba	male	3.9	missing	yellow	none	tan-green	Tatooine	unk_born	
16	Lando Calrissian	male	1.78	79.0	brown	blank	dark	Socorro	31BBY	un
17	Boba Fett	male	1.83	78.0	brown	black	brown	Kamino	31.5BBY	un
18	Jango Fett	male	1.83	79.0	brown	black	brown	ConcordDawn	66BBY	2
19	Grievous	male	2.16	159.0	gold	black	orange	Kalee	unk_born	1
20	Chief Chirpa	male	1.0	50.0	black	gray	brown	Endor	unk_born	

Then let's re-plot our data

```
Plots.scatter(
    csv_df.Weight,
    csv_df.Height,
    title = "Height vs Weight of StarWars characters",
    xlabel = "Weight",
    ylabel = "Height",
    label = false,
    ylims = (0, 3),
)
```



Better!

Tip

Read the CSV documentation for other parsing options.

DataFrames.jl supports manipulation using functions similar to pandas. For example, split the dataframe into groups based on eye-color:

```
by_eyecolor = DataFrames.groupby(csv_df, :Eyecolor)
```

GroupedDataFrame with 7 groups based on key: Eyecolor

First Group (5 rows): Eyecolor = ”blue”

	Name	Gender	Height	Weight	Eyecolor	Haircolor	Skincolor	Homeland	Born	Died
	String	String	Float64	Float64?	String?	String?	String?	String	String	String
1	Anakin Skywalker	male	1.88	84.0	blue	blond	fair	Tatooine	41.9BBY	4ABY
2	Luke Skywalker	male	1.72	77.0	blue	blond	fair	Tatooine	19BBY	unk_die
3	Qui-Gon Jinn	male	1.93	88.5	blue	brown	light	unk_planet	92BBY	32BBY
4	Sheev Palpatine	male	1.73	75.0	blue	red	pale	Naboo	82BBY	10ABY
5	Chewbacca	male	2.28	112.0	blue	brown	missing	Kashyyyk	200BBY	25ABY

. . .

Last Group (1 row): Eyecolor = ”black”

	Name	Gender	Height	Weight	Eyecolor	Haircolor	Skincolor	Homeland	Born	Died	Je
	String	String	Float64	Float64?	String?	String?	String?	String	String	String	Stri
1	Chief Chirpa	male	1.0	50.0	black	gray	brown	Endor	unk_born	4ABY	no_j

Then recombine into a single dataframe based on a function operating over the split dataframes:

```
eyecolor_count = DataFrames.combine(by_eyecolor) do df
    return DataFrames.nrow(df)
end
```

	Eyecolor	x1
	String?	Int64
1	blue	5
2	brown	8
3	bluegray	1
4	missing	2
5	yellow	2
6	gold	1
7	black	1

We can rename columns:

```
DataFrames.rename!(eyecolor_count, :x1 => :count)
```

	Eyecolor	count
	String?	Int64
1	blue	5
2	brown	8
3	bluegray	1
4	missing	2
5	yellow	2
6	gold	1
7	black	1

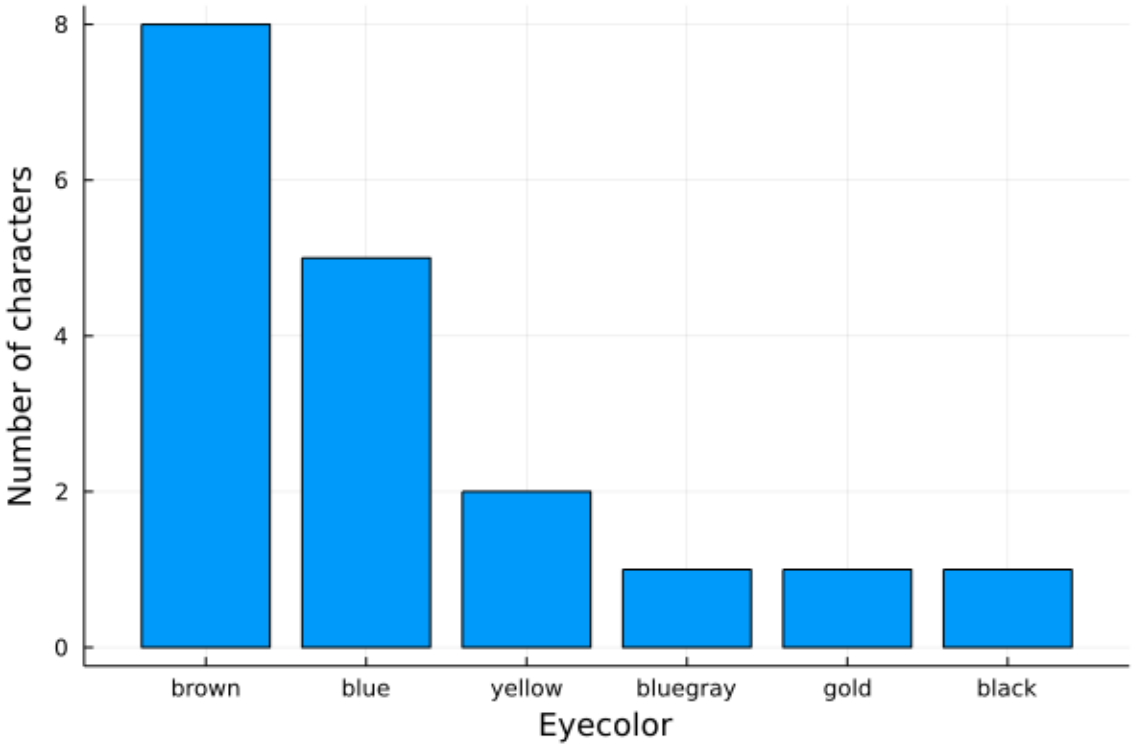
Drop some missing rows:

```
DataFrames.dropmissing!(eyecolor_count, :Eyecolor)
```

	Eyecolor	count
	String	Int64
1	blue	5
2	brown	8
3	bluegray	1
4	yellow	2
5	gold	1
6	black	1

Then we can visualize the data:

```
sort!(eyecolor_count, :count, rev = true)
Plots.bar(
    eyecolor_count.Eyecolor,
    eyecolor_count.count,
    xlabel = "Eyecolor",
    ylabel = "Number of characters",
    label = false,
)
```



Other Delimited Files

We can also use the `CSV.jl` package to read any other delimited text file format.

By default, CSV.File will try to detect a file's delimiter from the first 10 lines of the file.

Candidate delimiters include `','`, `'\t'`, `' '`, `'|'`, `';'`, and `':'`. If it can't auto-detect the delimiter, it will assume `','`.

Let's take the example of space separated data.

```
ss_df = CSV.read(joinpath(DATA_DIR, "Cereal.txt"), DataFrames.DataFrame)
```

	Name	Cups	Calories	Carbs	Fat	Fiber	Potassium	Protein	Sodium	Sugars
	String	Float64	Int64	Float64	Int64	Float64	Int64	Int64	Int64	Int64
1	CapnCrunch	0.75	120	12.0	2	0.0	35	1	220	12
2	CocoaPuffs	1.0	110	12.0	1	0.0	55	1	180	13
3	Trix	1.0	110	13.0	1	0.0	25	1	140	12
4	AppleJacks	1.0	110	11.0	0	1.0	30	2	125	14
5	CornChex	1.0	110	22.0	0	0.0	25	2	280	3
6	CornFlakes	1.0	100	21.0	0	1.0	35	2	290	2
7	Nut&Honey	0.67	120	15.0	1	0.0	40	2	190	9
8	Smacks	0.75	110	9.0	1	1.0	40	2	70	15
9	MultiGrain	1.0	100	15.0	1	2.0	90	2	220	6
10	CracklinOat	0.5	110	10.0	3	4.0	160	3	140	7
11	GrapeNuts	0.25	110	17.0	0	3.0	90	3	179	3
12	HoneyNutCheerios	0.75	110	11.5	1	1.5	90	3	250	10
13	NutriGrain	0.67	140	21.0	2	3.0	130	3	220	7
14	Product19	1.0	100	20.0	0	1.0	45	3	320	3
15	TotalRaisinBran	1.0	140	15.0	1	4.0	230	3	190	14
16	WheatChex	0.67	100	17.0	1	3.0	115	3	230	3
17	Oatmeal	0.5	130	13.5	2	1.5	120	3	170	10
18	Life	0.67	100	12.0	2	2.0	95	4	150	6
19	Maypo	1.0	100	16.0	1	0.0	95	4	0	3
20	QuakerOats	0.5	100	14.0	1	2.0	110	4	135	6
21	Muesli	1.0	150	16.0	3	3.0	170	4	150	11
22	Cheerios	1.25	110	17.0	2	2.0	105	6	290	1
23	SpecialK	1.0	110	16.0	0	1.0	55	6	230	3

We can also specify the delimiter by passing the `delim` argument.

```
delim_df = CSV.read(
    joinpath(DATA_DIR, "Soccer.txt"),
    DataFrames.DataFrame,
    delim = "::",
)
```

	Team	Played	Wins	Draws	Losses	Goals_for	Goals_against
	String	Int64	Int64	Int64	Int64	String	String
1	Barcelona	38	30	4	4	110 goals	21 goals
2	Real Madrid	38	30	2	6	118 goals	38 goals
3	Atletico Madrid	38	23	9	6	67 goals	29 goals
4	Valencia	38	22	11	5	70 goals	32 goals
5	Seville	38	23	7	8	71 goals	45 goals
6	Villarreal	38	16	12	10	48 goals	37 goals
7	Athletic Bilbao	38	15	10	13	42 goals	41 goals
8	Celta Vigo	38	13	12	13	47 goals	44 goals
9	Malaga	38	14	8	16	42 goals	48 goals
10	Espanyol	38	13	10	15	47 goals	51 goals
11	Rayo Vallecano	38	15	4	19	46 goals	68 goals
12	Real Sociedad	38	11	13	14	44 goals	51 goals
13	Elche	38	11	8	19	35 goals	62 goals
14	Levante	38	9	10	19	34 goals	67 goals
15	Getafe	38	10	7	21	33 goals	64 goals
16	Deportivo La Coruna	38	7	14	17	35 goals	60 goals
17	Granada	38	7	14	17	29 goals	64 goals
18	Eibar	38	9	8	21	34 goals	55 goals
19	Almeria	38	8	8	22	35 goals	64 goals
20	Cordoba	38	3	11	24	22 goals	68 goals

Working with DataFrames

Now that we have read the required data into a DataFrame, let us look at some basic operations we can perform on it.

Querying Basic Information

The `size` function gets us the dimensions of the DataFrame:

```
DataFrames.size(ss_df)
```

```
(23, 10)
```

We can also use the `nrow` and `ncol` functions to get the number of rows and columns respectively:

```
DataFrames.nrow(ss_df), DataFrames.ncol(ss_df)
```

```
(23, 10)
```

The `describe` function gives basic summary statistics of data in a DataFrame:

```
DataFrames.describe(ss_df)
```

	variable	mean	min	median	max	nmissing	eltype
	Symbol	Union…	Any	Union…	Any	Int64	DataType
1	Name		AppleJacks		WheatChex	0	String
2	Cups	0.823043	0.25	1.0	1.25	0	Float64
3	Calories	113.043	100	110.0	150	0	Int64
4	Carbs	15.0435	9.0	15.0	22.0	0	Float64
5	Fat	1.13043	0	1.0	3	0	Int64
6	Fiber	1.56522	0.0	1.5	4.0	0	Float64
7	Potassium	86.3043	25	90.0	230	0	Int64
8	Protein	2.91304	1	3.0	6	0	Int64
9	Sodium	189.957	0	190.0	320	0	Int64
10	Sugars	7.52174	1	7.0	15	0	Int64

Names of every column can be obtained by the `names` function:

```
DataFrames.names(ss_df)
```

```
10-element Vector{String}:
 "Name"
 "Cups"
 "Calories"
 "Carbs"
 "Fat"
 "Fiber"
 "Potassium"
 "Protein"
 "Sodium"
 "Sugars"
```

Corresponding data types are obtained using the broadcasted `eltype` function:

```
eltype.(ss_df)
```

	Name	Cups	Calories	Carbs	Fat	Fiber	Potassium	Protein	Sodium	Sugars
	DataType	DataType	DataType	DataType	DataType	DataType	DataType	DataType	DataType	DataType
1	Char	Float64	Int64	Float64	Int64	Float64	Int64	Int64	Int64	Int64
2	Char	Float64	Int64	Float64	Int64	Float64	Int64	Int64	Int64	Int64
3	Char	Float64	Int64	Float64	Int64	Float64	Int64	Int64	Int64	Int64
4	Char	Float64	Int64	Float64	Int64	Float64	Int64	Int64	Int64	Int64
5	Char	Float64	Int64	Float64	Int64	Float64	Int64	Int64	Int64	Int64
6	Char	Float64	Int64	Float64	Int64	Float64	Int64	Int64	Int64	Int64
7	Char	Float64	Int64	Float64	Int64	Float64	Int64	Int64	Int64	Int64
8	Char	Float64	Int64	Float64	Int64	Float64	Int64	Int64	Int64	Int64
9	Char	Float64	Int64	Float64	Int64	Float64	Int64	Int64	Int64	Int64
10	Char	Float64	Int64	Float64	Int64	Float64	Int64	Int64	Int64	Int64
11	Char	Float64	Int64	Float64	Int64	Float64	Int64	Int64	Int64	Int64
12	Char	Float64	Int64	Float64	Int64	Float64	Int64	Int64	Int64	Int64
13	Char	Float64	Int64	Float64	Int64	Float64	Int64	Int64	Int64	Int64
14	Char	Float64	Int64	Float64	Int64	Float64	Int64	Int64	Int64	Int64
15	Char	Float64	Int64	Float64	Int64	Float64	Int64	Int64	Int64	Int64
16	Char	Float64	Int64	Float64	Int64	Float64	Int64	Int64	Int64	Int64
17	Char	Float64	Int64	Float64	Int64	Float64	Int64	Int64	Int64	Int64
18	Char	Float64	Int64	Float64	Int64	Float64	Int64	Int64	Int64	Int64
19	Char	Float64	Int64	Float64	Int64	Float64	Int64	Int64	Int64	Int64
20	Char	Float64	Int64	Float64	Int64	Float64	Int64	Int64	Int64	Int64
21	Char	Float64	Int64	Float64	Int64	Float64	Int64	Int64	Int64	Int64
22	Char	Float64	Int64	Float64	Int64	Float64	Int64	Int64	Int64	Int64
23	Char	Float64	Int64	Float64	Int64	Float64	Int64	Int64	Int64	Int64

Accessing the Data

Similar to regular arrays, we use numerical indexing to access elements of a DataFrame:

```
csv_df[1, 1]
```

```
"Anakin Skywalker"
```

The following are different ways to access a column:

```
csv_df[!, 1]
```

```
20-element Vector{String}:
 "Anakin Skywalker"
 "Padme Amidala"
 "Luke Skywalker"
 "Leia Skywalker"
 "Qui-Gon Jinn"
 "Obi-Wan Kenobi"
 "Han Solo"
 "Sheev Palpatine"
 "R2-D2"
 "C-3PO"
 "Yoda"
 "Darth Maul"
 "Dooku"
```

```
 "Chewbacca"
 "Jabba"
 "Lando Calrissian"
 "Boba Fett"
 "Jango Fett"
 "Grievous"
 "Chief Chirpa"
```

```
csv_df[!, :Name]
```

```
20-element Vector{String}:
 "Anakin Skywalker"
 "Padme Amidala"
 "Luke Skywalker"
 "Leia Skywalker"
 "Qui-Gon Jinn"
 "Obi-Wan Kenobi"
 "Han Solo"
 "Sheev Palpatine"
 "R2-D2"
 "C-3PO"
 "Yoda"
 "Darth Maul"
 "Dooku"
 "Chewbacca"
 "Jabba"
 "Lando Calrissian"
 "Boba Fett"
 "Jango Fett"
 "Grievous"
 "Chief Chirpa"
```

```
csv_df.Name
```

```
20-element Vector{String}:
 "Anakin Skywalker"
 "Padme Amidala"
 "Luke Skywalker"
 "Leia Skywalker"
 "Qui-Gon Jinn"
 "Obi-Wan Kenobi"
 "Han Solo"
 "Sheev Palpatine"
 "R2-D2"
 "C-3PO"
 "Yoda"
 "Darth Maul"
 "Dooku"
 "Chewbacca"
 "Jabba"
 "Lando Calrissian"
 "Boba Fett"
 "Jango Fett"
 "Grievous"
 "Chief Chirpa"
```

```
csv_df[:, 1] # Note that this creates a copy.
```

```
20-element Vector{String}:
 "Anakin Skywalker"
 "Padme Amidala"
 "Luke Skywalker"
 "Leia Skywalker"
 "Qui-Gon Jinn"
 "Obi-Wan Kenobi"
 "Han Solo"
 "Sheev Palpatine"
 "R2-D2"
 "C-3PO"
 "Yoda"
 "Darth Maul"
 "Dooku"
 "Chewbacca"
 "Jabba"
 "Lando Calrissian"
 "Boba Fett"
 "Jango Fett"
 "Grievous"
 "Chief Chirpa"
```

The following are different ways to access a row:

```
csv_df[1:1, :]
```

	Name	Gender	Height	Weight	Eyecolor	Haircolor	Skincolor	Homeland	Born	Died
	String	String	Float64	Float64?	String?	String?	String?	String	String	String
1	Anakin Skywalker	male	1.88	84.0	blue	blond	fair	Tatooine	41.9BBY	4ABY

```
csv_df[1, :] # This produces a DataFrameRow.
```

	Name	Gender	Height	Weight	Eyecolor	Haircolor	Skincolor	Homeland	Born	Died
	String	String	Float64	Float64?	String?	String?	String?	String	String	String
1	Anakin Skywalker	male	1.88	84.0	blue	blond	fair	Tatooine	41.9BBY	4ABY

We can change the values just as we normally assign values.

Assign a range to scalar:

```
csv_df[1:3, :Height] .= 1.83
```

```
3-element view(::Vector{Float64}, 1:3) with eltype Float64:
 1.83
 1.83
 1.83
```

Assign a vector:

```
csv_df[4:6, :Height] = [1.8, 1.6, 1.8]
```

```
3-element Vector{Float64}:
 1.8
 1.6
 1.8
```

```
csv_df
```

	Name	Gender	Height	Weight	Eyecolor	Haircolor	Skincolor	Homeland	Born	
	String	String	Float64	Float64?	String?	String?	String?	String	String	S
1	Anakin Skywalker	male	1.83	84.0	blue	blond	fair	Tatooine	41.9BBY	
2	Padme Amidala	female	1.83	45.0	brown	brown	light	Naboo	46BBY	1
3	Luke Skywalker	male	1.83	77.0	blue	blond	fair	Tatooine	19BBY	un
4	Leia Skywalker	female	1.8	49.0	brown	brown	light	Alderaan	19BBY	un
5	Qui-Gon Jinn	male	1.6	88.5	blue	brown	light	unk_planet	92BBY	3
6	Obi-Wan Kenobi	male	1.8	77.0	bluegray	auburn	fair	Stewjon	57BBY	
7	Han Solo	male	1.8	80.0	brown	brown	light	Corellia	29BBY	un
8	Sheev Palpatine	male	1.73	75.0	blue	red	pale	Naboo	82BBY	1
9	R2-D2	male	0.96	32.0	missing	missing	missing	Naboo	33BBY	un
10	C-3PO	male	1.67	75.0	missing	missing	missing	Tatooine	112BBY	
11	Yoda	male	0.66	17.0	brown	brown	green	unk_planet	896BBY	
12	Darth Maul	male	1.75	80.0	yellow	none	red	Dathomir	54BBY	un
13	Dooku	male	1.93	86.0	brown	brown	light	Serenno	102BBY	1
14	Chewbacca	male	2.28	112.0	blue	brown	missing	Kashyyyk	200BBY	2
15	Jabba	male	3.9	missing	yellow	none	tan-green	Tatooine	unk_born	
16	Lando Calrissian	male	1.78	79.0	brown	blank	dark	Socorro	31BBY	un
17	Boba Fett	male	1.83	78.0	brown	black	brown	Kamino	31.5BBY	un
18	Jango Fett	male	1.83	79.0	brown	black	brown	ConcordDawn	66BBY	2
19	Grievous	male	2.16	159.0	gold	black	orange	Kalee	unk_born	1
20	Chief Chirpa	male	1.0	50.0	black	gray	brown	Endor	unk_born	

Tip

There are a lot more things which can be done with a DataFrame. Read the docs for more information.

For information on dplyr-type syntax:

- Read the DataFrames.jl documentation
- Check out DataFramesMeta.jl

A Complete Modelling Example - Passport Problem

Let's now apply what we have learnt to solve a real modelling problem.

Data manipulation

The Passport Index Dataset lists travel visa requirements for 199 countries, in `.csv` format. Our task is to find the minimum number of passports required to visit all countries.

```
passport_data = CSV.read(
    joinpath(DATA_DIR, "passport-index-matrix.csv"),
    DataFrames.DataFrame,
)
```

	Passport	Afghanistan	Albania	Algeria	Andorra	Angola	Antigua and Barbuda	Ar
	String	Int64	Int64	Int64	Int64	Int64	Int64	
1	Afghanistan	-1	0	0	0	0	0	
2	Albania	0	-1	0	3	0	3	
3	Algeria	0	0	-1	0	1	0	
4	Andorra	0	3	0	-1	0	3	
5	Angola	0	0	0	0	-1	0	
6	Antigua and Barbuda	0	3	0	3	0	-1	
7	Argentina	0	3	0	3	1	3	
8	Armenia	0	3	0	0	0	3	
9	Australia	0	3	0	3	1	3	
10	Austria	0	3	0	3	1	3	
11	Azerbaijan	0	3	0	0	0	3	
12	Bahamas	0	3	0	3	0	3	
13	Bahrain	0	3	0	0	0	0	
14	Bangladesh	0	0	0	0	0	0	
15	Barbados	0	3	0	3	0	3	
16	Belarus	0	3	0	0	0	3	
17	Belgium	0	3	0	3	1	3	
18	Belize	0	0	0	0	0	3	
19	Benin	0	0	0	0	0	0	
20	Bhutan	0	0	0	0	0	0	
21	Bolivia	0	0	0	0	0	0	
22	Bosnia and Herzegovina	0	3	0	3	0	0	
23	Botswana	0	0	0	0	3	3	
24	Brazil	0	3	0	3	1	3	
25	Brunei	0	3	0	3	0	3	
26	Bulgaria	0	3	0	3	1	3	
27	Burkina Faso	0	0	0	0	0	0	
28	Burundi	0	0	0	0	0	0	
29	Cambodia	0	0	0	0	0	0	
30	Cameroon	0	0	0	0	0	0	
31	Canada	0	3	0	3	1	3	
32	Cape Verde	0	0	0	0	1	0	
33	Central African Republic	0	0	0	0	0	0	
34	Chad	0	0	0	0	0	0	
35	Chile	0	3	0	3	1	3	
36	China	0	3	0	0	1	3	
37	Colombia	0	3	0	3	0	3	
38	Comoros	0	0	0	0	0	0	
39	Congo	0	0	0	0	0	0	
40	DR Congo	0	0	0	0	0	0	
41	Costa Rica	0	3	0	3	0	0	
42	Ivory Coast	0	0	0	0	0	0	
43	Croatia	0	3	0	3	1	3	
44	Cuba	0	0	0	0	1	3	
45	Cyprus	0	3	0	3	1	3	
46	Czech Republic	0	3	0	3	1	3	
47	Denmark	0	3	0	3	1	3	
48	Djibouti	0	0	0	0	0	0	
49	Dominica	0	0	0	3	0	3	
50	Dominican Republic	0	0	0	0	0	0	
51	Ecuador	0	0	0	0	0	0	
52	Egypt	0	0	0	0	0	0	
53	El Salvador	0	3	0	3	0	0	
54	Equatorial Guinea	0	0	0	0	0	0	
55	Eritrea	0	0	0	0	0	0	

In this dataset, the first column represents a passport (=from) and each remaining column represents a foreign country (=to).

The values in each cell are as follows:

- 3 = visa-free travel
- 2 = eTA is required
- 1 = visa can be obtained on arrival
- 0 = visa is required
- -1 is for all instances where passport and destination are the same

Our task is to find out the minimum number of passports needed to visit every country without requiring a visa.

The values we are interested in are -1 and 3. Let's modify the dataframe so that the -1 and 3 are 1 (true), and all others are `0` (false):

```
function modifier(x)
    if x == -1 || x == 3
        return 1
    else
        return 0
    end
end

for country in passport_data.Passport
    passport_data[!, country] = modifier.(passport_data[!, country])
end

passport_data
```

	Passport	Afghanistan	Albania	Algeria	Andorra	Angola	Antigua and Barbuda	Ar
	String	Int64	Int64	Int64	Int64	Int64	Int64	
1	Afghanistan	1	0	0	0	0	0	
2	Albania	0	1	0	1	0	1	
3	Algeria	0	0	1	0	0	0	
4	Andorra	0	1	0	1	0	1	
5	Angola	0	0	0	0	1	0	
6	Antigua and Barbuda	0	1	0	1	0	1	
7	Argentina	0	1	0	1	0	1	
8	Armenia	0	1	0	0	0	1	
9	Australia	0	1	0	1	0	1	
10	Austria	0	1	0	1	0	1	
11	Azerbaijan	0	1	0	0	0	1	
12	Bahamas	0	1	0	1	0	1	
13	Bahrain	0	1	0	0	0	0	
14	Bangladesh	0	0	0	0	0	0	
15	Barbados	0	1	0	1	0	1	
16	Belarus	0	1	0	0	0	1	
17	Belgium	0	1	0	1	0	1	
18	Belize	0	0	0	0	0	1	
19	Benin	0	0	0	0	0	0	
20	Bhutan	0	0	0	0	0	0	
21	Bolivia	0	0	0	0	0	0	
22	Bosnia and Herzegovina	0	1	0	1	0	0	
23	Botswana	0	0	0	0	1	1	
24	Brazil	0	1	0	1	0	1	
25	Brunei	0	1	0	1	0	1	
26	Bulgaria	0	1	0	1	0	1	
27	Burkina Faso	0	0	0	0	0	0	
28	Burundi	0	0	0	0	0	0	
29	Cambodia	0	0	0	0	0	0	
30	Cameroon	0	0	0	0	0	0	
31	Canada	0	1	0	1	0	1	
32	Cape Verde	0	0	0	0	0	0	
33	Central African Republic	0	0	0	0	0	0	
34	Chad	0	0	0	0	0	0	
35	Chile	0	1	0	1	0	1	
36	China	0	1	0	0	0	1	
37	Colombia	0	1	0	1	0	1	
38	Comoros	0	0	0	0	0	0	
39	Congo	0	0	0	0	0	0	
40	DR Congo	0	0	0	0	0	0	
41	Costa Rica	0	1	0	1	0	0	
42	Ivory Coast	0	0	0	0	0	0	
43	Croatia	0	1	0	1	0	1	
44	Cuba	0	0	0	0	0	1	
45	Cyprus	0	1	0	1	0	1	
46	Czech Republic	0	1	0	1	0	1	
47	Denmark	0	1	0	1	0	1	
48	Djibouti	0	0	0	0	0	0	
49	Dominica	0	0	0	1	0	1	
50	Dominican Republic	0	0	0	0	0	0	
51	Ecuador	0	0	0	0	0	0	
52	Egypt	0	0	0	0	0	0	
53	El Salvador	0	1	0	1	0	0	
54	Equatorial Guinea	0	0	0	0	0	0	
55	Eritrea	0	0	0	0	0	0	

The values in the cells now represent:

- 1 = no visa required for travel
- 0 = visa required for travel

JuMP Modeling

To model the problem as a mixed-integer linear program, we need a binary decision variable x_c for each country c. x_c is 1 if we select passport c and 0 otherwise. Our objective is to minimize the sum $\sum x_c$ over all countries.

Since we wish to visit all the countries, for every country, we should own at least one passport that lets us travel to that country visa free. For one destination, this can be mathematically represented as $\sum_{c \in C} a_{c,d} \cdot x_d \geq 1$, where a is the `passport_data` dataframe.

Thus, we can represent this problem using the following model:

$$\begin{aligned} \min \quad & \sum_{c \in C} x_c \\ \text{s.t.} \quad & \sum_{c \in C} a_{c,d} x_c \geq 1 \quad \forall d \in C \\ & x_c \in \{0,1\} \quad \forall c \in C. \end{aligned}$$

We'll now solve the problem using JuMP:

```
using JuMP
import GLPK
```

First, create the set of countries:

```
C = passport_data.Passport
```

```
199-element Vector{String}:
 "Afghanistan"
 "Albania"
 "Algeria"
 "Andorra"
 "Angola"
 "Antigua and Barbuda"
 "Argentina"
 "Armenia"
 "Australia"
 "Austria"

 "Uruguay"
 "Uzbekistan"
 "Vanuatu"
 "Vatican"
 "Venezuela"
 "Viet Nam"
 "Yemen"
 "Zambia"
 "Zimbabwe"
```

Then, create the model and initialize the decision variables:

```
model = Model(GLPK.Optimizer)
@variable(model, x[C], Bin)
@objective(model, Min, sum(x))
@constraint(model, [d in C], passport_data[!, d]' * x >= 1)
model
```

```
A JuMP Model
Minimization problem with:
Variables: 199
Objective function type: AffExpr
`AffExpr`-in-`MathOptInterface.GreaterThan{Float64}`: 199 constraints
`VariableRef`-in-`MathOptInterface.ZeroOne`: 199 constraints
Model mode: AUTOMATIC
CachingOptimizer state: EMPTY_OPTIMIZER
Solver name: GLPK
Names registered in the model: x
```

Now optimize!

```
optimize!(model)
```

We can use the `solution_summary` function to get an overview of the solution:

```
solution_summary(model)
```

```
* Solver : GLPK

* Status
  Termination status : OPTIMAL
  Primal status      : FEASIBLE_POINT
  Dual status        : NO_SOLUTION
  Message from the solver:
  "Solution is optimal"

* Candidate solution
  Objective value      : 23.0
  Objective bound      : 23.0

* Work counters
  Solve time (sec)   : 0.00290
```

Solution

Let's have a look at the solution in more detail:

```
println("Minimum number of passports needed: ", objective_value(model))
```

```
Minimum number of passports needed: 23.0
```

```
println("Optimal passports:")
for c in C
    if value(x[c]) > 0.5
        println(" * ", c)
    end
end
```

```
Optimal passports:
 * Afghanistan
 * Angola
 * Australia
 * Austria
 * Comoros
 * Congo
 * Eritrea
 * Gambia
 * Georgia
 * Hong Kong
 * India
 * Iraq
 * Kenya
 * Madagascar
 * Maldives
 * North Korea
 * Papua New Guinea
 * Singapore
 * Somalia
 * Sri Lanka
 * Tunisia
 * United Arab Emirates
 * United States
```

Interesting! We need some passports, like Australia and the United States, which have widespread access to a large number of countries. However, we also need passports like North Korea which only have visa-free access to a very limited number of countries.

> **Note**
>
> We use `value(x[c]) > 0.5` rather than `value(x[c]) == 1` to avoid excluding solutions like `x[c] = 0.99999` that are "1" to some tolerance.

> **Tip**
>
> This tutorial was generated using Literate.jl. View the source `.jl` file on GitHub.

3.5 Performance tips

By now you should have read the other "getting started" tutorials. You're almost ready to write your own models, but before you do so there are some important things to be aware of.

Read the Julia performance tips

The first thing to do is read the Performance tips section of the Julia manual. The most important rule is to avoid global variables! This is particularly important if you're learning JuMP after using a language like MATLAB.

The "time-to-first-solve" issue

Similar to the infamous time-to-first-plot plotting problem, JuMP suffers from time-to-first-solve latency. This latency occurs because the first time you call JuMP code in each session, Julia needs to compile a lot of code specific to your problem. This issue is actively being worked on, but there are a few things you can do to improve things.

Don't call JuMP from the command line

In other languages, you might be used to a workflow like:

```
$ julia my_script.jl
```

This doesn't work for JuMP, because we have to pay the compilation latency every time you run the script. Instead, use one of the suggested workflows from the Julia documentation.

Disable bridges if none are being used

At present, the majority of the latency problems are caused by JuMP's bridging mechanism. If you only use constraints that are natively supported by the solver, you can disable bridges by passing `add_bridges = false` to `Model`.

```
model = Model(GLPK.Optimizer; add_bridges = false)
```

```
A JuMP Model
Feasibility problem with:
Variables: 0
Model mode: AUTOMATIC
CachingOptimizer state: EMPTY_OPTIMIZER
Solver name: GLPK
```

Use PackageCompiler

As a final option, consider using PackageCompiler.jl to create a custom sysimage.

This is a good option if you have finished prototyping a model, and you now want to call it frequently from the command line without paying the compilation price.

Use macros to build expressions

What

Use JuMP's macros (or `add_to_expression!`) to build expressions. Avoid constructing expressions outside the macros.

Why

Constructing an expression outside the macro results in intermediate copies of the expression. For example,

```
x[1] + x[2] + x[3]
```

is equivalent to

```
a = x[1]
b = a + x[2]
c = b + x[3]
```

Since we only care about c, the a and b expressions are not needed and constructing them slows the program down!

JuMP's macros rewrite the expressions to operate in-place and avoid these extra copies. Because they allocate less memory, they are faster, particularly for large expressions.

Example

```
model = Model()
@variable(model, x[1:3])
```

```
3-element Vector{VariableRef}:
 x[1]
 x[2]
 x[3]
```

Here's what happens if we construct the expression outside the macro:

```
@allocated x[1] + x[2] + x[3]
```

```
1456
```

> **Info**
>
> The `@allocated` measures how many bytes were allocated during the evaluation of an expression. Fewer is better.

If we use the `@expression` macro, we get many fewer allocations:

```
@allocated @expression(model, x[1] + x[2] + x[3])
```

```
912
```

> **Tip**
>
> This tutorial was generated using Literate.jl. View the source `.jl` file on GitHub.

Chapter 4

Linear programs

4.1 Tips and tricks

Originally Contributed by: Arpit Bhatia

> **Tip**
>
> A good source of tips is the Mosek Modeling Cookbook.

This tutorial collates some tips and tricks you can use when formulating mixed-integer programs. It uses the following packages:

```
using JuMP
```

Boolean Operators on Binary Variables

Binary variables can be used to construct logical operators. Here are some example.

OR

$$x_3 = x_1 \vee x_2$$

```
model = Model()
@variable(model, x[1:3], Bin)
@constraints(model, begin
    x[1] <= x[3]
    x[2] <= x[3]
    x[3] <= x[1] + x[2]
end)
```

And

$$x_3 = x_1 \wedge x_2$$

```
model = Model()
@variable(model, x[1:3], Bin)
@constraints(model, begin
    x[3] <= x[1]
    x[3] <= x[2]
    x[3] >= x[1] + x[2] - 1
end)
```

Not

$$x_1 \neg x_2$$

```
model = Model()
@variable(model, x[1:2], Bin)
@constraint(model, x[1] == 1 - x[2])
```

$$x_1 + x_2 = 1.0$$

Implies

$$x_1 \implies x_2$$

```
model = Model()
@variable(model, x[1:2], Bin)
@constraint(model, x[1] <= x[2])
```

$$x_1 - x_2 \leq 0.0$$

Big-M Disjunctive Constraints (OR)

Problem Suppose that we have two constraints $a^\top x \leq b$ and $c^\top x \leq d$, and we want at least one to hold.

Trick Introduce a "big-M" multiplied by a binary variable to relax one of the constraints.

Example Either $x_1 \leq 1$ and/or $x_2 \leq 2$.

```
model = Model()
@variable(model, x[1:2])
@variable(model, y, Bin)
M = 100
@constraint(model, x[1] <= 1 + M * y)
@constraint(model, x[2] <= 2 + M * (1 - y))
```

$$x_2 + 100y \leq 102.0$$

> **Warning**
>
> If `M` is too small, the solution may be suboptimal. If `M` is too big, the solver may encounter numerical issues. Try to use domain knowledge to choose an `M` that is just right. Gurobi has a good documentation section on this topic.

Indicator Constraints ($\implies$)

Problem

Suppose we want to model that a certain linear inequality must be satisfied when some other event occurs, i.e., for a binary variable z, we want to model the implication:

$$z = 1 \implies a^T x \leq b$$

Trick 1

Some solvers have native support for indicator constraints.

Example $x_1 + x_2 \leq 1$ if $z = 1$.

```
model = Model()
@variable(model, x[1:2])
@variable(model, z, Bin)
@constraint(model, z => {sum(x) <= 1})
```

$$z => x_1 + x_2 \leq 1.0$$

Example $x_1 + x_2 \leq 1$ if $z = 0$.

```
model = Model()
@variable(model, x[1:2])
@variable(model, z, Bin)
@constraint(model, !z => {sum(x) <= 1})
```

$$!z => x_1 + x_2 \leq 1.0$$

Trick 2

If the solver doesn't support indicator constraints, you an use the big-M trick.

Example $x_1 + x_2 \leq 1$ if $z = 1$.

```
model = Model()
@variable(model, x[1:2])
@variable(model, z, Bin)
M = 100
@constraint(model, sum(x) <= 1 + M * (1 - z))
```

$$x_1 + x_2 + 100z \leq 101.0$$

Example $x_1 + x_2 \leq 1$ if $z = 0$.

```
model = Model()
@variable(model, x[1:2])
@variable(model, z, Bin)
M = 100
@constraint(model, sum(x) <= 1 + M * z)
```

$$x_1 + x_2 - 100z \leq 1.0$$

Semi-Continuous Variables

A semi-continuous variable is a continuous variable between bounds $[l, u]$ that also can assume the value zero. ie. $x \in \{0\} \cup [l, u]$.

Example $x \in \{0\} \cup [1, 2]$

```
model = Model()
@variable(model, x in MOI.Semicontinuous(1.0, 2.0))
```

$$x$$

Semi-Integer Variables

A semi-integer variable is a variable which assumes integer values between bounds $[l, u]$ and can also assume the value zero: $x \in \{0\} \cup [l, u] \cap \mathbb{Z}$.

```
model = Model()
@variable(model, x in MOI.Semiinteger(5.0, 10.0))
```

$$x$$

Special Ordered Sets of Type I (SOS1)

A Special Ordered Set of Type I is a set of variables, at most one of which can take a non-zero value, all others being at 0.

They most frequently apply where a set of variables are actually binary variables. In other words, we have to choose at most one from a set of possibilities.

```
model = Model()
@variable(model, x[1:3], Bin)
@constraint(model, x in SOS1())
```

$$[x_1, x_2, x_3] \in \text{MathOptInterface.SOS1\{Float64\}([1.0, 2.0, 3.0])}$$

You can optionally pass SOS1 a weight vector like

```
@constraint(model, x in SOS1([0.2, 0.5, 0.3]))
```

$$[x_1, x_2, x_3] \in \text{MathOptInterface.SOS1\{Float64\}([0.2, 0.5, 0.3])}$$

If the decision variables are related and have a physical ordering, then the weight vector, although not used directly in the constraint, can help the solver make a better decision in the solution process.

Special Ordered Sets of Type II (SOS2)

A Special Ordered Set of type 2 is a set of non-negative variables, of which at most two can be non-zero, and if two are non-zero these must be consecutive in their ordering.

```
model = Model()
@variable(model, x[1:3])
@constraint(model, x in MOI.SOS2([3.0, 1.0, 2.0]))
```

$$[x_1, x_2, x_3] \in \text{MathOptInterface.SOS2\{Float64\}([3.0, 1.0, 2.0])}$$

The ordering provided by the weight vector is more important in this case as the variables need to be consecutive according to the ordering. For example, in the above constraint, the possible pairs are:

- Consecutive
  - (`x[1]` and `x[3]`) as they correspond to 3 and 2 resp. and thus can be non-zero
  - (`x[2]` and `x[3]`) as they correspond to 1 and 2 resp. and thus can be non-zero
- Non-consecutive
  - (`x[1]` and `x[2]`) as they correspond to 3 and 1 resp. and thus cannot be non-zero

Piecewise linear approximations

SOSII constraints are most often used to form piecewise linear approximations of a function.

Given a set of points for x:

```
x̂ = -1:0.5:2
```

```
-1.0:0.5:2.0
```

and a set of corresponding points for y:

```
ŷ = x̂ .^ 2
```

```
7-element Vector{Float64}:
 1.0
 0.25
 0.0
 0.25
 1.0
 2.25
 4.0
```

the piecewise linear approximation is constructed by representing x and y as convex combinations of $\hat{x}$ and $\hat{y}$.

```
N = length(x̂)
model = Model()
@variable(model, -1 <= x <= 2)
@variable(model, y)
@variable(model, 0 <= λ[1:N] <= 1)
@objective(model, Max, y)
@constraints(model, begin
    x == sum(x̂[i] * λ[i] for i in 1:N)
    y == sum(ŷ[i] * λ[i] for i in 1:N)
    sum(λ) == 1
    λ in SOS2()
end)
```

Tip

This tutorial was generated using Literate.jl. View the source `.jl` file on GitHub.

4.2 The diet problem

This tutorial solves the classic "diet problem", also known as the Stigler diet.

Required packages

```
using JuMP
import DataFrames
import GLPK
```

Formulation

Suppose we wish to cook a nutritionally balanced meal by choosing the quantity of each food f to eat from a set of foods F in our kitchen.

Each food f has a cost, c_f, as well as a macronutrient profile $a_{m,f}$ for each macronutrient $m \in M$.

Because we care about a nutritionally balanced meal, we set some minimum and maximum limits for each nutrient, which we denote l_m and u_m respectively.

Furthermore, because we are optimizers, we seek the minimum cost solution.

With a little effort, we can formulate our dinner problem as the following linear program:

$$\begin{aligned} \min \quad & \sum_{f \in F} c_f x_f \\ \text{s.t.} \quad & l_m \leq \sum_{f \in F} a_{m,f} x_f \leq u_m, && \forall m \in M \\ & x_f \geq 0, && \forall f \in F \end{aligned}$$

In the rest of this tutorial, we will create and solve this problem in JuMP, and learn what we should cook for dinner.

Data

First, we need some data for the problem:

```
foods = DataFrames.DataFrame(
    [
        "hamburger" 2.49 410 24 26 730
        "chicken" 2.89 420 32 10 1190
        "hot dog" 1.50 560 20 32 1800
        "fries" 1.89 380 4 19 270
        "macaroni" 2.09 320 12 10 930
        "pizza" 1.99 320 15 12 820
        "salad" 2.49 320 31 12 1230
        "milk" 0.89 100 8 2.5 125
        "ice cream" 1.59 330 8 10 180
    ],
    ["name", "cost", "calories", "protein", "fat", "sodium"],
)
```

	name	cost	calories	protein	fat	sodium
	Any	Any	Any	Any	Any	Any
1	hamburger	2.49	410	24	26	730
2	chicken	2.89	420	32	10	1190
3	hot dog	1.5	560	20	32	1800
4	fries	1.89	380	4	19	270
5	macaroni	2.09	320	12	10	930
6	pizza	1.99	320	15	12	820
7	salad	2.49	320	31	12	1230
8	milk	0.89	100	8	2.5	125
9	ice cream	1.59	330	8	10	180

Here, F is `foods.name` and c_f is `foods.cost`. (We're also playing a bit loose the term "macronutrient" by including calories and sodium.)

> **Tip**
>
> Although we hard-coded the data here, you could also read it in from a file. See Getting started with data and plotting for details.

We also need our minimum and maximum limits:

```
limits = DataFrames.DataFrame(
    [
        "calories" 1800 2200
        "protein" 91 Inf
        "fat" 0 65
        "sodium" 0 1779
    ],
    ["name", "min", "max"],
)
```

	name	min	max
	Any	Any	Any
1	calories	1800	2200
2	protein	91	Inf
3	fat	0	65
4	sodium	0	1779

JuMP formulation

Now we're ready to convert our mathematical formulation into a JuMP model.

First, create a new JuMP model. Since we have a linear program, we'll use GLPK as our optimizer:

```
model = Model(GLPK.Optimizer)
```

```
A JuMP Model
Feasibility problem with:
Variables: 0
Model mode: AUTOMATIC
CachingOptimizer state: EMPTY_OPTIMIZER
Solver name: GLPK
```

Next, we create a set of decision variables `x`, indexed over the foods in the `data` DataFrame. Each `x` has a lower bound of `0`.

```
@variable(model, x[foods.name] >= 0)
```

```
1-dimensional DenseAxisArray{VariableRef,1,...} with index sets:
    Dimension 1, Any["hamburger", "chicken", "hot dog", "fries", "macaroni", "pizza", "salad", "milk
     ", "ice cream"]
And data, a 9-element Vector{VariableRef}:
 x[hamburger]
 x[chicken]
 x[hot dog]
 x[fries]
 x[macaroni]
 x[pizza]
 x[salad]
 x[milk]
 x[ice cream]
```

Our objective is to minimze the total cost of purchasing food. We can write that as a sum over the rows in `data`.

```
@objective(
    model,
    Min,
    sum(food["cost"] * x[food["name"]] for food in eachrow(foods)),
)
```

$$2.49x_{hamburger}+2.89x_{chicken}+1.5x_{hotdog}+1.89x_{fries}+2.09x_{macaroni}+1.99x_{pizza}+2.49x_{salad}+0.89x_{milk}+1.59x_{icecream}$$

For the next component, we need to add a constraint that our total intake of each component is within the limits contained in the `limits` DataFrame. To make this more readable, we introduce a JuMP `@expression`

```
for limit in eachrow(limits)
    intake = @expression(
        model,
        sum(food[limit["name"]] * x[food["name"]] for food in eachrow(foods)),
    )
    @constraint(model, limit.min <= intake <= limit.max)
end
```

What does our model look like?

```
print(model)
```

```
Min 2.49 x[hamburger] + 2.89 x[chicken] + 1.5 x[hot dog] + 1.89 x[fries] + 2.09 x[macaroni] + 1.99 x
     [pizza] + 2.49 x[salad] + 0.89 x[milk] + 1.59 x[ice cream]
Subject to
 410 x[hamburger] + 420 x[chicken] + 560 x[hot dog] + 380 x[fries] + 320 x[macaroni] + 320 x[pizza]
     + 320 x[salad] + 100 x[milk] + 330 x[ice cream] ∈ [1800.0, 2200.0]
 24 x[hamburger] + 32 x[chicken] + 20 x[hot dog] + 4 x[fries] + 12 x[macaroni] + 15 x[pizza] + 31 x[
     salad] + 8 x[milk] + 8 x[ice cream] ∈ [91.0, Inf]
 26 x[hamburger] + 10 x[chicken] + 32 x[hot dog] + 19 x[fries] + 10 x[macaroni] + 12 x[pizza] + 12 x
     [salad] + 2.5 x[milk] + 10 x[ice cream] ∈ [0.0, 65.0]
 730 x[hamburger] + 1190 x[chicken] + 1800 x[hot dog] + 270 x[fries] + 930 x[macaroni] + 820 x[pizza
     ] + 1230 x[salad] + 125 x[milk] + 180 x[ice cream] ∈ [0.0, 1779.0]
 x[hamburger] ≥ 0.0
 x[chicken] ≥ 0.0
 x[hot dog] ≥ 0.0
 x[fries] ≥ 0.0
 x[macaroni] ≥ 0.0
 x[pizza] ≥ 0.0
 x[salad] ≥ 0.0
 x[milk] ≥ 0.0
 x[ice cream] ≥ 0.0
```

Solution

```
optimize!(model)


solution_summary(model)
```

```
* Solver : GLPK

* Status
  Termination status : OPTIMAL
  Primal status      : FEASIBLE_POINT
  Dual status        : FEASIBLE_POINT
  Message from the solver:
  "Solution is optimal"

* Candidate solution
  Objective value      : 11.82886111111111
  Objective bound      : -Inf
  Dual objective value : 11.82886111111111
```

```
* Work counters
  Solve time (sec)   : 0.00006
```

Success! We found an optimal solution. Let's see what the optimal solution is:

```
for food in foods.name
    println(food, " = ", value(x[food]))
end
```

```
hamburger = 0.6045138888888888
chicken = 0.0
hot dog = 0.0
fries = 0.0
macaroni = 0.0
pizza = 0.0
salad = 0.0
milk = 6.9701388888888935
ice cream = 2.591319444444441
```

That's a lot of milk and ice cream! And sadly, we only get `0.6` of a hamburger.

Problem modification

JuMP makes it easy to take an existing model and modify it by adding extra constraints. Let's see what happens if we add a constraint that we can buy at most 6 units of milk or ice cream combined.

```
@constraint(model, x["milk"] + x["ice cream"] <= 6)
```

$$x_{milk} + x_{icecream} \leq 6.0$$

```
optimize!(model)


solution_summary(model)
```

```
* Solver : GLPK

* Status
  Termination status : INFEASIBLE
  Primal status      : NO_SOLUTION
  Dual status        : INFEASIBILITY_CERTIFICATE
  Message from the solver:
  "No feasible primal-dual solution exists."

* Candidate solution
  Objective value      : 11.828861111111111
  Objective bound      : -Inf
  Dual objective value : 3.5614583333333307

* Work counters
  Solve time (sec)   : 0.00005
```

Uh oh! The exists no feasible solution to our problem. Looks like we're stuck eating ice cream for dinner.

Tip

This tutorial was generated using Literate.jl. View the source `.jl` file on GitHub.

4.3 The cannery problem

Original author: Louis Luangkesorn, January 30, 2015.

This tutorial solves the cannery problem from Dantzig, Linear Programming and Extensions, Princeton University Press, Princeton, NJ, 1963. This class of problem is known as a transshipment problem.

The purpose of this tutorial is to demonstrate how to use JSON data in the formulation of a JuMP model.

Required packages

```
using JuMP
import GLPK
import JSON
```

Formulation

The cannery problem assumes we are optimizing the shipment of cans from production plants $p \in P$ to markets $m \in M$.

Each production plant p has a capacity, c_p, and each market m has a demand d_m. The distance from plant to market is $d_{p,m}$.

With a little effort, we can formulate our problem as the following linear program:

$$\begin{aligned}
\min & \sum_{p \in P} \sum_{m \in M} d_{p,m} x_{p,m} \\
\text{s.t.} & \sum_{m \in M} x_{p,m} \leq c_p, & \forall p \in P \\
& \sum_{p \in P} x_{p,m} \geq d_m, & \forall m \in M \\
& x_{p,m} \geq 0, & \forall p \in P, m \in M
\end{aligned}$$

Data

A key feature of the tutorial is to demonstrate how to load data from JSON.

For simplicity, we've hard-coded it below. But if the data was available as a `.json` file, we could use `data = JSON.parsefile(filename)` to read in the data.

```
data = JSON.parse("""
{
    "plants": {
        "Seattle": {"capacity": 350},
        "San-Diego": {"capacity": 600}
    },
```

```
    "markets": {
        "New-York": {"demand": 300},
        "Chicago": {"demand": 300},
        "Topeka": {"demand": 300}
    },
    "distances": {
        "Seattle => New-York": 2.5,
        "Seattle => Chicago": 1.7,
        "Seattle => Topeka": 1.8,
        "San-Diego => New-York": 2.5,
        "San-Diego => Chicago": 1.8,
        "San-Diego => Topeka": 1.4
    }
}
""")
```

```
Dict{String, Any} with 3 entries:
  "plants"    => Dict{String, Any}("Seattle"=>Dict{String, Any}("capacity…"=>350
  "distances" => Dict{String, Any}("San-Diego => New-York"=>2.5, "Seattle => …To
  "markets"   => Dict{String, Any}("Chicago"=>Dict{String, Any}("demand"=>300)…,
```

Create the set of plants:

```
P = keys(data["plants"])
```

```
KeySet for a Dict{String, Any} with 2 entries. Keys:
  "Seattle"
  "San-Diego"
```

Create the set of markets:

```
M = keys(data["markets"])
```

```
KeySet for a Dict{String, Any} with 3 entries. Keys:
  "Chicago"
  "Topeka"
  "New-York"
```

We also need a function to compute the distance from plant to market:

```
distance(p::String, m::String) = data["distances"]["$(p) => $(m)"]
```

```
distance (generic function with 1 method)
```

JuMP formulation

Now we're ready to convert our mathematical formulation into a JuMP model.

First, create a new JuMP model. Since we have a linear program, we'll use GLPK as our optimizer:

```
model = Model(GLPK.Optimizer)
```

```
A JuMP Model
Feasibility problem with:
Variables: 0
Model mode: AUTOMATIC
CachingOptimizer state: EMPTY_OPTIMIZER
Solver name: GLPK
```

Our decision variables are indexed over the set of plants and markets:

```
@variable(model, x[P, M] >= 0)
```

```
2-dimensional DenseAxisArray{VariableRef,2,...} with index sets:
    Dimension 1, ["Seattle", "San-Diego"]
    Dimension 2, ["Chicago", "Topeka", "New-York"]
And data, a 2×3 Matrix{VariableRef}:
 x[Seattle,Chicago]    x[Seattle,Topeka]    x[Seattle,New-York]
 x[San-Diego,Chicago]  x[San-Diego,Topeka]  x[San-Diego,New-York]
```

We need a constraint that each plant can ship no more than its capacity:

```
@constraint(model, [p in P], sum(x[p, :]) <= data["plants"][p]["capacity"])
```

```
1-dimensional DenseAxisArray{ConstraintRef{Model, MathOptInterface.ConstraintIndex{MathOptInterface.
     ScalarAffineFunction{Float64}, MathOptInterface.LessThan{Float64}}, ScalarShape},1,...} with
     index sets:
    Dimension 1, ["Seattle", "San-Diego"]
And data, a 2-element Vector{ConstraintRef{Model, MathOptInterface.ConstraintIndex{MathOptInterface.
     ScalarAffineFunction{Float64}, MathOptInterface.LessThan{Float64}}, ScalarShape}}:
 x[Seattle,Chicago] + x[Seattle,Topeka] + x[Seattle,New-York] ≤ 350.0
 x[San-Diego,Chicago] + x[San-Diego,Topeka] + x[San-Diego,New-York] ≤ 600.0
```

and each market must receive at least its demand:

```
@constraint(model, [m in M], sum(x[:, m]) >= data["markets"][m]["demand"])
```

```
1-dimensional DenseAxisArray{ConstraintRef{Model, MathOptInterface.ConstraintIndex{MathOptInterface.
     ScalarAffineFunction{Float64}, MathOptInterface.GreaterThan{Float64}}, ScalarShape},1,...} with
     index sets:
    Dimension 1, ["Chicago", "Topeka", "New-York"]
And data, a 3-element Vector{ConstraintRef{Model, MathOptInterface.ConstraintIndex{MathOptInterface.
     ScalarAffineFunction{Float64}, MathOptInterface.GreaterThan{Float64}}, ScalarShape}}:
 x[Seattle,Chicago] + x[San-Diego,Chicago] ≥ 300.0
 x[Seattle,Topeka] + x[San-Diego,Topeka] ≥ 300.0
 x[Seattle,New-York] + x[San-Diego,New-York] ≥ 300.0
```

Finally, our objective is to minimize the transportation distance:

```
@objective(model, Min, sum(distance(p, m) * x[p, m] for p in P, m in M))
```

$$1.7x_{Seattle,Chicago}+1.8x_{Seattle,Topeka}+2.5x_{Seattle,New-York}+1.8x_{San-Diego,Chicago}+1.4x_{San-Diego,Topeka}+2.5x_{San-D}$$

Solution

```
optimize!(model)

solution_summary(model)
```

```
* Solver : GLPK

* Status
  Termination status : OPTIMAL
  Primal status      : FEASIBLE_POINT
  Dual status        : FEASIBLE_POINT
  Message from the solver:
  "Solution is optimal"

* Candidate solution
  Objective value      : 1680.0
  Objective bound      : -Inf
  Dual objective value : 1680.0

* Work counters
  Solve time (sec)   : 0.00004
```

What's the optimal shipment?

```
println("RESULTS:")
for p in P, m in M
    println(p, " => ", m, ": ", value(x[p, m]))
end
```

```
RESULTS:
Seattle => Chicago: 300.0
Seattle => Topeka: 0.0
Seattle => New-York: 0.0
San-Diego => Chicago: 0.0
San-Diego => Topeka: 300.0
San-Diego => New-York: 300.0
```

Tip

This tutorial was generated using Literate.jl. View the source `.jl` file on GitHub.

4.4 Facility Location

It was originally contributed by Mathieu Tanneau (@mtanneau) and Alexis Montoison (@amontoison).

```
using JuMP
import GLPK
import LinearAlgebra
import Plots
import Random
```

Uncapacitated facility location

Problem description

We are given

- A set $M = \{1, \ldots, m\}$ of clients
- A set $N = \{1, \ldots, n\}$ of sites where a facility can be built

Decision variables Decision variables are split into two categories:

- Binary variable y_j indicates whether facility j is built or not
- Binary variable $x_{i,j}$ indicates whether client i is assigned to facility j

Objective The objective is to minimize the total cost of serving all clients. This costs breaks down into two components:

- Fixed cost of building a facility.

In this example, this cost is $f_j = 1,\ \forall j$.

- Cost of serving clients from the assigned facility.

In this example, the cost $c_{i,j}$ of serving client i from facility j is the Euclidean distance between the two.

Constraints

- Each customer must be served by exactly one facility
- A facility cannot serve any client unless it is open

MILP formulation

The problem can be formulated as the following MILP:

$$
\begin{aligned}
\min_{x,y} \quad & \sum_{i,j} c_{i,j} x_{i,j} + \sum_{j} f_j y_j \\
s.t. & \sum_{j} x_{i,j} = 1, && \forall i \in M \\
& x_{i,j} \leq y_j, && \forall i \in M, j \in N \\
& x_{i,j}, y_j \in \{0, 1\}, && \forall i \in M, j \in N
\end{aligned}
$$

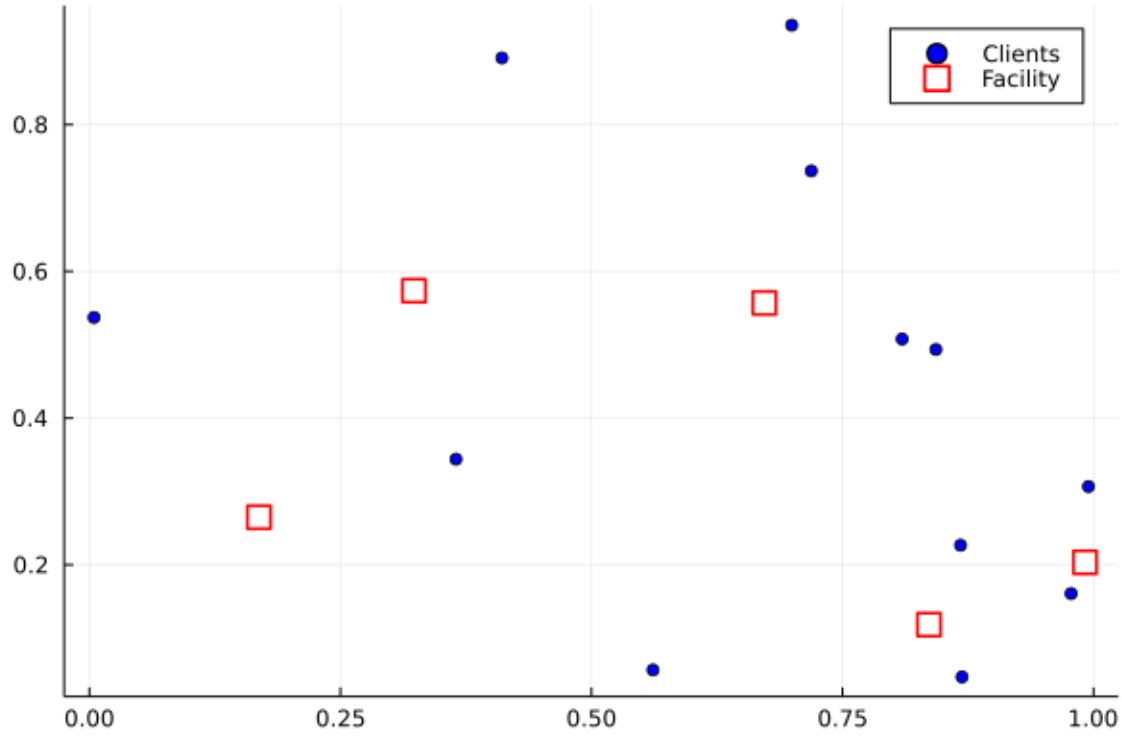
where the first set of constraints ensures that each client is served exactly once, and the second set of constraints ensures that no client is served from an unopened facility.

Problem data

```
Random.seed!(314)

# number of clients
m = 12
# number of facility locations
n = 5

# Clients' locations
Xc = rand(m)
Yc = rand(m)

# Facilities' potential locations
Xf = rand(n)
Yf = rand(n)

# Fixed costs
f = ones(n);

# Distance
c = zeros(m, n)
for i in 1:m
    for j in 1:n
        c[i, j] = LinearAlgebra.norm([Xc[i] - Xf[j], Yc[i] - Yf[j]], 2)
    end
end
```

Display the data

```
Plots.scatter(
    Xc,
    Yc,
    label = "Clients",
    markershape = :circle,
    markercolor = :blue,
)
Plots.scatter!(
    Xf,
    Yf,
    label = "Facility",
    markershape = :square,
    markercolor = :white,
    markersize = 6,
    markerstrokecolor = :red,
    markerstrokewidth = 2,
)
```



JuMP implementation

Create a JuMP model

```
ufl = Model(GLPK.Optimizer)
```

```
A JuMP Model
Feasibility problem with:
Variables: 0
Model mode: AUTOMATIC
CachingOptimizer state: EMPTY_OPTIMIZER
Solver name: GLPK
```

Variables

```
@variable(ufl, y[1:n], Bin);
@variable(ufl, x[1:m, 1:n], Bin);
```

Each client is served exactly once

```
@constraint(ufl, client_service[i in 1:m], sum(x[i, j] for j in 1:n) == 1);
```

A facility must be open to serve a client

```
@constraint(ufl, open_facility[i in 1:m, j in 1:n], x[i, j] <= y[j]);
```

Objective

```
@objective(ufl, Min, f'y + sum(c .* x));
```

Solve the uncapacitated facility location problem with GLPK

```
optimize!(ufl)
println("Optimal value: ", objective_value(ufl))
```

```
Optimal value: 5.226417970467934
```

Visualizing the solution

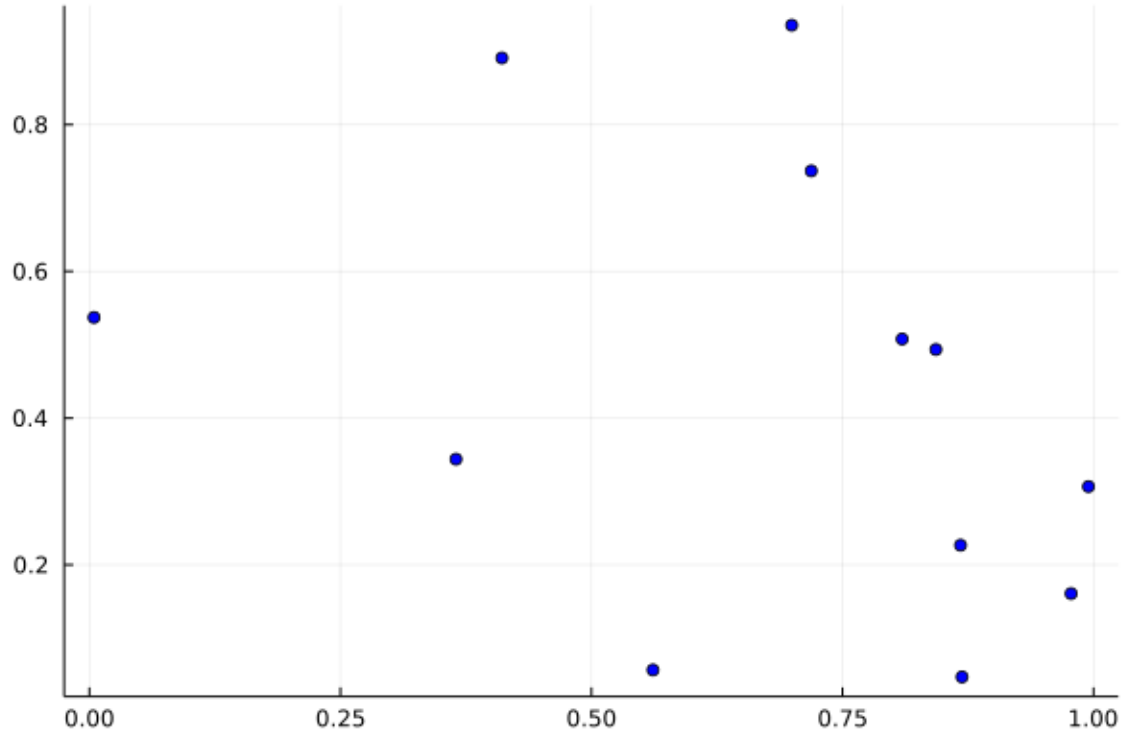
The threshold 1e-5 ensure that edges between clients and facilities are drawn when x[i, j] ≈ 1.

```
x_ = value.(x) .> 1 - 1e-5
y_ = value.(y) .> 1 - 1e-5
```

```
5-element BitVector:
 0
 0
 1
 1
 0
```

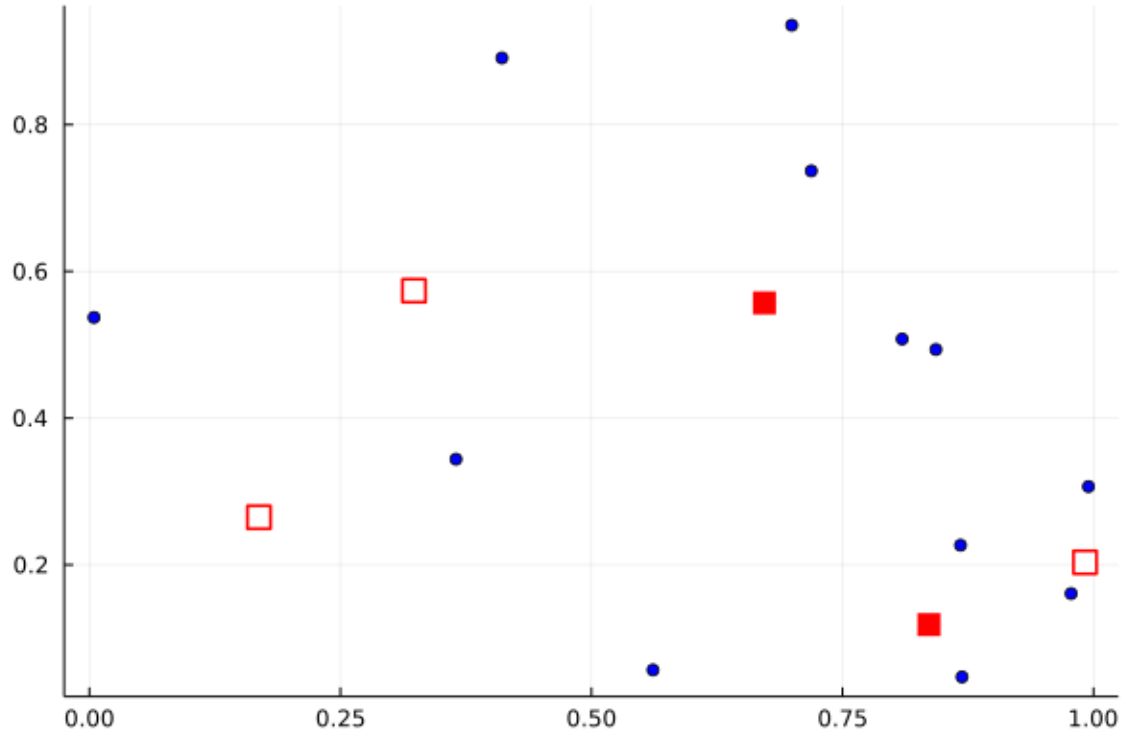
Display clients

```
p = Plots.scatter(
    Xc,
    Yc,
    markershape = :circle,
    markercolor = :blue,
    label = nothing,
)
```



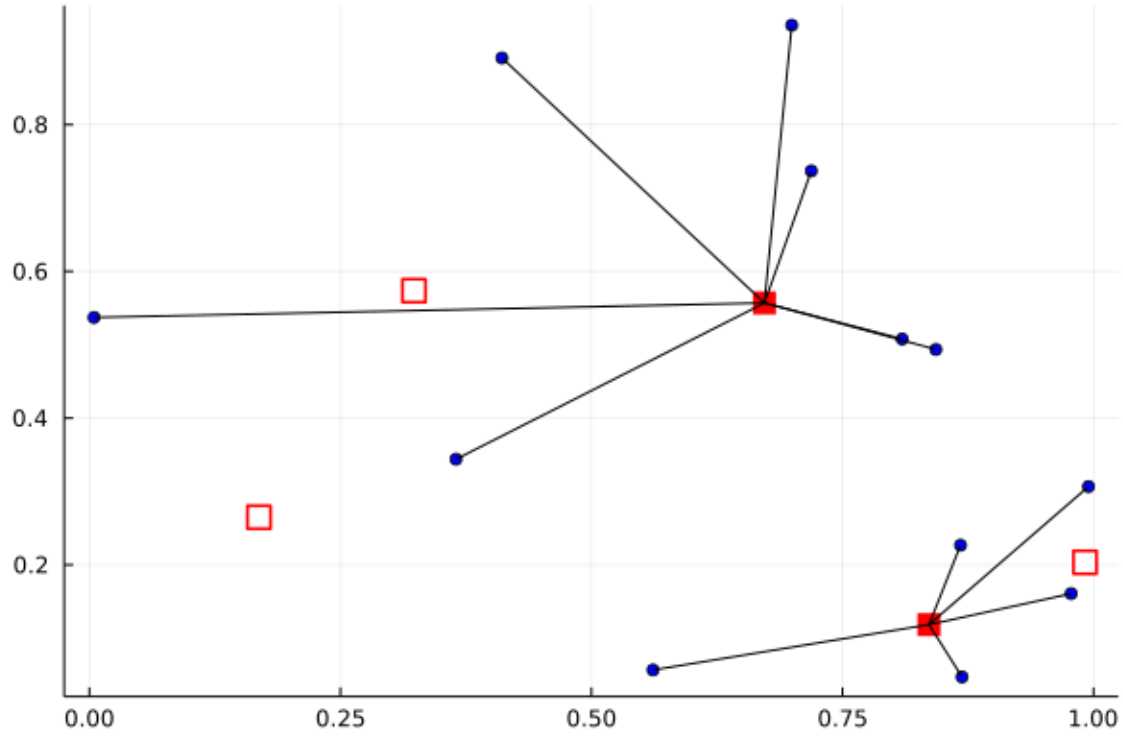
Show open facility

```
mc = [(y_[j] ? :red : :white) for j in 1:n]
Plots.scatter!(
    Xf,
    Yf,
    markershape = :square,
    markercolor = mc,
    markersize = 6,
    markerstrokecolor = :red,
    markerstrokewidth = 2,
    label = nothing,
)
```



Show client-facility assignment

```
for i in 1:m
    for j in 1:n
        if x_[i, j] == 1
            Plots.plot!(
                [Xc[i], Xf[j]],
                [Yc[i], Yf[j]],
                color = :black,
                label = nothing,
            )
        end
    end
end

p
```



Capacitated Facility location

Problem formulation

The capacitated variant introduces a capacity constraint on each facility, i.e., clients have a certain level of demand to be served, while each facility only has finite capacity which cannot be exceeded.

Specifically,

- The demand of client i is denoted by $a_i \geq 0$
- The capacity of facility j is denoted by $q_j \geq 0$

The capacity constraints then write

$$\sum_i a_i x_{i,j} \leq q_j y_j \quad \forall j \in N$$

Note that, if y_j is set to 0, the capacity constraint above automatically forces $x_{i,j}$ to 0.

Thus, the capacitated facility location can be formulated as follows

$$
\begin{aligned}
\min_{x,y} \quad & \sum_{i,j} c_{i,j} x_{i,j} + \sum_{j} f_j y_j \\
s.t. & \sum_{j} x_{i,j} = 1, && \forall i \in M \\
& \sum_{i} a_i x_{i,j} \leq q_j y_j, && \forall j \in N \\
& x_{i,j}, y_j \in \{0,1\}, && \forall i \in M, j \in N
\end{aligned}
$$

For simplicity, we will assume that there is enough capacity to serve the demand, i.e., there exists at least one feasible solution.

Demands

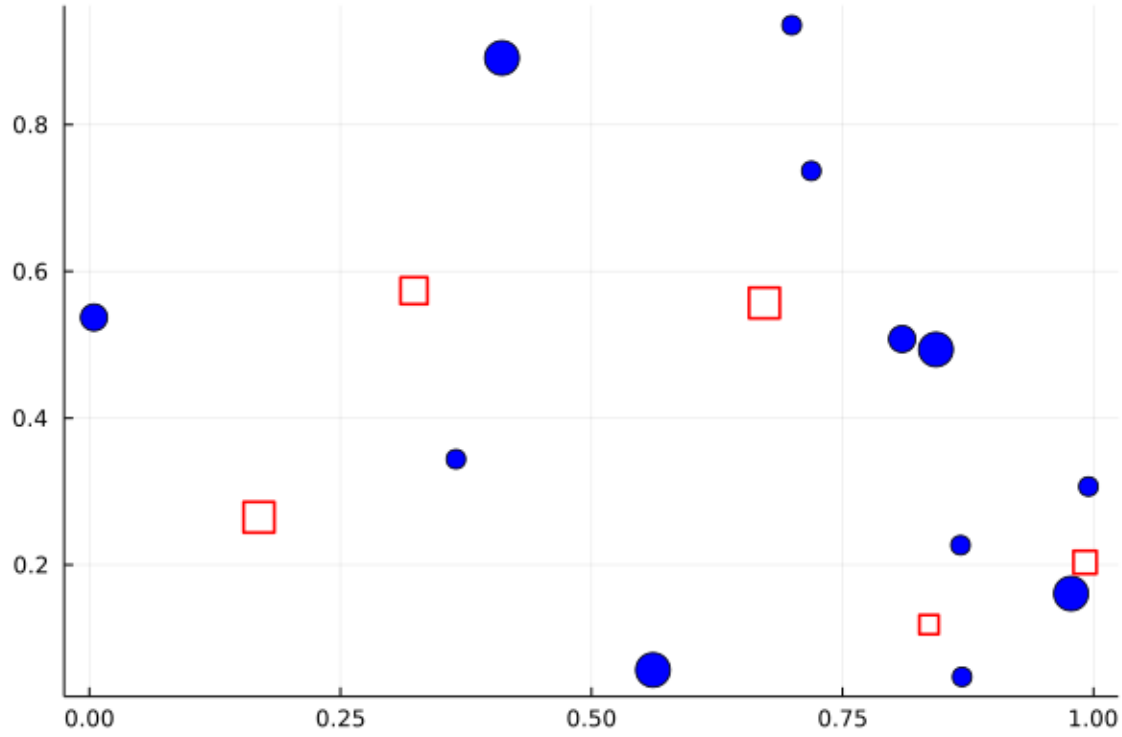
```
a = rand(1:3, m);
```

Capacities

```
q = rand(5:10, n);
```

Display the data

```
Plots.scatter(
    Xc,
    Yc,
    label = nothing,
    markershape = :circle,
    markercolor = :blue,
    markersize = 2 .* (2 .+ a),
)

Plots.scatter!(
    Xf,
    Yf,
    label = nothing,
    markershape = :rect,
    markercolor = :white,
    markersize = q,
    markerstrokecolor = :red,
    markerstrokewidth = 2,
)
```



JuMP implementation

Create a JuMP model

```
cfl = Model(GLPK.Optimizer)
```

```
A JuMP Model
Feasibility problem with:
Variables: 0
Model mode: AUTOMATIC
CachingOptimizer state: EMPTY_OPTIMIZER
Solver name: GLPK
```

Variables

```
@variable(cfl, y[1:n], Bin);
@variable(cfl, x[1:m, 1:n], Bin);
```

Each client is served exactly once

```
@constraint(cfl, client_service[i in 1:m], sum(x[i, :]) == 1);
```

Capacity constraint

```
@constraint(cfl, capacity, x'a .<= (q .* y));
```

Objective

```
@objective(cfl, Min, f'y + sum(c .* x));
```

Solve the problem

```
optimize!(cfl)
println("Optimal value: ", objective_value(cfl))
```

```
Optimal value: 6.17371506253207
```

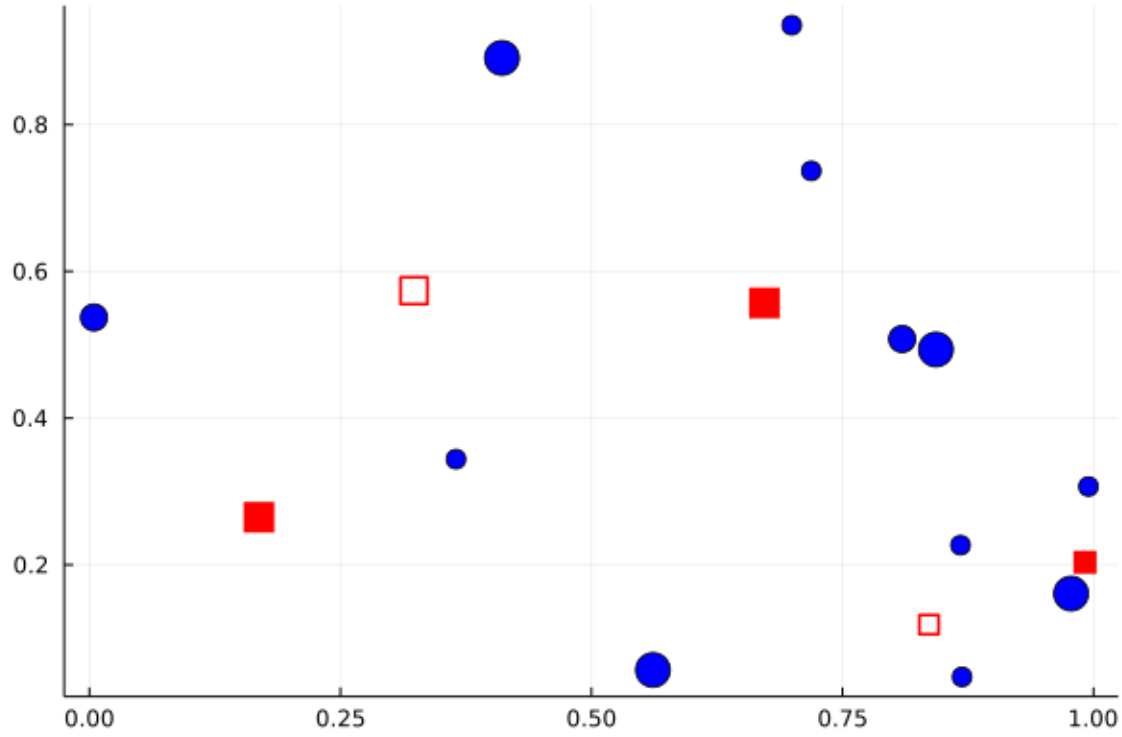
Visualizing the solution

The threshold 1e-5 ensure that edges between clients and facilities are drawn when x[i, j] ≈ 1.

```
x_ = value.(x) .> 1 - 1e-5;
y_ = value.(y) .> 1 - 1e-5;
```

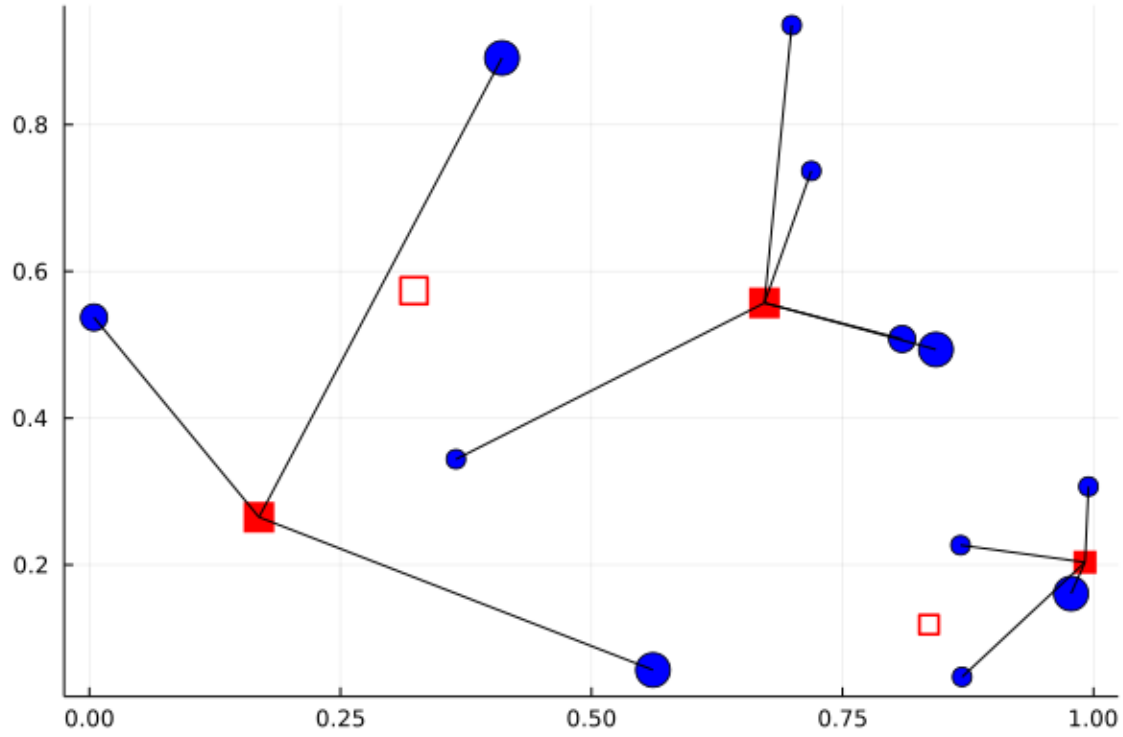
Display the solution

```
p = Plots.scatter(
    Xc,
    Yc,
    label = nothing,
    markershape = :circle,
    markercolor = :blue,
    markersize = 2 .* (2 .+ a),
)

mc = [(y_[j] ? :red : :white) for j in 1:n]
Plots.scatter!(
    Xf,
    Yf,
    label = nothing,
    markershape = :rect,
    markercolor = mc,
    markersize = q,
    markerstrokecolor = :red,
    markerstrokewidth = 2,
)
```



Show client-facility assignment

```
for i in 1:m
    for j in 1:n
        if x_[i, j] == 1
            Plots.plot!(
                [Xc[i], Xf[j]],
                [Yc[i], Yf[j]],
                color = :black,
                label = nothing,
            )
            break
        end
    end
end
p
```



Further Reading

- Benders decomposition is a method of choice for solving facility location problems.
- Benchmark instances can be found here.

Tip

This tutorial was generated using Literate.jl. View the source `.jl` file on GitHub.

4.5 The factory schedule example

This is a Julia translation of part 5 from "Introduction to to Linear Programming with Python" available at https://github.com/benalexkeen/Introduction-to-linear-programming

For 2 factories (A, B), minimize the cost of production over the course of 12 months while meeting monthly demand. Factory B has a planned outage during month 5.

It was originally contributed by @Crghilardi.

```
using JuMP
import GLPK
import Test

function example_factory_schedule()
```

```
# Sets in the problem:
months, factories = 1:12, [:A, :B]
# This function takes a matrix and converts it to a JuMP container so we can
# refer to elements such as `d_max_cap[1, :A]`.
containerize(A::Matrix) = Containers.DenseAxisArray(A, months, factories)
# Maximum production capacity in (month, factory) [units/month]:
d_max_cap = containerize(
    [
        100000 50000
        110000 55000
        120000 60000
        145000 100000
        160000 0
        140000 70000
        155000 60000
        200000 100000
        210000 100000
        197000 100000
        80000 120000
        150000 150000
    ],
)
# Minimum production capacity in (month, factory) [units/month]:
d_min_cap = containerize(
    [
        20000 20000
        20000 20000
        20000 20000
        20000 20000
        20000 0
        20000 20000
        20000 20000
        20000 20000
        20000 20000
        20000 20000
        20000 20000
        20000 20000
    ],
)
# Variable cost of production in (month, factory) [$/unit]:
d_var_cost = containerize([
    10 5
    11 4
    12 3
    9 5
    8 0
    8 6
    5 4
    7 6
    9 8
    10 11
    8 10
    8 12
])
# Fixed cost of production in (month, factory) # [$/month]:
d_fixed_cost = containerize(
```

```
        [
            500 600
            500 600
            500 600
            500 600
            500 0
            500 600
            500 600
            500 600
            500 600
            500 600
            500 600
            500 600
        ],
    )
    # Demand in each month [units/month]:
    d_demand = [
        120_000,
        100_000,
        130_000,
        130_000,
        140_000,
        130_000,
        150_000,
        170_000,
        200_000,
        190_000,
        140_000,
        100_000,
    ]
    # The model!
    model = Model(GLPK.Optimizer)
    # Decision variables
    @variables(model, begin
        status[m in months, f in factories], Bin
        production[m in months, f in factories], Int
    end)
    # The production cannot be less than minimum capacity.
    @constraint(
        model,
        [m in months, f in factories],
        production[m, f] >= d_min_cap[m, f] * status[m, f],
    )
    # The production cannot be more that maximum capacity.
    @constraint(
        model,
        [m in months, f in factories],
        production[m, f] <= d_max_cap[m, f] * status[m, f],
    )
    # The production must equal demand in a given month.
    @constraint(model, [m in months], sum(production[m, :]) == d_demand[m])
    # Factory B is shut down during month 5, so production and status are both
    # zero.
    fix(status[5, :B], 0.0)
    fix(production[5, :B], 0.0)
    # The objective is to minimize the cost of production across all time
```

```
    ##periods.
    @objective(
        model,
        Min,
        sum(
            d_fixed_cost[m, f] * status[m, f] +
            d_var_cost[m, f] * production[m, f] for m in months, f in factories
        )
    )
    # Optimize the problem
    optimize!(model)
    # Check the solution!
    Test.@testset "Check the solution against known optimal" begin
        Test.@test termination_status(model) == OPTIMAL
        Test.@test objective_value(model) == 12_906_400.0
        Test.@test value.(production)[1, :A] == 70_000
        Test.@test value.(status)[1, :A] == 1
        Test.@test value.(status)[5, :B] == 0
        Test.@test value.(production)[5, :B] == 0
    end
    println("The production schedule is:")
    println(value.(production))
    return
end

example_factory_schedule()
```

```
Test Summary:                           | Pass  Total
Check the solution against known optimal |    6      6
The production schedule is:
2-dimensional DenseAxisArray{Float64,2,...} with index sets:
    Dimension 1, 1:12
    Dimension 2, [:A, :B]
And data, a 12×2 Matrix{Float64}:
  70000.0   50000.0
  45000.0   55000.0
  70000.0   60000.0
  30000.0  100000.0
 140000.0       0.0
  60000.0   70000.0
  90000.0   60000.0
  70000.0  100000.0
 100000.0  100000.0
 190000.0       0.0
  80000.0   60000.0
 100000.0       0.0
```

Tip

This tutorial was generated using Literate.jl. View the source `.jl` file on GitHub.

4.6 Finance

Originally Contributed by: Arpit Bhatia

Optimization models play an increasingly important role in financial decisions. Many computational finance problems can be solved efficiently using modern optimization techniques.

In this tutorial we will discuss two such examples taken from the book Optimization Methods in Finance.

This tutorial uses the following packages

```
using JuMP
import GLPK
```

Short Term Financing

Corporations routinely face the problem of financing short term cash commitments such as the following:

Month	Jan	Feb	Mar	Apr	May	Jun
Net Cash Flow	-150	-100	200	-200	50	300

Net cash flow requirements are given in thousands of dollars. The company has the following sources of funds:

- A line of credit of up to $100K at an interest rate of 1% per month,
- In any one of the first three months, it can issue 90-day commercial paper bearing a total interest of 2% for the 3-month period,
- Excess funds can be invested at an interest rate of 0.3% per month.

Our task is to find out the most economical way to use these 3 sources such that we end up with the most amount of money at the end of June.

We model this problem in the following manner:

We will use the following decision variables:

- the amount u_i drawn from the line of credit in month i
- the amount v_i of commercial paper issued in month i
- the excess funds w_i in month i

Here we have three types of constraints:

1. for every month, cash inflow = cash outflow for each month
2. upper bounds on u_i
3. nonnegativity of the decision variables u_i, v_i and w_i.

Our objective will be to simply maximize the company's wealth in June, which say we represent with the variable m.

```
financing = Model(GLPK.Optimizer)

@variables(financing, begin
    0 <= u[1:5] <= 100
    0 <= v[1:3]
    0 <= w[1:5]
    m
end)

@objective(financing, Max, m)

@constraints(
    financing,
    begin
        u[1] + v[1] - w[1] == 150 # January
        u[2] + v[2] - w[2] - 1.01u[1] + 1.003w[1] == 100 # February
        u[3] + v[3] - w[3] - 1.01u[2] + 1.003w[2] == -200 # March
        u[4] - w[4] - 1.02v[1] - 1.01u[3] + 1.003w[3] == 200 # April
        u[5] - w[5] - 1.02v[2] - 1.01u[4] + 1.003w[4] == -50 # May
        -m - 1.02v[3] - 1.01u[5] + 1.003w[5] == -300 # June
    end
)

optimize!(financing)

objective_value(financing)
```

```
92.49694915254233
```

Combinatorial Auctions

In many auctions, the value that a bidder has for a set of items may not be the sum of the values that he has for individual items.

Examples are equity trading, electricity markets, pollution right auctions and auctions for airport landing slots.

To take this into account, combinatorial auctions allow the bidders to submit bids on combinations of items.

Let $M = \{1, 2, \ldots, m\}$ be the set of items that the auctioneer has to sell. A bid is a pair $B_j = (S_j, p_j)$ where $S_j \subseteq M$ is a nonempty set of items and p_j is the price offer for this set.

Suppose that the auctioneer has received n bids $B_1, B_2, \ldots, B_n$. The goal of this problem is to help an auctioneer determine the winners in order to maximize his revenue.

We model this problem by taking a decision variable y_j for every bid. We add a constraint that each item i is sold at most once. This gives us the following model:

$$\begin{aligned} \max \quad & \sum_{i=1}^{n} p_j y_j \\ \text{s.t.} \quad & \sum_{j:i \in S_j} y_j \leq 1 \quad \forall i = \{1, 2 \ldots m\} \\ & y_j \in \{0, 1\} \quad \forall j \in \{1, 2 \ldots n\} \end{aligned}$$

```
bid_values = [6 3 12 12 8 16]
bid_items = [[1], [2], [3 4], [1 3], [2 4], [1 3 4]]

auction = Model(GLPK.Optimizer)
@variable(auction, y[1:6], Bin)
@objective(auction, Max, sum(y' .* bid_values))
for i in 1:6
    @constraint(auction, sum(y[j] for j in 1:6 if i in bid_items[j]) <= 1)
end

optimize!(auction)

objective_value(auction)
```

```
21.0
```

```
value.(y)
```

```
6-element Vector{Float64}:
 1.0
 1.0
 1.0
 0.0
 0.0
 0.0
```

Tip

This tutorial was generated using Literate.jl. View the source `.jl` file on GitHub.

4.7 Geographical Clustering

Originally Contributed by: Matthew Helm (with help from Mathieu Tanneau on Julia Discourse)

The goal of this exercise is to cluster n cities into k groups, minimizing the total pairwise distance between cities and ensuring that the variance in the total populations of each group is relatively small.

This tutorial uses the following packages:

```
using JuMP
import DataFrames
import GLPK
import LinearAlgebra
```

For this example, we'll use the 20 most populous cities in the United States.

```
cities = DataFrames.DataFrame(
    city = [
        "New York, NY",
        "Los Angeles, CA",
        "Chicago, IL",
```

```
        "Houston, TX",
        "Philadelphia, PA",
        "Phoenix, AZ",
        "San Antonio, TX",
        "San Diego, CA",
        "Dallas, TX",
        "San Jose, CA",
        "Austin, TX",
        "Indianapolis, IN",
        "Jacksonville, FL",
        "San Francisco, CA",
        "Columbus, OH",
        "Charlotte, NC",
        "Fort Worth, TX",
        "Detroit, MI",
        "El Paso, TX",
        "Memphis, TN",
    ],
    population = [
        8.405,
        3.884,
        2.718,
        2.195,
        1.553,
        1.513,
        1.409,
        1.355,
        1.257,
        0.998,
        0.885,
        0.843,
        0.842,
        0.837,
        0.822,
        0.792,
        0.792,
        0.688,
        0.674,
        0.653,
    ],
    lat = [
        40.7127,
        34.0522,
        41.8781,
        29.7604,
        39.9525,
        33.4483,
        29.4241,
        32.7157,
        32.7766,
        37.3382,
        30.2671,
        39.7684,
        30.3321,
        37.7749,
        39.9611,
```

```
        35.2270,
        32.7554,
        42.3314,
        31.7775,
        35.1495,
    ],
    lon = [
        -74.0059,
        -118.2436,
        -87.6297,
        -95.3698,
        -75.1652,
        -112.0740,
        -98.4936,
        -117.1610,
        -96.7969,
        -121.8863,
        -97.7430,
        -86.1580,
        -81.6556,
        -122.4194,
        -82.9987,
        -80.8431,
        -97.3307,
        -83.0457,
        -106.4424,
        -90.0489,
    ],
)
```

	city	population	lat	lon
	String	Float64	Float64	Float64
1	New York, NY	8.405	40.7127	-74.0059
2	Los Angeles, CA	3.884	34.0522	-118.244
3	Chicago, IL	2.718	41.8781	-87.6297
4	Houston, TX	2.195	29.7604	-95.3698
5	Philadelphia, PA	1.553	39.9525	-75.1652
6	Phoenix, AZ	1.513	33.4483	-112.074
7	San Antonio, TX	1.409	29.4241	-98.4936
8	San Diego, CA	1.355	32.7157	-117.161
9	Dallas, TX	1.257	32.7766	-96.7969
10	San Jose, CA	0.998	37.3382	-121.886
11	Austin, TX	0.885	30.2671	-97.743
12	Indianapolis, IN	0.843	39.7684	-86.158
13	Jacksonville, FL	0.842	30.3321	-81.6556
14	San Francisco, CA	0.837	37.7749	-122.419
15	Columbus, OH	0.822	39.9611	-82.9987
16	Charlotte, NC	0.792	35.227	-80.8431
17	Fort Worth, TX	0.792	32.7554	-97.3307
18	Detroit, MI	0.688	42.3314	-83.0457
19	El Paso, TX	0.674	31.7775	-106.442
20	Memphis, TN	0.653	35.1495	-90.0489

Model Specifics

We will cluster these 20 cities into 3 different groups and we will assume that the ideal or target population P for a group is simply the total population of the 20 cities divided by 3:

```
n = size(cities, 1)
k = 3
P = sum(cities.population) / k
```

```
11.038333333333334
```

Obtaining the distances between each city

Let's compute the pairwise Haversine distance between each of the cities in our data set and store the result in a variable we'll call `dm`:

```
"""
    haversine(lat1, long1, lat2, long2, r = 6372.8)

Compute the haversine distance between two points on a sphere of radius `r`,
where the points are given by the latitude/longitude pairs `lat1/long1` and
`lat2/long2` (in degrees).
"""
function haversine(lat1, long1, lat2, long2, r = 6372.8)
    lat1, long1 = deg2rad(lat1), deg2rad(long1)
    lat2, long2 = deg2rad(lat2), deg2rad(long2)
    hav(a, b) = sin((b - a) / 2)^2
    inner_term = hav(lat1, lat2) + cos(lat1) * cos(lat2) * hav(long1, long2)
    d = 2 * r * asin(sqrt(inner_term))
    # Round distance to nearest kilometer.
    return round(Int, d)
end
```

```
Main.haversine
```

Our distance matrix is symmetric so we'll convert it to a `LowerTriangular` matrix so that we can better interpret the objective value of our model:

```
dm = LinearAlgebra.LowerTriangular([
    haversine(cities.lat[i], cities.lon[i], cities.lat[j], cities.lon[j])
    for i in 1:n, j in 1:n
])
```

```
20×20 LinearAlgebra.LowerTriangular{Int64, Matrix{Int64}}:
    0     ⋅     ⋅     ⋅     ⋅     ⋅     ⋅  …     ⋅     ⋅     ⋅     ⋅     ⋅  ⋅
 3937     0     ⋅     ⋅     ⋅     ⋅     ⋅        ⋅     ⋅     ⋅     ⋅     ⋅  ⋅
 1145  2805     0     ⋅     ⋅     ⋅     ⋅        ⋅     ⋅     ⋅     ⋅     ⋅  ⋅
 2282  2207  1516     0     ⋅     ⋅     ⋅        ⋅     ⋅     ⋅     ⋅     ⋅  ⋅
  130  3845  1068  2157     0     ⋅     ⋅        ⋅     ⋅     ⋅     ⋅     ⋅  ⋅
 3445   574  2337  1633  3345     0     ⋅  …     ⋅     ⋅     ⋅     ⋅     ⋅  ⋅
 2546  1934  1695   304  2423  1363     0        ⋅     ⋅     ⋅     ⋅     ⋅  ⋅
 3908   179  2787  2094  3812   481  1813        ⋅     ⋅     ⋅     ⋅     ⋅  ⋅
 2206  1993  1295   362  2089  1424   406        ⋅     ⋅     ⋅     ⋅     ⋅  ⋅
```

```
 4103   492  2958  2588  4023   989  2336        ·     ·     ·     ·     ·  ·
 2432  1972  1577   235  2310  1398   118  …     ·     ·     ·     ·     ·  ·
 1036  2907   265  1394   938  2409  1609        ·     ·     ·     ·     ·  ·
 1345  3450  1391  1321  1221  2883  1626        ·     ·     ·     ·     ·  ·
 4130   559  2986  2644  4052  1051  2394        ·     ·     ·     ·     ·  ·
  767  3177   444  1598   668  2679  1834        0     ·     ·     ·     ·  ·
  855  3405   946  1490   725  2863  1777  …   560     0     ·     ·     ·  ·
 2251  1944  1327   382  2134  1375   387     1511  1543     0     ·     ·  ·
  774  3186   382  1780   711  2716  1994      264   813  1646     0     ·  ·
 3054  1130  2010  1081  2945   559   804     2292  2398   864  2374     0  ·
 1534  2576   777   780  1415  2028  1017      820   837   722  1003  1565  0
```

Build the model

Now that we have the basics taken care of, we can set up our model, create decision variables, add constraints, and then solve.

First, we'll set up a model that leverages the Cbc solver. Next, we'll set up a binary variable $x_{i,k}$ that takes the value 1 if city i is in group k and 0 otherwise. Each city must be in a group, so we'll add the constraint $\sum_k x_{i,k} = 1$ for every i.

```
model = Model(GLPK.Optimizer)
set_silent(model)
```

```
@variable(model, x[1:n, 1:k], Bin)
```

```
20×3 Matrix{VariableRef}:
 x[1,1]   x[1,2]   x[1,3]
 x[2,1]   x[2,2]   x[2,3]
 x[3,1]   x[3,2]   x[3,3]
 x[4,1]   x[4,2]   x[4,3]
 x[5,1]   x[5,2]   x[5,3]
 x[6,1]   x[6,2]   x[6,3]
 x[7,1]   x[7,2]   x[7,3]
 x[8,1]   x[8,2]   x[8,3]
 x[9,1]   x[9,2]   x[9,3]
 x[10,1]  x[10,2]  x[10,3]
 x[11,1]  x[11,2]  x[11,3]
 x[12,1]  x[12,2]  x[12,3]
 x[13,1]  x[13,2]  x[13,3]
 x[14,1]  x[14,2]  x[14,3]
 x[15,1]  x[15,2]  x[15,3]
 x[16,1]  x[16,2]  x[16,3]
 x[17,1]  x[17,2]  x[17,3]
 x[18,1]  x[18,2]  x[18,3]
 x[19,1]  x[19,2]  x[19,3]
 x[20,1]  x[20,2]  x[20,3]
```

```
@constraint(model, [i = 1:n], sum(x[i, :]) == 1)
```

```
20-element Vector{ConstraintRef{Model, MathOptInterface.ConstraintIndex{MathOptInterface.
    ScalarAffineFunction{Float64}, MathOptInterface.EqualTo{Float64}}, ScalarShape}}:
 x[1,1] + x[1,2] + x[1,3] = 1.0
```

```
x[2,1] + x[2,2] + x[2,3] = 1.0
x[3,1] + x[3,2] + x[3,3] = 1.0
x[4,1] + x[4,2] + x[4,3] = 1.0
x[5,1] + x[5,2] + x[5,3] = 1.0
x[6,1] + x[6,2] + x[6,3] = 1.0
x[7,1] + x[7,2] + x[7,3] = 1.0
x[8,1] + x[8,2] + x[8,3] = 1.0
x[9,1] + x[9,2] + x[9,3] = 1.0
x[10,1] + x[10,2] + x[10,3] = 1.0
x[11,1] + x[11,2] + x[11,3] = 1.0
x[12,1] + x[12,2] + x[12,3] = 1.0
x[13,1] + x[13,2] + x[13,3] = 1.0
x[14,1] + x[14,2] + x[14,3] = 1.0
x[15,1] + x[15,2] + x[15,3] = 1.0
x[16,1] + x[16,2] + x[16,3] = 1.0
x[17,1] + x[17,2] + x[17,3] = 1.0
x[18,1] + x[18,2] + x[18,3] = 1.0
x[19,1] + x[19,2] + x[19,3] = 1.0
x[20,1] + x[20,2] + x[20,3] = 1.0
```

To reduce symmetry, we fix the first city to belong to the first group.

```
fix(x[1, 1], 1; force = true)
```

The total population of a group k is $Q_k = \sum_i x_{i,k} q_i$ where q_i is simply the ith value from the `population` column in our `cities` DataFrame. Let's add constraints so that $\alpha \leq (Q_k - P) \leq \beta$. We'll set α equal to -3 million and β equal to 3. By adjusting these thresholds you'll find that there is a tradeoff between having relatively even populations between groups and having geographically close cities within each group. In other words, the larger the absolute values of α and β, the closer together the cities in a group will be but the variance between the group populations will be higher.

```
@variable(model, -3 <= population_diff[1:k] <= 3)
@constraint(model, population_diff .== x' * cities.population .- P)
```

```
3-element Vector{ConstraintRef{Model, MathOptInterface.ConstraintIndex{MathOptInterface.
    ScalarAffineFunction{Float64}, MathOptInterface.EqualTo{Float64}}, ScalarShape}}:
 -8.405 x[1,1] - 3.884 x[2,1] - 2.718 x[3,1] - 2.195 x[4,1] - 1.553 x[5,1] - 1.513 x[6,1] - 1.409 x
    [7,1] - 1.355 x[8,1] - 1.257 x[9,1] - 0.998 x[10,1] - 0.885 x[11,1] - 0.843 x[12,1] - 0.842 x
    [13,1] - 0.837 x[14,1] - 0.822 x[15,1] - 0.792 x[16,1] - 0.792 x[17,1] - 0.688 x[18,1] - 0.674 x
    [19,1] - 0.653 x[20,1] + population_diff[1] = -11.038333333333334
 -8.405 x[1,2] - 3.884 x[2,2] - 2.718 x[3,2] - 2.195 x[4,2] - 1.553 x[5,2] - 1.513 x[6,2] - 1.409 x
    [7,2] - 1.355 x[8,2] - 1.257 x[9,2] - 0.998 x[10,2] - 0.885 x[11,2] - 0.843 x[12,2] - 0.842 x
    [13,2] - 0.837 x[14,2] - 0.822 x[15,2] - 0.792 x[16,2] - 0.792 x[17,2] - 0.688 x[18,2] - 0.674 x
    [19,2] - 0.653 x[20,2] + population_diff[2] = -11.038333333333334
 -8.405 x[1,3] - 3.884 x[2,3] - 2.718 x[3,3] - 2.195 x[4,3] - 1.553 x[5,3] - 1.513 x[6,3] - 1.409 x
    [7,3] - 1.355 x[8,3] - 1.257 x[9,3] - 0.998 x[10,3] - 0.885 x[11,3] - 0.843 x[12,3] - 0.842 x
    [13,3] - 0.837 x[14,3] - 0.822 x[15,3] - 0.792 x[16,3] - 0.792 x[17,3] - 0.688 x[18,3] - 0.674 x
    [19,3] - 0.653 x[20,3] + population_diff[3] = -11.038333333333334
```

Now we need to add one last binary variable $z_{i,j}$ to our model that we'll use to compute the total distance between the cities in our groups, defined as $\sum_{i,j} d_{i,j} z_{i,j}$. Variable $z_{i,j}$ will equal 1 if cities i and j are in the same group, and 0 if they are not in the same group.

To ensure that $z_{i,j} = 1$ if and only if cities i and j are in the same group, we add the constraints $z_{i,j} \geq x_{i,k} + x_{j,k} - 1$ for every pair i, j and every k:

```
@variable(model, z[i = 1:n, j = 1:i], Bin)
```

```
JuMP.Containers.SparseAxisArray{VariableRef, 2, Tuple{Int64, Int64}} with 210 entries:
  [12, 10]  =  z[12,10]
  [12, 2 ]  =  z[12,2]
  [12, 3 ]  =  z[12,3]
  [16, 12]  =  z[16,12]
  [16, 14]  =  z[16,14]
  [16, 16]  =  z[16,16]
  [17, 12]  =  z[17,12]
  [18, 14]  =  z[18,14]

  [18, 16]  =  z[18,16]
  [18, 18]  =  z[18,18]
  [19, 12]  =  z[19,12]
  [19, 14]  =  z[19,14]
  [19, 16]  =  z[19,16]
  [19, 19]  =  z[19,19]
  [20, 15]  =  z[20,15]
```

```
for k in 1:k, i in 1:n, j in 1:i
    @constraint(model, z[i, j] >= x[i, k] + x[j, k] - 1)
end
```

We can now add an objective to our model which will simply be to minimize the dot product of z and our distance matrix, `dm`.

```
@objective(model, Min, sum(dm[i, j] * z[i, j] for i in 1:n, j in 1:i))
```

$$3937z_{2,1}+1145z_{3,1}+2805z_{3,2}+2282z_{4,1}+2207z_{4,2}+1516z_{4,3}+130z_{5,1}+3845z_{5,2}+1068z_{5,3}+2157z_{5,4}+3445z_{6,1}+574z_{6,2}+$$

We can then call `optimize!` and review the results.

```
optimize!(model)
```

Reviewing the Results

Now that we have results, we can add a column to our `cities` DataFrame for the group and then loop through our x variable to assign each city to its group. Once we have that, we can look at the total population for each group and also look at the cities in each group to verify that they are grouped by geographic proximity.

```
cities.group = zeros(n)

for i in 1:n, j in 1:k
    if round(Int, value(x[i, j])) == 1
        cities.group[i] = j
    end
end

for group in DataFrames.groupby(cities, :group)
```

```
    @show group
    println("")
    @show sum(group.population)
    println("")
end
```

```
group = 7×5 SubDataFrame
 Row │ city              population  lat      lon       group
     │ String            Float64     Float64  Float64   Float64
─────┼──────────────────────────────────────────────────────────
   1 │ New York, NY           8.405  40.7127  -74.0059      1.0
   2 │ Philadelphia, PA       1.553  39.9525  -75.1652      1.0
   3 │ Indianapolis, IN       0.843  39.7684  -86.158       1.0
   4 │ Jacksonville, FL       0.842  30.3321  -81.6556      1.0
   5 │ Columbus, OH           0.822  39.9611  -82.9987      1.0
   6 │ Charlotte, NC          0.792  35.227   -80.8431      1.0
   7 │ Detroit, MI            0.688  42.3314  -83.0457      1.0

sum(group.population) = 13.944999999999999

group = 6×5 SubDataFrame
 Row │ city              population  lat      lon       group
     │ String            Float64     Float64  Float64   Float64
─────┼──────────────────────────────────────────────────────────
   1 │ Los Angeles, CA         3.884  34.0522  -118.244      2.0
   2 │ Phoenix, AZ             1.513  33.4483  -112.074      2.0
   3 │ San Diego, CA           1.355  32.7157  -117.161      2.0
   4 │ San Jose, CA            0.998  37.3382  -121.886      2.0
   5 │ San Francisco, CA       0.837  37.7749  -122.419      2.0
   6 │ El Paso, TX             0.674  31.7775  -106.442      2.0

sum(group.population) = 9.261000000000001

group = 7×5 SubDataFrame
 Row │ city              population  lat      lon       group
     │ String            Float64     Float64  Float64   Float64
─────┼──────────────────────────────────────────────────────────
   1 │ Chicago, IL           2.718  41.8781  -87.6297      3.0
   2 │ Houston, TX           2.195  29.7604  -95.3698      3.0
   3 │ San Antonio, TX       1.409  29.4241  -98.4936      3.0
   4 │ Dallas, TX            1.257  32.7766  -96.7969      3.0
   5 │ Austin, TX            0.885  30.2671  -97.743       3.0
   6 │ Fort Worth, TX        0.792  32.7554  -97.3307      3.0
   7 │ Memphis, TN           0.653  35.1495  -90.0489      3.0

sum(group.population) = 9.909
```

Tip

This tutorial was generated using Literate.jl. View the source `.jl` file on GitHub.

4.8 The knapsack problem

Formulate and solve a simple knapsack problem:

```
max sum(p_j x_j)
 st sum(w_j x_j) <= C
    x binary
```

```
using JuMP
import GLPK
import Test

function example_knapsack(; verbose = true)
    profit = [5, 3, 2, 7, 4]
    weight = [2, 8, 4, 2, 5]
    capacity = 10
    model = Model(GLPK.Optimizer)
    @variable(model, x[1:5], Bin)
    # Objective: maximize profit
    @objective(model, Max, profit' * x)
    # Constraint: can carry all
    @constraint(model, weight' * x <= capacity)
    # Solve problem using MIP solver
    optimize!(model)
    if verbose
        println("Objective is: ", objective_value(model))
        println("Solution is:")
        for i in 1:5
            print("x[$i] = ", value(x[i]))
            println(", p[$i]/w[$i] = ", profit[i] / weight[i])
        end
    end
    Test.@test termination_status(model) == OPTIMAL
    Test.@test primal_status(model) == FEASIBLE_POINT
    Test.@test objective_value(model) == 16.0
    return
end

example_knapsack()
```

```
Objective is: 16.0
Solution is:
x[1] = 1.0, p[1]/w[1] = 2.5
x[2] = 0.0, p[2]/w[2] = 0.375
x[3] = 0.0, p[3]/w[3] = 0.5
x[4] = 1.0, p[4]/w[4] = 3.5
x[5] = 1.0, p[5]/w[5] = 0.8
```

Tip

This tutorial was generated using Literate.jl. View the source `.jl` file on GitHub.

4.9 The multi-commodity flow problem

JuMP implementation of the multicommodity transportation model AMPL: A Modeling Language for Mathematical Programming, 2nd ed by Robert Fourer, David Gay, and Brian W. Kernighan 4-1.

Originally contributed by Louis Luangkesorn, February 26, 2015.

```
using JuMP
import GLPK
import Test

function example_multi(; verbose = true)
    orig = ["GARY", "CLEV", "PITT"]
    dest = ["FRA", "DET", "LAN", "WIN", "STL", "FRE", "LAF"]
    prod = ["bands", "coils", "plate"]
    numorig = length(orig)
    numdest = length(dest)
    numprod = length(prod)
    # supply(prod, orig) amounts available at origins
    supply = [
        400 700 800
        800 1600 1800
        200 300 300
    ]
    # demand(prod, dest) amounts required at destinations
    demand = [
        300 300 100 75 650 225 250
        500 750 400 250 950 850 500
        100 100 0 50 200 100 250
    ]
    # limit(orig, dest) of total units from any origin to destination
    limit = [625.0 for j in 1:numorig, i in 1:numdest]
    # cost(dest, orig, prod) Shipment cost per unit
    cost = reshape(
        [
            [
                [30, 10, 8, 10, 11, 71, 6]
                [22, 7, 10, 7, 21, 82, 13]
                [19, 11, 12, 10, 25, 83, 15]
            ]
            [
                [39, 14, 11, 14, 16, 82, 8]
                [27, 9, 12, 9, 26, 95, 17]
                [24, 14, 17, 13, 28, 99, 20]
            ]
            [
                [41, 15, 12, 16, 17, 86, 8]
                [29, 9, 13, 9, 28, 99, 18]
                [26, 14, 17, 13, 31, 104, 20]
            ]
        ],
        7,
        3,
        3,
    )
    # DECLARE MODEL
```

```
    multi = Model(GLPK.Optimizer)
    # VARIABLES
    @variable(multi, trans[1:numorig, 1:numdest, 1:numprod] >= 0)
    # OBJECTIVE
    @objective(
        multi,
        Max,
        sum(
            cost[j, i, p] * trans[i, j, p] for i in 1:numorig, j in 1:numdest,
            p in 1:numprod
        )
    )
    # CONSTRAINTS
    # Supply constraint
    @constraint(
        multi,
        supply_con[i in 1:numorig, p in 1:numprod],
        sum(trans[i, j, p] for j in 1:numdest) == supply[p, i]
    )
    # Demand constraint
    @constraint(
        multi,
        demand_con[j in 1:numdest, p in 1:numprod],
        sum(trans[i, j, p] for i in 1:numorig) == demand[p, j]
    )
    # Total shipment constraint
    @constraint(
        multi,
        total_con[i in 1:numorig, j in 1:numdest],
        sum(trans[i, j, p] for p in 1:numprod) - limit[i, j] <= 0
    )
    optimize!(multi)
    Test.@test termination_status(multi) == OPTIMAL
    Test.@test primal_status(multi) == FEASIBLE_POINT
    Test.@test objective_value(multi) == 225_700.0
    if verbose
        println("RESULTS:")
        for i in 1:length(orig)
            for j in 1:length(dest)
                for p in 1:length(prod)
                    print(
                        " $(prod[p]) $(orig[i]) $(dest[j]) = $(value(trans[i, j, p]))\t",
                    )
                end
                println()
            end
        end
    end
    return
end

example_multi()
```

```
RESULTS:
 bands GARY FRA = 25.0   coils GARY FRA = 500.0  plate GARY FRA = 100.0
 bands GARY DET = 125.0  coils GARY DET = 0.0    plate GARY DET = 50.0
```

```
bands GARY LAN = 0.0    coils GARY LAN = 0.0    plate GARY LAN = 0.0
bands GARY WIN = 0.0    coils GARY WIN = 0.0    plate GARY WIN = 50.0
bands GARY STL = 250.0  coils GARY STL = 300.0  plate GARY STL = 0.0
bands GARY FRE = 0.0    coils GARY FRE = 0.0    plate GARY FRE = 0.0
bands GARY LAF = 0.0    coils GARY LAF = 0.0    plate GARY LAF = 0.0
bands CLEV FRA = 275.0  coils CLEV FRA = 0.0    plate CLEV FRA = 0.0
bands CLEV DET = 100.0  coils CLEV DET = 200.0  plate CLEV DET = 50.0
bands CLEV LAN = 100.0  coils CLEV LAN = 0.0    plate CLEV LAN = 0.0
bands CLEV WIN = 0.0    coils CLEV WIN = 75.0   plate CLEV WIN = 0.0
bands CLEV STL = 0.0    coils CLEV STL = 625.0  plate CLEV STL = 0.0
bands CLEV FRE = 225.0  coils CLEV FRE = 325.0  plate CLEV FRE = 0.0
bands CLEV LAF = 0.0    coils CLEV LAF = 375.0  plate CLEV LAF = 250.0
bands PITT FRA = 0.0    coils PITT FRA = 0.0    plate PITT FRA = 0.0
bands PITT DET = 75.0   coils PITT DET = 550.0  plate PITT DET = 0.0
bands PITT LAN = 0.0    coils PITT LAN = 400.0  plate PITT LAN = 0.0
bands PITT WIN = 75.0   coils PITT WIN = 175.0  plate PITT WIN = 0.0
bands PITT STL = 400.0  coils PITT STL = 25.0   plate PITT STL = 200.0
bands PITT FRE = 0.0    coils PITT FRE = 525.0  plate PITT FRE = 100.0
bands PITT LAF = 250.0  coils PITT LAF = 125.0  plate PITT LAF = 0.0
```

Tip

This tutorial was generated using Literate.jl. View the source `.jl` file on GitHub.

4.10 N-Queens

Originally Contributed by: Matthew Helm (with help from @mtanneau on Julia Discourse)

The N-Queens problem involves placing N queens on an N x N chessboard such that none of the queens attacks another. In chess, a queen can move vertically, horizontally, and diagonally so there cannot be more than one queen on any given row, column, or diagonal.

Note that none of the queens above are able to attack any other as a result of their careful placement.

```
using JuMP
import GLPK
import LinearAlgebra
```

N-Queens

```
N = 8

model = Model(GLPK.Optimizer)
```

```
A JuMP Model
Feasibility problem with:
Variables: 0
Model mode: AUTOMATIC
CachingOptimizer state: EMPTY_OPTIMIZER
Solver name: GLPK
```

Next, let's create an N x N chessboard of binary values. 0 will represent an empty space on the board and 1 will represent a space occupied by one of our queens:

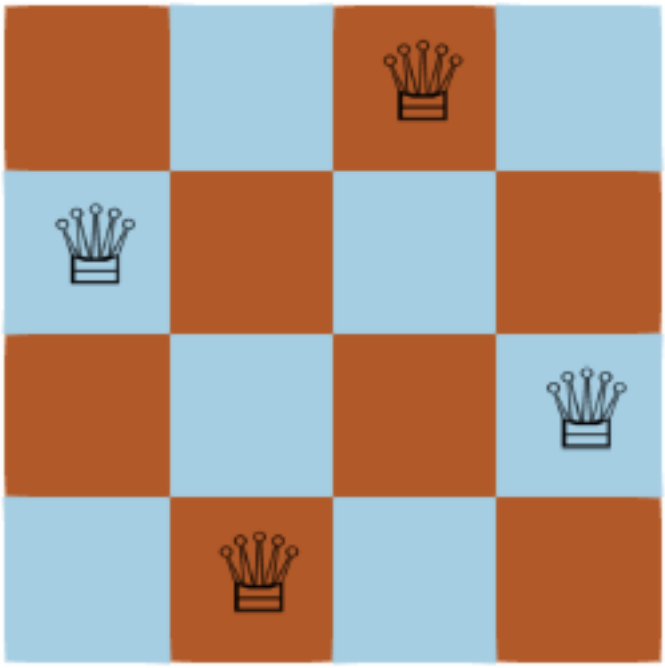


Figure 4.1: Four Queens

```
@variable(model, x[1:N, 1:N], Bin)
```

```
8×8 Matrix{VariableRef}:
 x[1,1]  x[1,2]  x[1,3]  x[1,4]  x[1,5]  x[1,6]  x[1,7]  x[1,8]
 x[2,1]  x[2,2]  x[2,3]  x[2,4]  x[2,5]  x[2,6]  x[2,7]  x[2,8]
 x[3,1]  x[3,2]  x[3,3]  x[3,4]  x[3,5]  x[3,6]  x[3,7]  x[3,8]
 x[4,1]  x[4,2]  x[4,3]  x[4,4]  x[4,5]  x[4,6]  x[4,7]  x[4,8]
 x[5,1]  x[5,2]  x[5,3]  x[5,4]  x[5,5]  x[5,6]  x[5,7]  x[5,8]
 x[6,1]  x[6,2]  x[6,3]  x[6,4]  x[6,5]  x[6,6]  x[6,7]  x[6,8]
 x[7,1]  x[7,2]  x[7,3]  x[7,4]  x[7,5]  x[7,6]  x[7,7]  x[7,8]
 x[8,1]  x[8,2]  x[8,3]  x[8,4]  x[8,5]  x[8,6]  x[8,7]  x[8,8]
```

Now we can add our constraints:

There must be exactly one queen in a given row/column

```
for i in 1:N
    @constraint(model, sum(x[i, :]) == 1)
    @constraint(model, sum(x[:, i]) == 1)
end
```

There can only be one queen on any given diagonal

```
for i in -(N - 1):(N-1)
    @constraint(model, sum(LinearAlgebra.diag(x, i)) <= 1)
    @constraint(model, sum(LinearAlgebra.diag(reverse(x, dims = 1), i)) <= 1)
end
```

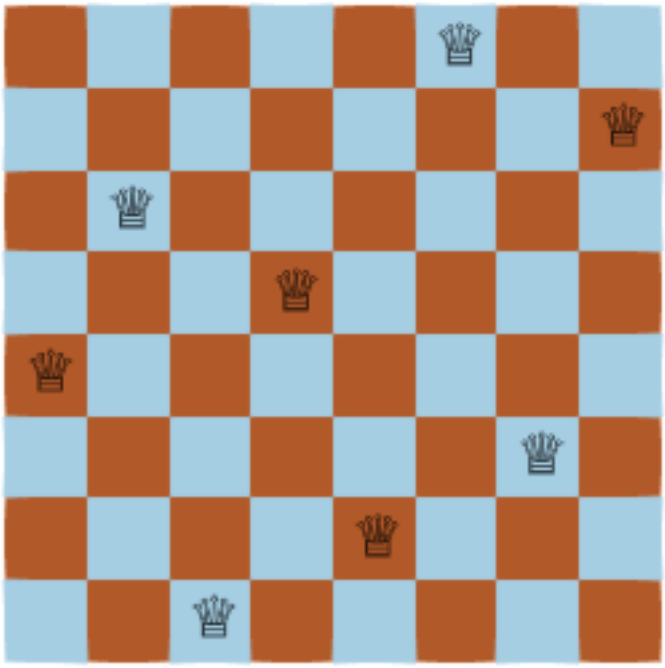


Figure 4.2: Four Queens

That's it! We are ready to put our model to work and see if it is able to find a feasible solution:

```
optimize!(model)
```

We can now review the solution that our model found:

```
solution = convert.(Int, value.(x))
```

```
8×8 Matrix{Int64}:
 0  0  0  0  0  1  0  0
 0  0  0  0  0  0  0  1
 0  1  0  0  0  0  0  0
 0  0  0  1  0  0  0  0
 1  0  0  0  0  0  0  0
 0  0  0  0  0  0  1  0
 0  0  0  0  1  0  0  0
 0  0  1  0  0  0  0  0
```

Tip

This tutorial was generated using Literate.jl. View the source `.jl` file on GitHub.

4.11 Network Flows

Originally Contributed by: Arpit Bhatia

In graph theory, a flow network (also known as a transportation network) is a directed graph where each edge has a capacity and each edge receives a flow. The amount of flow on an edge cannot exceed the capacity of the edge.

Often in operations research, a directed graph is called a network, the vertices are called nodes and the edges are called arcs.

A flow must satisfy the restriction that the amount of flow into a node equals the amount of flow out of it, unless it is a source, which has only outgoing flow, or sink, which has only incoming flow.

A network can be used to model traffic in a computer network, circulation with demands, fluids in pipes, currents in an electrical circuit, or anything similar in which something travels through a network of nodes.

```
using JuMP
import GLPK
import LinearAlgebra
```

The Shortest Path Problem

Suppose that each arc (i,j) of a graph is assigned a scalar cost $a_{i,j}$, and suppose that we define the cost of a forward path to be the sum of the costs of its arcs.

Given a pair of nodes, the shortest path problem is to find a forward path that connects these nodes and has minimum cost.

$$\min \sum_{\forall e(i,j) \in E} a_{i,j} \times x_{i,j}$$

$$s.t. \quad b(i) = \sum_j x_{ij} - \sum_k x_{ki} = \begin{cases} 1 & \text{if } i \text{ is the starting node,} \\ -1 & \text{if } i \text{ is the ending node,} \\ 0 & \text{otherwise.} \end{cases}$$

$$x_e \in \{0,1\} \quad \forall e \in E$$

```
G = [
    0 100 30 0 0
    0 0 20 0 0
    0 0 0 10 60
    0 15 0 0 50
    0 0 0 0 0
]

n = size(G)[1]

shortest_path = Model(GLPK.Optimizer)

@variable(shortest_path, x[1:n, 1:n], Bin)
```

```
5×5 Matrix{VariableRef}:
 x[1,1]  x[1,2]  x[1,3]  x[1,4]  x[1,5]
 x[2,1]  x[2,2]  x[2,3]  x[2,4]  x[2,5]
```

```
 x[3,1]  x[3,2]  x[3,3]  x[3,4]  x[3,5]
 x[4,1]  x[4,2]  x[4,3]  x[4,4]  x[4,5]
 x[5,1]  x[5,2]  x[5,3]  x[5,4]  x[5,5]
```

Arcs with zero cost are not a part of the path as they do no exist

```
@constraint(shortest_path, [i = 1:n, j = 1:n; G[i, j] == 0], x[i, j] == 0)
```

```
JuMP.Containers.SparseAxisArray{ConstraintRef{Model, MathOptInterface.ConstraintIndex{
    MathOptInterface.ScalarAffineFunction{Float64}, MathOptInterface.EqualTo{Float64}}, ScalarShape
    }, 2, Tuple{Int64, Int64}} with 18 entries:
  [1, 1]  =  x[1,1] = 0.0
  [1, 4]  =  x[1,4] = 0.0
  [2, 2]  =  x[2,2] = 0.0
  [2, 4]  =  x[2,4] = 0.0
  [2, 5]  =  x[2,5] = 0.0
  [3, 1]  =  x[3,1] = 0.0
  [3, 2]  =  x[3,2] = 0.0
  [3, 3]  =  x[3,3] = 0.0

  [4, 1]  =  x[4,1] = 0.0
  [4, 3]  =  x[4,3] = 0.0
  [4, 4]  =  x[4,4] = 0.0
  [5, 2]  =  x[5,2] = 0.0
  [5, 3]  =  x[5,3] = 0.0
  [5, 4]  =  x[5,4] = 0.0
  [5, 5]  =  x[5,5] = 0.0
```

Flow conservation constraint

```
@constraint(
    shortest_path,
    [i = 1:n; i != 1 && i != 2],
    sum(x[i, :]) == sum(x[:, i])
)
```

```
JuMP.Containers.SparseAxisArray{ConstraintRef{Model, MathOptInterface.ConstraintIndex{
    MathOptInterface.ScalarAffineFunction{Float64}, MathOptInterface.EqualTo{Float64}}, ScalarShape
    }, 1, Tuple{Int64}} with 3 entries:
  [3]  =  x[3,1] + x[3,2] - x[1,3] - x[2,3] - x[4,3] - x[5,3] + x[3,4] + x[3,5] = 0.0
  [4]  =  x[4,1] + x[4,2] + x[4,3] - x[1,4] - x[2,4] - x[3,4] - x[5,4] + x[4,5] = 0.0
  [5]  =  x[5,1] + x[5,2] + x[5,3] + x[5,4] - x[1,5] - x[2,5] - x[3,5] - x[4,5] = 0.0
```

Flow coming out of source = 1

```
@constraint(shortest_path, sum(x[1, :]) - sum(x[:, 1]) == 1)
```

$$-x_{2,1} - x_{3,1} - x_{4,1} - x_{5,1} + x_{1,2} + x_{1,3} + x_{1,4} + x_{1,5} = 1.0$$

Flowing coming out of destination = -1 i.e. Flow entering destination = 1

```
@constraint(shortest_path, sum(x[2, :]) - sum(x[:, 2]) == -1)
@objective(shortest_path, Min, LinearAlgebra.dot(G, x))

optimize!(shortest_path)
objective_value(shortest_path)
```

```
55.0
```

```
value.(x)
```

```
5×5 Matrix{Float64}:
 0.0  0.0  1.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  1.0  0.0
 0.0  1.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0
```

The Assignment Problem

Suppose that there are n persons and n objects that we have to match on a one-to-one basis. There is a benefit or value $a_{i,j}$ for matching person i with object j, and we want to assign persons to objects so as to maximize the total benefit.

There is also a restriction that person i can be assigned to object j only if (i, j) belongs to a given set of pairs A.

Mathematically, we want to find a set of person-object pairs $(1, j_1), ..., (n, j_n)$ from A such that the objects $j_1, ..., j_n$ are all distinct, and the total benefit $\sum_{i=1}^{y} a_{ij_i}$ is maximized.

$$
\begin{aligned}
\max \quad & \sum_{(i,j)\in A} a_{i,j} \times y_{i,j} \\
s.t. \quad & \sum_{\{j|(i,j)\in A\}} y_{i,j} = 1 \qquad \forall i = \{1, 2....n\} \\
& \sum_{\{i|(i,j)\in A\}} y_{i,j} = 1 \qquad \forall j = \{1, 2....n\} \\
& y_{i,j} \in \{0, 1\} \quad \forall (i, j) \in \{1, 2...k\}
\end{aligned}
$$

```
G = [
    6 4 5 0
    0 3 6 0
    5 0 4 3
    7 5 5 5
]

n = size(G)[1]

assignment = Model(GLPK.Optimizer)
@variable(assignment, y[1:n, 1:n], Bin)
```

```
4×4 Matrix{VariableRef}:
 y[1,1]  y[1,2]  y[1,3]  y[1,4]
 y[2,1]  y[2,2]  y[2,3]  y[2,4]
 y[3,1]  y[3,2]  y[3,3]  y[3,4]
 y[4,1]  y[4,2]  y[4,3]  y[4,4]
```

One person can only be assigned to one object

```
@constraint(assignment, [i = 1:n], sum(y[:, i]) == 1)
```

```
4-element Vector{ConstraintRef{Model, MathOptInterface.ConstraintIndex{MathOptInterface.
    ScalarAffineFunction{Float64}, MathOptInterface.EqualTo{Float64}}, ScalarShape}}:
 y[1,1] + y[2,1] + y[3,1] + y[4,1] = 1.0
 y[1,2] + y[2,2] + y[3,2] + y[4,2] = 1.0
 y[1,3] + y[2,3] + y[3,3] + y[4,3] = 1.0
 y[1,4] + y[2,4] + y[3,4] + y[4,4] = 1.0
```

One object can only be assigned to one person

```
@constraint(assignment, [j = 1:n], sum(y[j, :]) == 1)
@objective(assignment, Max, LinearAlgebra.dot(G, y))

optimize!(assignment)
objective_value(assignment)
```

```
20.0
```

```
value.(y)
```

```
4×4 Matrix{Float64}:
 0.0  1.0  0.0  0.0
 0.0  0.0  1.0  0.0
 1.0  0.0  0.0  0.0
 0.0  0.0  0.0  1.0
```

The Max-Flow Problem

In the max-flow problem, we have a graph with two special nodes: the $source$, denoted by s, and the $sink$, denoted by t.

The objective is to move as much flow as possible from s into t while observing the capacity constraints.

$$
\begin{aligned}
\max \quad & \sum_{v:(s,v)\in E} f(s,v) \\
s.t. \quad & \sum_{u:(u,v)\in E} f(u,v) = \sum_{w:(v,w)\in E} f(v,w) && \forall v \in V - \{s,t\} \\
& f(u,v) \le c(u,v) && \forall (u,v) \in E \\
& f(u,v) \ge 0 && \forall (u,v) \in E
\end{aligned}
$$

```
G = [
    0 3 2 2 0 0 0 0
    0 0 0 0 5 1 0 0
    0 0 0 0 1 3 1 0
    0 0 0 0 0 1 0 0
    0 0 0 0 0 0 0 4
    0 0 0 0 0 0 0 2
    0 0 0 0 0 0 0 4
    0 0 0 0 0 0 0 0
]

n = size(G)[1]

max_flow = Model(GLPK.Optimizer)

@variable(max_flow, f[1:n, 1:n] >= 0)
```

```
8×8 Matrix{VariableRef}:
 f[1,1]  f[1,2]  f[1,3]  f[1,4]  f[1,5]  f[1,6]  f[1,7]  f[1,8]
 f[2,1]  f[2,2]  f[2,3]  f[2,4]  f[2,5]  f[2,6]  f[2,7]  f[2,8]
 f[3,1]  f[3,2]  f[3,3]  f[3,4]  f[3,5]  f[3,6]  f[3,7]  f[3,8]
 f[4,1]  f[4,2]  f[4,3]  f[4,4]  f[4,5]  f[4,6]  f[4,7]  f[4,8]
 f[5,1]  f[5,2]  f[5,3]  f[5,4]  f[5,5]  f[5,6]  f[5,7]  f[5,8]
 f[6,1]  f[6,2]  f[6,3]  f[6,4]  f[6,5]  f[6,6]  f[6,7]  f[6,8]
 f[7,1]  f[7,2]  f[7,3]  f[7,4]  f[7,5]  f[7,6]  f[7,7]  f[7,8]
 f[8,1]  f[8,2]  f[8,3]  f[8,4]  f[8,5]  f[8,6]  f[8,7]  f[8,8]
```

Capacity constraints

```
@constraint(max_flow, [i = 1:n, j = 1:n], f[i, j] <= G[i, j])
```

```
8×8 Matrix{ConstraintRef{Model, MathOptInterface.ConstraintIndex{MathOptInterface.
    ScalarAffineFunction{Float64}, MathOptInterface.LessThan{Float64}}, ScalarShape}}:
 f[1,1] ≤ 0.0  f[1,2] ≤ 3.0  f[1,3] ≤ 2.0  …  f[1,7] ≤ 0.0  f[1,8] ≤ 0.0
 f[2,1] ≤ 0.0  f[2,2] ≤ 0.0  f[2,3] ≤ 0.0     f[2,7] ≤ 0.0  f[2,8] ≤ 0.0
 f[3,1] ≤ 0.0  f[3,2] ≤ 0.0  f[3,3] ≤ 0.0     f[3,7] ≤ 1.0  f[3,8] ≤ 0.0
 f[4,1] ≤ 0.0  f[4,2] ≤ 0.0  f[4,3] ≤ 0.0     f[4,7] ≤ 0.0  f[4,8] ≤ 0.0
 f[5,1] ≤ 0.0  f[5,2] ≤ 0.0  f[5,3] ≤ 0.0     f[5,7] ≤ 0.0  f[5,8] ≤ 4.0
 f[6,1] ≤ 0.0  f[6,2] ≤ 0.0  f[6,3] ≤ 0.0  …  f[6,7] ≤ 0.0  f[6,8] ≤ 2.0
 f[7,1] ≤ 0.0  f[7,2] ≤ 0.0  f[7,3] ≤ 0.0     f[7,7] ≤ 0.0  f[7,8] ≤ 4.0
 f[8,1] ≤ 0.0  f[8,2] ≤ 0.0  f[8,3] ≤ 0.0     f[8,7] ≤ 0.0  f[8,8] ≤ 0.0
```

Flow conservation constraints

```
@constraint(max_flow, [i = 1:n; i != 1 && i != 8], sum(f[i, :]) == sum(f[:, i]))
@objective(max_flow, Max, sum(f[1, :]))

optimize!(max_flow)
objective_value(max_flow)
```

```
6.0
```

```
value.(f)
```

```
8×8 Matrix{Float64}:
 0.0  3.0  2.0  1.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  3.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  1.0  1.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  1.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  4.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  2.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
```

Tip

This tutorial was generated using Literate.jl. View the source `.jl` file on GitHub.

4.12 The workforce scheduling problem

This model determines a set of workforce levels that will most economically meet demands and inventory requirements over time. The formulation is motivated by the experiences of a large producer in the United States. The data are for three products and 13 periods.

Problem taken from the Appendix C of the expanded version of Fourer, Gay, and Kernighan, A Modeling Language for Mathematical Programming

Originally contributed by Louis Luangkesorn, February 26, 2015.

```
using JuMP
import GLPK
import Test

function example_prod(; verbose = true)
    # PRODUCTION SETS AND PARAMETERS
    prd = ["18REG" "24REG" "24PRO"]
    # Members of the product group
    numprd = length(prd)
    pt = [1.194, 1.509, 1.509]
    # Crew-hours to produce 1000 units
    pc = [2304, 2920, 2910]
    # Nominal production cost per 1000, used
    # to compute inventory and shortage costs
    #
    # TIME PERIOD SETS AND PARAMETERS
    firstperiod = 1
    # Index of first production period to be modeled
    lastperiod = 13
    # Index of last production period to be modeled
    numperiods = firstperiod:lastperiod
    # 'planning horizon' := first..last;
    # EMPLOYMENT PARAMETERS
    # Workers per crew
    cs = 18
    # Regular-time hours per shift
    sl = 8
    # Wage per hour for regular-time labor
    rtr = 16.00
```

```
    # Wage per hour for overtime labor
    otr = 43.85
    # Crews employed at start of first period
    iw = 8
    # Regular working days in a production period
    dpp = [19.5, 19, 20, 19, 19.5, 19, 19, 20, 19, 20, 20, 18, 18]
    # Maximum crew-hours of overtime in a period
    ol = [96, 96, 96, 96, 96, 96, 96, 96, 96, 96, 96, 96, 96]
    # Lower limit on average employment in a period
    cmin = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
    # Upper limit on average employment in a period
    cmax = [8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8]
    # Penalty cost of hiring a crew
    hc = [
        7500,
        7500,
        7500,
        7500,
        15000,
        15000,
        15000,
        15000,
        15000,
        15000,
        7500,
        7500,
        7500,
    ]
    # Penalty cost of laying off a crew
    lc = [
        7500,
        7500,
        7500,
        7500,
        15000,
        15000,
        15000,
        15000,
        15000,
        15000,
        7500,
        7500,
        7500,
    ]
    # DEMAND PARAMETERS
    d18REG = [
        63.8,
        76,
        88.4,
        913.8,
        115,
        133.8,
        79.6,
        111,
        121.6,
        470,
```

```
        78.4,
        99.4,
        140.4,
        63.8,
    ]
    d24REG = [
        1212,
        306.2,
        319,
        208.4,
        298,
        328.2,
        959.6,
        257.6,
        335.6,
        118,
        284.8,
        970,
        343.8,
        1212,
    ]
    d24PRO = [0, 0, 0, 0, 0, 0, 0, 0, 0, 1102, 0, 0, 0, 0]
    # Requirements (in 1000s) to be met from current production and inventory
    dem = Array[d18REG, d24REG, d24PRO]
    # true if product will be the subject of a special promotion in the period
    pro = Array[
        [0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0],
        [1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 1],
        [0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0],
    ]
    # INVENTORY AND SHORTAGE PARAMETERS
    # Proportion of non-promoted demand that must be in inventory the previous
    # period
    rir = 0.75
    # Proportion of promoted demand that must be in inventory the previous
    # period
    pir = 0.80
    # Upper limit on number of periods that any product may sit in inventory
    life = 2
    # Inventory cost per 1000 units is cri times nominal production cost
    cri = [0.015, 0.015, 0.015]
    # Shortage cost per 1000 units is crs times nominal production cost
    crs = [1.1, 1.1, 1.1]
    # Inventory at start of first period; age unknown
    iinv = [82, 792.2, 0]
    # Initial inventory still available for allocation at end of period t
    iil = [
        [
            max(0, iinv[p] - sum(dem[p][v] for v in firstperiod:t)) for
            t in numperiods
        ] for p in 1:numprd
    ]
    # Lower limit on inventory at end of period t
    function checkpro(
        product,
        timeperiod,
```

```
        production,
        promotionalrate,
        regularrate,
    )
        if production[product][timeperiod+1] == 1
            return promotionalrate
        else
            return regularrate
        end
    end
    minv = [
        [dem[p][t+1] * checkpro(p, t, pro, pir, rir) for t in numperiods]
        for p in 1:numprd
    ]
    # DEFINE MODEL
    prod = Model(GLPK.Optimizer)
    # VARIABLES
    # Average number of crews employed in each period
    @variable(prod, Crews[0:lastperiod] >= 0)
    # Crews hired from previous to current period
    @variable(prod, Hire[numperiods] >= 0)
    # Crews laid off from previous to current period
    @variable(prod, Layoff[numperiods] >= 0)
    # Production using regular-time labor, in 1000s
    @variable(prod, Rprd[1:numprd, numperiods] >= 0)
    # Production using overtime labor, in 1000s
    @variable(prod, Oprd[1:numprd, numperiods] >= 0)
    # a numperiods old -- produced in period (t+1)-a --
    # and still in storage at the end of period t
    @variable(prod, Inv[1:numprd, numperiods, 1:life] >= 0)
    # Accumulated unsatisfied demand at the end of period t
    @variable(prod, Short[1:numprd, numperiods] >= 0)
    # CONSTRAINTS
    # Hours needed to accomplish all regular-time production in a period must
    # not exceed hours available on all shifts
    @constraint(
        prod,
        [t = numperiods],
        sum(pt[p] * Rprd[p, t] for p in 1:numprd) <= sl * dpp[t] * Crews[t]
    )
    # Hours needed to accomplish all overtime production in a period must not
    # exceed the specified overtime limit
    @constraint(
        prod,
        [t = numperiods],
        sum(pt[p] * Oprd[p, t] for p in 1:numprd) <= ol[t]
    )
    # Use given initial workforce
    @constraint(prod, Crews[firstperiod-1] == iw)
    # Workforce changes by hiring or layoffs
    @constraint(
        prod,
        [t in numperiods],
        Crews[t] == Crews[t-1] + Hire[t] - Layoff[t]
    )
    # Workforce must remain within specified bounds
```

```
    @constraint(prod, [t in numperiods], cmin[t] <= Crews[t])
    @constraint(prod, [t in numperiods], Crews[t] <= cmax[t])
    # 'first demand requirement
    @constraint(
        prod,
        [p in 1:numprd],
        Rprd[p, firstperiod] + Oprd[p, firstperiod] + Short[p, firstperiod] -
        Inv[p, firstperiod, 1] == max(0, dem[p][firstperiod] - iinv[p])
    )
    # Production plus increase in shortage plus decrease in inventory must
    # equal demand
    for t in (firstperiod+1):lastperiod
        @constraint(
            prod,
            [p in 1:numprd],
            Rprd[p, t] + Oprd[p, t] + Short[p, t] - Short[p, t-1] +
            sum(Inv[p, t-1, a] - Inv[p, t, a] for a in 1:life) ==
            max(0, dem[p][t] - iil[p][t-1])
        )
    end
    # Inventory in storage at end of period t must meet specified minimum
    @constraint(
        prod,
        [p in 1:numprd, t in numperiods],
        sum(Inv[p, t, a] + iil[p][t] for a in 1:life) >= minv[p][t]
    )
    # In the vth period (starting from first) no inventory may be more than v
    # numperiods old (initial inventories are handled separately)
    @constraint(
        prod,
        [p in 1:numprd, v in 1:(life-1), a in (v+1):life],
        Inv[p, firstperiod+v-1, a] == 0
    )
    # New inventory cannot exceed production in the most recent period
    @constraint(
        prod,
        [p in 1:numprd, t in numperiods],
        Inv[p, t, 1] <= Rprd[p, t] + Oprd[p, t]
    )
    # Inventory left from period (t+1)-p can only decrease as time goes on
    secondperiod = firstperiod + 1
    @constraint(
        prod,
        [p in 1:numprd, t in 2:lastperiod, a in 2:life],
        Inv[p, t, a] <= Inv[p, t-1, a-1]
    )
    # OBJECTIVE
    # Full regular wages for all crews employed, plus penalties for hiring and
    # layoffs, plus wages for any overtime worked, plus inventory and shortage
    # costs. (All other production costs are assumed to depend on initial
    # inventory and on demands, and so are not included explicitly.)
    @objective(
        prod,
        Min,
        sum(
            rtr * sl * dpp[t] * cs * Crews[t] +
```

```
            hc[t] * Hire[t] +
            lc[t] * Layoff[t] +
            sum(
                otr * cs * pt[p] * Oprd[p, t] +
                sum(cri[p] * pc[p] * Inv[p, t, a] for a in 1:life) +
                crs[p] * pc[p] * Short[p, t] for p in 1:numprd
            ) for t in numperiods
        )
    )
    # Obtain solution
    optimize!(prod)
    Test.@test termination_status(prod) == OPTIMAL
    Test.@test primal_status(prod) == FEASIBLE_POINT
    Test.@test objective_value(prod) ≈ 4_426_822.89 atol = 1e-2
    if verbose
        println("RESULTS:")
        println("Crews")
        for t in 0:length(Crews.data)-1
            print(" $(value(Crews[t])) ")
        end
        println()
        println("Hire")
        for t in 1:length(Hire.data)
            print(" $(value(Hire[t])) ")
        end
        println()
        println("Layoff")
        for t in 1:length(Layoff.data)
            print(" $(value(Layoff[t])) ")
        end
        println()
    end
    return
end

example_prod()
```

```
RESULTS:
Crews
 8.0  6.439849038461538  5.947339134615386  5.947339134615383  5.947339134615383  5.947339134615383
      6.162946095647776  6.3400572039473655  7.805664375000001  7.805664375000001  7.805664375000001
      7.805664375000001  8.0  8.0
Hire
 0.0  0.0  0.0  0.0  0.0  0.21560696103239252  0.17711110829958976  1.4656071710526355  0.0  0.0
     0.0  0.194335624999999  0.0
Layoff
 1.5601509615384623  0.4925099038461518  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
```

Tip

This tutorial was generated using Literate.jl. View the source `.jl` file on GitHub.

4.13 The SteelT3 problem

The steelT3 model from AMPL: A Modeling Language for Mathematical Programming, 2nd ed by Robert Fourer, David Gay, and Brian W. Kernighan.

Originally contributed by Louis Luangkesorn, April 3, 2015.

```
using JuMP
import GLPK
import Test

function example_steelT3(; verbose = true)
    T = 4
    prod = ["bands", "coils"]
    area = Dict(
        "bands" => ("east", "north"),
        "coils" => ("east", "west", "export"),
    )
    avail = [40, 40, 32, 40]
    rate = Dict("bands" => 200, "coils" => 140)
    inv0 = Dict("bands" => 10, "coils" => 0)
    prodcost = Dict("bands" => 10, "coils" => 11)
    invcost = Dict("bands" => 2.5, "coils" => 3)
    revenue = Dict(
        "bands" => Dict(
            "east" => [25.0, 26.0, 27.0, 27.0],
            "north" => [26.5, 27.5, 28.0, 28.5],
        ),
        "coils" => Dict(
            "east" => [30, 35, 37, 39],
            "west" => [29, 32, 33, 35],
            "export" => [25, 25, 25, 28],
        ),
    )
    market = Dict(
        "bands" => Dict(
            "east" => [2000, 2000, 1500, 2000],
            "north" => [4000, 4000, 2500, 4500],
        ),
        "coils" => Dict(
            "east" => [1000, 800, 1000, 1100],
            "west" => [2000, 1200, 2000, 2300],
            "export" => [1000, 500, 500, 800],
        ),
    )
    # Model
    model = Model(GLPK.Optimizer)
    # Decision Variables
    @variables(
        model,
        begin
            make[p in prod, t in 1:T] >= 0
            inventory[p in prod, t in 0:T] >= 0
            0 <= sell[p in prod, a in area[p], t in 1:T] <= market[p][a][t]
        end
    )
```

```
    @constraints(
        model,
        begin
            [p = prod, a = area[p], t = 1:T], sell[p, a, t] <= market[p][a][t]
            # Total of hours used by all products may not exceed hours available,
            # in each week
            [t in 1:T], sum(1 / rate[p] * make[p, t] for p in prod) <= avail[t]
            # Initial inventory must equal given value
            [p in prod], inventory[p, 0] == inv0[p]
            # Tons produced and taken from inventory must equal tons sold and put
            # into inventory.
            [p in prod, t in 1:T],
            make[p, t] + inventory[p, t-1] ==
            sum(sell[p, a, t] for a in area[p]) + inventory[p, t]
        end
    )
    # Maximize total profit: total revenue less costs for all products in all
    # weeks.
    @objective(
        model,
        Max,
        sum(
            revenue[p][a][t] * sell[p, a, t] - prodcost[p] * make[p, t] -
            invcost[p] * inventory[p, t] for p in prod, a in area[p], t in 1:T
        )
    )
    optimize!(model)
    Test.@test termination_status(model) == OPTIMAL
    Test.@test primal_status(model) == FEASIBLE_POINT
    Test.@test objective_value(model) == 172850.0
    if verbose
        println("RESULTS:")
        for p in prod
            println("make $(p)")
            for t in 1:T
                print(value(make[p, t]), "\t")
            end
            println()
            println("Inventory $(p)")
            for t in 1:T
                print(value(inventory[p, t]), "\t")
            end
            println()
            for a in area[p]
                println("sell $(p) $(a)")
                for t in 1:T
                    print(value(sell[p, a, t]), "\t")
                end
                println()
            end
        end
    end
    return
end

example_steelT3()
```

```
RESULTS:
make bands
5990.0  6000.0  4000.0  6500.0
Inventory bands
0.0     0.0     0.0     0.0
sell bands east
2000.0  2000.0  1500.0  2000.0
sell bands north
4000.0  4000.0  2500.0  4500.0
make coils
0.0     800.0   1000.0  1050.0
Inventory coils
0.0     0.0     0.0     0.0
sell coils east
0.0     800.0   1000.0  1050.0
sell coils west
0.0     0.0     0.0     0.0
sell coils export
0.0     0.0     0.0     0.0
```

Tip

This tutorial was generated using Literate.jl. View the source `.jl` file on GitHub.

4.14 Sudoku

Originally Contributed by: Iain Dunning

Sudoku is a popular number puzzle. The goal is to place the digits 1,...,9 on a nine-by-nine grid, with some of the digits already filled in. Your solution must satisfy the following rules:

- The numbers 1 to 9 must appear in each 3x3 square
- The numbers 1 to 9 must appear in each row
- The numbers 1 to 9 must appear in each column

Here is a partially solved Sudoku problem:

Solving a Sudoku isn't an optimization problem with an objective; its actually a feasibility problem: we wish to find a feasible solution that satisfies these rules. You can think of it as an optimization problem with an objective of 0.

We can model this problem using 0-1 integer programming: a problem where all the decision variables are binary. We'll use JuMP to create the model, and then we can solve it with any integer programming solver.

```
using JuMP
using GLPK
```

We will define a binary variable (a variable that is either 0 or 1) for each possible number in each possible cell. The meaning of each variable is as follows: `x[i,j,k] = 1 if and only if cell (i,j) has number k`, where `i` is the row and `j` is the column.

Create a model

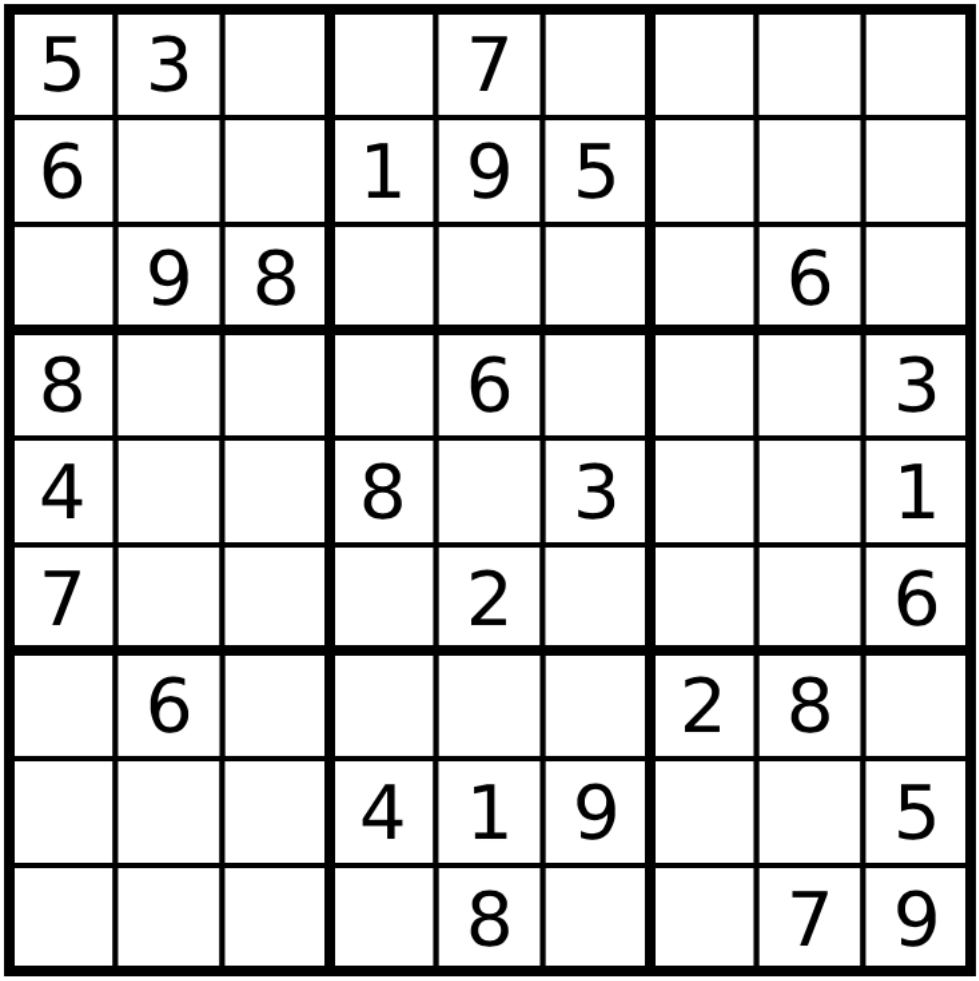


Figure 4.3: Partially solved Sudoku

```
sudoku = Model(GLPK.Optimizer)
```

```
A JuMP Model
Feasibility problem with:
Variables: 0
Model mode: AUTOMATIC
CachingOptimizer state: EMPTY_OPTIMIZER
Solver name: GLPK
```

Create our variables

```
@variable(sudoku, x[i = 1:9, j = 1:9, k = 1:9], Bin)
```

```
9×9×9 Array{VariableRef, 3}:
[:, :, 1] =
 x[1,1,1]  x[1,2,1]  x[1,3,1]  x[1,4,1]  …  x[1,7,1]  x[1,8,1]  x[1,9,1]
 x[2,1,1]  x[2,2,1]  x[2,3,1]  x[2,4,1]     x[2,7,1]  x[2,8,1]  x[2,9,1]
```

```
 x[3,1,1]  x[3,2,1]  x[3,3,1]  x[3,4,1]     x[3,7,1]  x[3,8,1]  x[3,9,1]
 x[4,1,1]  x[4,2,1]  x[4,3,1]  x[4,4,1]     x[4,7,1]  x[4,8,1]  x[4,9,1]
 x[5,1,1]  x[5,2,1]  x[5,3,1]  x[5,4,1]     x[5,7,1]  x[5,8,1]  x[5,9,1]
 x[6,1,1]  x[6,2,1]  x[6,3,1]  x[6,4,1]  …  x[6,7,1]  x[6,8,1]  x[6,9,1]
 x[7,1,1]  x[7,2,1]  x[7,3,1]  x[7,4,1]     x[7,7,1]  x[7,8,1]  x[7,9,1]
 x[8,1,1]  x[8,2,1]  x[8,3,1]  x[8,4,1]     x[8,7,1]  x[8,8,1]  x[8,9,1]
 x[9,1,1]  x[9,2,1]  x[9,3,1]  x[9,4,1]     x[9,7,1]  x[9,8,1]  x[9,9,1]

[:, :, 2] =
 x[1,1,2]  x[1,2,2]  x[1,3,2]  x[1,4,2]  …  x[1,7,2]  x[1,8,2]  x[1,9,2]
 x[2,1,2]  x[2,2,2]  x[2,3,2]  x[2,4,2]     x[2,7,2]  x[2,8,2]  x[2,9,2]
 x[3,1,2]  x[3,2,2]  x[3,3,2]  x[3,4,2]     x[3,7,2]  x[3,8,2]  x[3,9,2]
 x[4,1,2]  x[4,2,2]  x[4,3,2]  x[4,4,2]     x[4,7,2]  x[4,8,2]  x[4,9,2]
 x[5,1,2]  x[5,2,2]  x[5,3,2]  x[5,4,2]     x[5,7,2]  x[5,8,2]  x[5,9,2]
 x[6,1,2]  x[6,2,2]  x[6,3,2]  x[6,4,2]  …  x[6,7,2]  x[6,8,2]  x[6,9,2]
 x[7,1,2]  x[7,2,2]  x[7,3,2]  x[7,4,2]     x[7,7,2]  x[7,8,2]  x[7,9,2]
 x[8,1,2]  x[8,2,2]  x[8,3,2]  x[8,4,2]     x[8,7,2]  x[8,8,2]  x[8,9,2]
 x[9,1,2]  x[9,2,2]  x[9,3,2]  x[9,4,2]     x[9,7,2]  x[9,8,2]  x[9,9,2]

[:, :, 3] =
 x[1,1,3]  x[1,2,3]  x[1,3,3]  x[1,4,3]  …  x[1,7,3]  x[1,8,3]  x[1,9,3]
 x[2,1,3]  x[2,2,3]  x[2,3,3]  x[2,4,3]     x[2,7,3]  x[2,8,3]  x[2,9,3]
 x[3,1,3]  x[3,2,3]  x[3,3,3]  x[3,4,3]     x[3,7,3]  x[3,8,3]  x[3,9,3]
 x[4,1,3]  x[4,2,3]  x[4,3,3]  x[4,4,3]     x[4,7,3]  x[4,8,3]  x[4,9,3]
 x[5,1,3]  x[5,2,3]  x[5,3,3]  x[5,4,3]     x[5,7,3]  x[5,8,3]  x[5,9,3]
 x[6,1,3]  x[6,2,3]  x[6,3,3]  x[6,4,3]  …  x[6,7,3]  x[6,8,3]  x[6,9,3]
 x[7,1,3]  x[7,2,3]  x[7,3,3]  x[7,4,3]     x[7,7,3]  x[7,8,3]  x[7,9,3]
 x[8,1,3]  x[8,2,3]  x[8,3,3]  x[8,4,3]     x[8,7,3]  x[8,8,3]  x[8,9,3]
 x[9,1,3]  x[9,2,3]  x[9,3,3]  x[9,4,3]     x[9,7,3]  x[9,8,3]  x[9,9,3]

[:, :, 4] =
 x[1,1,4]  x[1,2,4]  x[1,3,4]  x[1,4,4]  …  x[1,7,4]  x[1,8,4]  x[1,9,4]
 x[2,1,4]  x[2,2,4]  x[2,3,4]  x[2,4,4]     x[2,7,4]  x[2,8,4]  x[2,9,4]
 x[3,1,4]  x[3,2,4]  x[3,3,4]  x[3,4,4]     x[3,7,4]  x[3,8,4]  x[3,9,4]
 x[4,1,4]  x[4,2,4]  x[4,3,4]  x[4,4,4]     x[4,7,4]  x[4,8,4]  x[4,9,4]
 x[5,1,4]  x[5,2,4]  x[5,3,4]  x[5,4,4]     x[5,7,4]  x[5,8,4]  x[5,9,4]
 x[6,1,4]  x[6,2,4]  x[6,3,4]  x[6,4,4]  …  x[6,7,4]  x[6,8,4]  x[6,9,4]
 x[7,1,4]  x[7,2,4]  x[7,3,4]  x[7,4,4]     x[7,7,4]  x[7,8,4]  x[7,9,4]
 x[8,1,4]  x[8,2,4]  x[8,3,4]  x[8,4,4]     x[8,7,4]  x[8,8,4]  x[8,9,4]
 x[9,1,4]  x[9,2,4]  x[9,3,4]  x[9,4,4]     x[9,7,4]  x[9,8,4]  x[9,9,4]

[:, :, 5] =
 x[1,1,5]  x[1,2,5]  x[1,3,5]  x[1,4,5]  …  x[1,7,5]  x[1,8,5]  x[1,9,5]
 x[2,1,5]  x[2,2,5]  x[2,3,5]  x[2,4,5]     x[2,7,5]  x[2,8,5]  x[2,9,5]
 x[3,1,5]  x[3,2,5]  x[3,3,5]  x[3,4,5]     x[3,7,5]  x[3,8,5]  x[3,9,5]
 x[4,1,5]  x[4,2,5]  x[4,3,5]  x[4,4,5]     x[4,7,5]  x[4,8,5]  x[4,9,5]
 x[5,1,5]  x[5,2,5]  x[5,3,5]  x[5,4,5]     x[5,7,5]  x[5,8,5]  x[5,9,5]
 x[6,1,5]  x[6,2,5]  x[6,3,5]  x[6,4,5]  …  x[6,7,5]  x[6,8,5]  x[6,9,5]
 x[7,1,5]  x[7,2,5]  x[7,3,5]  x[7,4,5]     x[7,7,5]  x[7,8,5]  x[7,9,5]
 x[8,1,5]  x[8,2,5]  x[8,3,5]  x[8,4,5]     x[8,7,5]  x[8,8,5]  x[8,9,5]
 x[9,1,5]  x[9,2,5]  x[9,3,5]  x[9,4,5]     x[9,7,5]  x[9,8,5]  x[9,9,5]

[:, :, 6] =
 x[1,1,6]  x[1,2,6]  x[1,3,6]  x[1,4,6]  …  x[1,7,6]  x[1,8,6]  x[1,9,6]
 x[2,1,6]  x[2,2,6]  x[2,3,6]  x[2,4,6]     x[2,7,6]  x[2,8,6]  x[2,9,6]
 x[3,1,6]  x[3,2,6]  x[3,3,6]  x[3,4,6]     x[3,7,6]  x[3,8,6]  x[3,9,6]
```

```
 x[4,1,6]  x[4,2,6]  x[4,3,6]  x[4,4,6]     x[4,7,6]  x[4,8,6]  x[4,9,6]
 x[5,1,6]  x[5,2,6]  x[5,3,6]  x[5,4,6]     x[5,7,6]  x[5,8,6]  x[5,9,6]
 x[6,1,6]  x[6,2,6]  x[6,3,6]  x[6,4,6]  …  x[6,7,6]  x[6,8,6]  x[6,9,6]
 x[7,1,6]  x[7,2,6]  x[7,3,6]  x[7,4,6]     x[7,7,6]  x[7,8,6]  x[7,9,6]
 x[8,1,6]  x[8,2,6]  x[8,3,6]  x[8,4,6]     x[8,7,6]  x[8,8,6]  x[8,9,6]
 x[9,1,6]  x[9,2,6]  x[9,3,6]  x[9,4,6]     x[9,7,6]  x[9,8,6]  x[9,9,6]

[:, :, 7] =
 x[1,1,7]  x[1,2,7]  x[1,3,7]  x[1,4,7]  …  x[1,7,7]  x[1,8,7]  x[1,9,7]
 x[2,1,7]  x[2,2,7]  x[2,3,7]  x[2,4,7]     x[2,7,7]  x[2,8,7]  x[2,9,7]
 x[3,1,7]  x[3,2,7]  x[3,3,7]  x[3,4,7]     x[3,7,7]  x[3,8,7]  x[3,9,7]
 x[4,1,7]  x[4,2,7]  x[4,3,7]  x[4,4,7]     x[4,7,7]  x[4,8,7]  x[4,9,7]
 x[5,1,7]  x[5,2,7]  x[5,3,7]  x[5,4,7]     x[5,7,7]  x[5,8,7]  x[5,9,7]
 x[6,1,7]  x[6,2,7]  x[6,3,7]  x[6,4,7]  …  x[6,7,7]  x[6,8,7]  x[6,9,7]
 x[7,1,7]  x[7,2,7]  x[7,3,7]  x[7,4,7]     x[7,7,7]  x[7,8,7]  x[7,9,7]
 x[8,1,7]  x[8,2,7]  x[8,3,7]  x[8,4,7]     x[8,7,7]  x[8,8,7]  x[8,9,7]
 x[9,1,7]  x[9,2,7]  x[9,3,7]  x[9,4,7]     x[9,7,7]  x[9,8,7]  x[9,9,7]

[:, :, 8] =
 x[1,1,8]  x[1,2,8]  x[1,3,8]  x[1,4,8]  …  x[1,7,8]  x[1,8,8]  x[1,9,8]
 x[2,1,8]  x[2,2,8]  x[2,3,8]  x[2,4,8]     x[2,7,8]  x[2,8,8]  x[2,9,8]
 x[3,1,8]  x[3,2,8]  x[3,3,8]  x[3,4,8]     x[3,7,8]  x[3,8,8]  x[3,9,8]
 x[4,1,8]  x[4,2,8]  x[4,3,8]  x[4,4,8]     x[4,7,8]  x[4,8,8]  x[4,9,8]
 x[5,1,8]  x[5,2,8]  x[5,3,8]  x[5,4,8]     x[5,7,8]  x[5,8,8]  x[5,9,8]
 x[6,1,8]  x[6,2,8]  x[6,3,8]  x[6,4,8]  …  x[6,7,8]  x[6,8,8]  x[6,9,8]
 x[7,1,8]  x[7,2,8]  x[7,3,8]  x[7,4,8]     x[7,7,8]  x[7,8,8]  x[7,9,8]
 x[8,1,8]  x[8,2,8]  x[8,3,8]  x[8,4,8]     x[8,7,8]  x[8,8,8]  x[8,9,8]
 x[9,1,8]  x[9,2,8]  x[9,3,8]  x[9,4,8]     x[9,7,8]  x[9,8,8]  x[9,9,8]

[:, :, 9] =
 x[1,1,9]  x[1,2,9]  x[1,3,9]  x[1,4,9]  …  x[1,7,9]  x[1,8,9]  x[1,9,9]
 x[2,1,9]  x[2,2,9]  x[2,3,9]  x[2,4,9]     x[2,7,9]  x[2,8,9]  x[2,9,9]
 x[3,1,9]  x[3,2,9]  x[3,3,9]  x[3,4,9]     x[3,7,9]  x[3,8,9]  x[3,9,9]
 x[4,1,9]  x[4,2,9]  x[4,3,9]  x[4,4,9]     x[4,7,9]  x[4,8,9]  x[4,9,9]
 x[5,1,9]  x[5,2,9]  x[5,3,9]  x[5,4,9]     x[5,7,9]  x[5,8,9]  x[5,9,9]
 x[6,1,9]  x[6,2,9]  x[6,3,9]  x[6,4,9]  …  x[6,7,9]  x[6,8,9]  x[6,9,9]
 x[7,1,9]  x[7,2,9]  x[7,3,9]  x[7,4,9]     x[7,7,9]  x[7,8,9]  x[7,9,9]
 x[8,1,9]  x[8,2,9]  x[8,3,9]  x[8,4,9]     x[8,7,9]  x[8,8,9]  x[8,9,9]
 x[9,1,9]  x[9,2,9]  x[9,3,9]  x[9,4,9]     x[9,7,9]  x[9,8,9]  x[9,9,9]
```

Now we can begin to add our constraints. We'll actually start with something obvious to us as humans, but what we need to enforce: that there can be only one number per cell.

```
for i in 1:9  ## For each row
    for j in 1:9  ## and each column
        # Sum across all the possible digits. One and only one of the digits
        # can be in this cell, so the sum must be equal to one.
        @constraint(sudoku, sum(x[i, j, k] for k in 1:9) == 1)
    end
end
```

Next we'll add the constraints for the rows and the columns. These constraints are all very similar, so much so that we can actually add them at the same time.

```
for ind in 1:9  ## Each row, OR each column
    for k in 1:9  ## Each digit
```

```
        # Sum across columns (j) - row constraint
        @constraint(sudoku, sum(x[ind, j, k] for j in 1:9) == 1)
        # Sum across rows (i) - column constraint
        @constraint(sudoku, sum(x[i, ind, k] for i in 1:9) == 1)
    end
end
```

Finally, we have the to enforce the constraint that each digit appears once in each of the nine 3x3 sub-grids. Our strategy will be to index over the top-left corners of each 3x3 square with `for` loops, then sum over the squares.

```
for i in 1:3:7
    for j in 1:3:7
        for k in 1:9
            # i is the top left row, j is the top left column.
            # We'll sum from i to i+2, e.g. i=4, r=4, 5, 6.
            @constraint(
                sudoku,
                sum(x[r, c, k] for r in i:(i+2), c in j:(j+2)) == 1
            )
        end
    end
end
```

The final step is to add the initial solution as a set of constraints. We'll solve the problem that is in the picture at the start of the tutorial. We'll put a `0` if there is no digit in that location.

The given digits

```
init_sol = [
    5 3 0 0 7 0 0 0 0
    6 0 0 1 9 5 0 0 0
    0 9 8 0 0 0 0 6 0
    8 0 0 0 6 0 0 0 3
    4 0 0 8 0 3 0 0 1
    7 0 0 0 2 0 0 0 6
    0 6 0 0 0 0 2 8 0
    0 0 0 4 1 9 0 0 5
    0 0 0 0 8 0 0 7 9
]
for i in 1:9
    for j in 1:9
        # If the space isn't empty
        if init_sol[i, j] != 0
            # Then the corresponding variable for that digit and location must
            # be 1.
            fix(x[i, j, init_sol[i, j]], 1; force = true)
        end
    end
end
```

solve problem

```
optimize!(sudoku)
```

Extract the values of x

```
x_val = value.(x)
```

```
9×9×9 Array{Float64, 3}:
[:, :, 1] =
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  1.0  0.0
 0.0  0.0  0.0  1.0  0.0  0.0  0.0  0.0  0.0
 1.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  1.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  1.0
 0.0  1.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  1.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  1.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  1.0  0.0  0.0

[:, :, 2] =
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  1.0
 0.0  0.0  1.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  1.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  1.0  0.0
 0.0  1.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  1.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  1.0  0.0  0.0
 1.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  1.0  0.0  0.0  0.0  0.0  0.0

[:, :, 3] =
 0.0  1.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  1.0  0.0  0.0
 0.0  0.0  0.0  1.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  1.0
 0.0  0.0  0.0  0.0  0.0  1.0  0.0  0.0  0.0
 0.0  0.0  1.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  1.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  1.0  0.0
 1.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0

[:, :, 4] =
 0.0  0.0  1.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  1.0  0.0
 0.0  0.0  0.0  0.0  1.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  1.0  0.0  0.0
 1.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  1.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  1.0
 0.0  0.0  0.0  1.0  0.0  0.0  0.0  0.0  0.0
 0.0  1.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0

[:, :, 5] =
 1.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  1.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  1.0  0.0  0.0
 0.0  1.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  1.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  1.0  0.0
```

```
 0.0  0.0  0.0  1.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  1.0
 0.0  0.0  1.0  0.0  0.0  0.0  0.0  0.0  0.0

[:, :, 6] =
 0.0  0.0  0.0  1.0  0.0  0.0  0.0  0.0  0.0
 1.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  1.0  0.0
 0.0  0.0  0.0  0.0  1.0  0.0  0.0  0.0  0.0
 0.0  0.0  1.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  1.0
 0.0  1.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  1.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  1.0  0.0  0.0  0.0

[:, :, 7] =
 0.0  0.0  0.0  0.0  1.0  0.0  0.0  0.0  0.0
 0.0  1.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  1.0
 0.0  0.0  0.0  1.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  1.0  0.0  0.0
 1.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  1.0  0.0  0.0  0.0
 0.0  0.0  1.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  1.0  0.0

[:, :, 8] =
 0.0  0.0  0.0  0.0  0.0  1.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  1.0
 0.0  0.0  1.0  0.0  0.0  0.0  0.0  0.0  0.0
 1.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  1.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  1.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  1.0  0.0
 0.0  1.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  1.0  0.0  0.0  0.0  0.0

[:, :, 9] =
 0.0  0.0  0.0  0.0  0.0  0.0  1.0  0.0  0.0
 0.0  0.0  0.0  0.0  1.0  0.0  0.0  0.0  0.0
 0.0  1.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  1.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  1.0  0.0
 0.0  0.0  0.0  1.0  0.0  0.0  0.0  0.0  0.0
 1.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  1.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  1.0
```

Create a matrix to store the solution

```
sol = zeros(Int, 9, 9)  # 9x9 matrix of integers
for i in 1:9
    for j in 1:9
        for k in 1:9
            # Integer programs are solved as a series of linear programs so the
            # values might not be precisely 0 and 1. We can just round them to
```

```
            # the nearest integer to make it easier.
            if round(Int, x_val[i, j, k]) == 1
                sol[i, j] = k
            end
        end
    end
end
```

Display the solution

```
sol
```

```
9×9 Matrix{Int64}:
 5  3  4  6  7  8  9  1  2
 6  7  2  1  9  5  3  4  8
 1  9  8  3  4  2  5  6  7
 8  5  9  7  6  1  4  2  3
 4  2  6  8  5  3  7  9  1
 7  1  3  9  2  4  8  5  6
 9  6  1  5  3  7  2  8  4
 2  8  7  4  1  9  6  3  5
 3  4  5  2  8  6  1  7  9
```

Which is the correct solution:

Tip

This tutorial was generated using Literate.jl. View the source `.jl` file on GitHub.

4.15 The transportation problem

Allocation of passenger cars to trains to minimize cars required or car-miles run. Based on:

Fourer, D.M. Gay and Brian W. Kernighan, A Modeling Language for Mathematical Programming, https://ampl.com/REFS/amplmod.ps.gz Appendix D.

Originally contributed by Louis Luangkesorn, January 30, 2015.

```
using JuMP
import GLPK
import Test

function example_transp()
    ORIG = ["GARY", "CLEV", "PITT"]
    DEST = ["FRA", "DET", "LAN", "WIN", "STL", "FRE", "LAF"]
    supply = [1_400, 2_600, 2_900]
    demand = [900, 1_200, 600, 400, 1_700, 1_100, 1_000]
    Test.@test sum(supply) == sum(demand)
    cost = [
        39 14 11 14 16 82 8
        27 9 12 9 26 95 17
        24 14 17 13 28 99 20
    ]
```

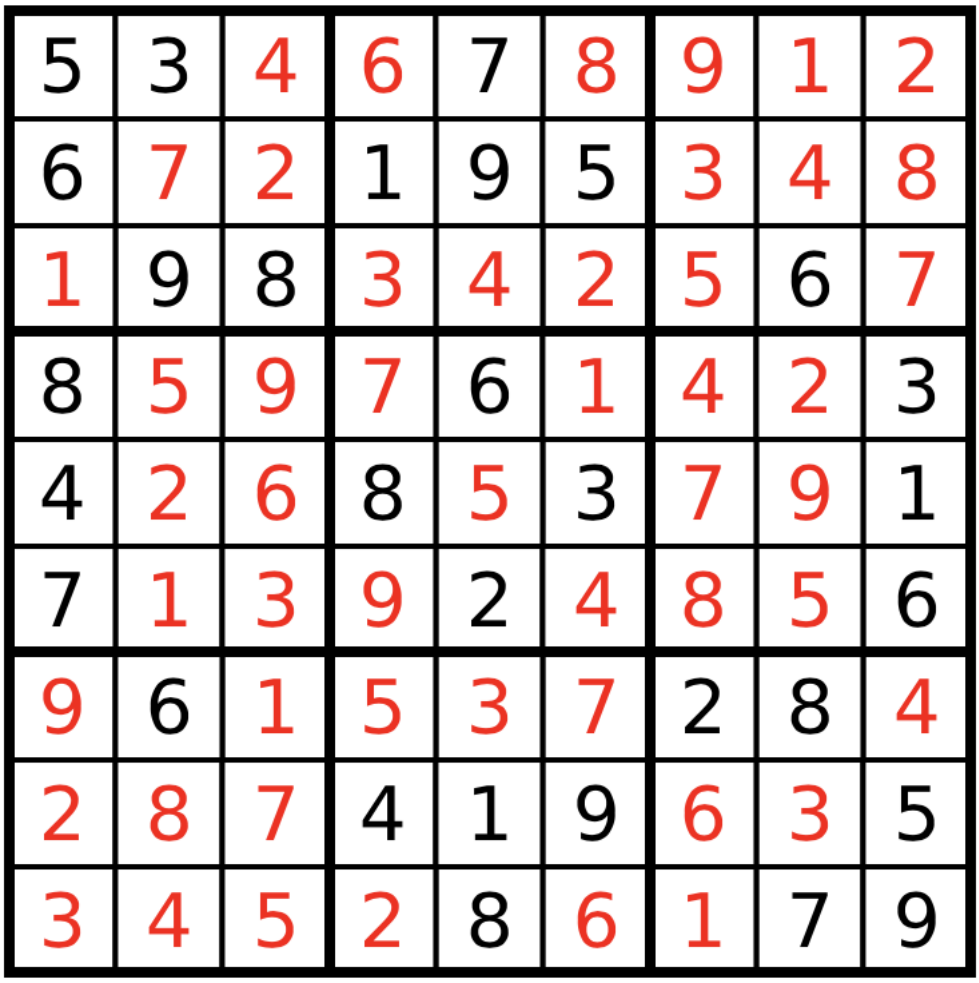


Figure 4.4: Solved Sudoku

```
model = Model(GLPK.Optimizer)
@variable(model, trans[1:length(ORIG), 1:length(DEST)] >= 0)
@objective(
    model,
    Min,
    sum(
        cost[i, j] * trans[i, j] for i in 1:length(ORIG),
        j in 1:length(DEST)
    )
)
@constraints(
    model,
    begin
        [i in 1:length(ORIG)], sum(trans[i, :]) == supply[i]
        [j in 1:length(DEST)], sum(trans[:, j]) == demand[j]
    end
)
optimize!(model)
```

```
    Test.@test termination_status(model) == OPTIMAL
    Test.@test primal_status(model) == FEASIBLE_POINT
    Test.@test objective_value(model) == 196200.0
    println("The optimal solution is:")
    println(value.(trans))
    return
end

example_transp()
```

```
The optimal solution is:
[0.0 0.0 0.0 0.0 300.0 1100.0 0.0; 0.0 1200.0 600.0 400.0 0.0 0.0 400.0; 900.0 0.0 0.0 0.0 1400.0
     0.0 600.0]
```

Tip

This tutorial was generated using Literate.jl. View the source `.jl` file on GitHub.

4.16 The urban planning problem

An "urban planning" problem based on an example from puzzlor.

```
using JuMP
import GLPK
import Test

function example_urban_plan()
    model = Model(GLPK.Optimizer)
    # x is indexed by row and column
    @variable(model, 0 <= x[1:5, 1:5] <= 1, Int)
    # y is indexed by R or C, the points, and an index in 1:5. Note how JuMP
    # allows indexing on arbitrary sets.
    rowcol = ["R", "C"]
    points = [5, 4, 3, -3, -4, -5]
    @variable(model, 0 <= y[rowcol, points, 1:5] <= 1, Int)
    # Objective - combine the positive and negative parts
    @objective(
        model,
        Max,
        sum(
            3 * (y["R", 3, i] + y["C", 3, i]) +
            1 * (y["R", 4, i] + y["C", 4, i]) +
            1 * (y["R", 5, i] + y["C", 5, i]) -
            3 * (y["R", -3, i] + y["C", -3, i]) -
            1 * (y["R", -4, i] + y["C", -4, i]) -
            1 * (y["R", -5, i] + y["C", -5, i]) for i in 1:5
        )
    )
    # Constrain the number of residential lots
    @constraint(model, sum(x) == 12)
    # Add the constraints that link the auxiliary y variables to the x variables
    for i in 1:5
        @constraints(model, begin
```

```
            # Rows
            y["R", 5, i] <= 1 / 5 * sum(x[i, :]) # sum = 5
            y["R", 4, i] <= 1 / 4 * sum(x[i, :]) # sum = 4
            y["R", 3, i] <= 1 / 3 * sum(x[i, :]) # sum = 3
            y["R", -3, i] >= 1 - 1 / 3 * sum(x[i, :]) # sum = 2
            y["R", -4, i] >= 1 - 1 / 2 * sum(x[i, :]) # sum = 1
            y["R", -5, i] >= 1 - 1 / 1 * sum(x[i, :]) # sum = 0
            # Columns
            y["C", 5, i] <= 1 / 5 * sum(x[:, i]) # sum = 5
            y["C", 4, i] <= 1 / 4 * sum(x[:, i]) # sum = 4
            y["C", 3, i] <= 1 / 3 * sum(x[:, i]) # sum = 3
            y["C", -3, i] >= 1 - 1 / 3 * sum(x[:, i]) # sum = 2
            y["C", -4, i] >= 1 - 1 / 2 * sum(x[:, i]) # sum = 1
            y["C", -5, i] >= 1 - 1 / 1 * sum(x[:, i]) # sum = 0
        end)
    end
    # Solve it
    optimize!(model)
    Test.@test termination_status(model) == OPTIMAL
    Test.@test primal_status(model) == FEASIBLE_POINT
    Test.@test objective_value(model) ≈ 14.0
    return
end

example_urban_plan()
```

Tip

This tutorial was generated using Literate.jl. View the source `.jl` file on GitHub.

4.17 Callbacks

This example uses the following packages:

```
using JuMP
import GLPK
import Random
```

Lazy constraints

An example using a lazy constraint callback.

```
function example_lazy_constraint()
    model = Model(GLPK.Optimizer)
    @variable(model, 0 <= x <= 2.5, Int)
    @variable(model, 0 <= y <= 2.5, Int)
    @objective(model, Max, y)
    lazy_called = false
    function my_callback_function(cb_data)
        lazy_called = true
        x_val = callback_value(cb_data, x)
        y_val = callback_value(cb_data, y)
```

```
        println("Called from (x, y) = ($x_val, $y_val)")
        status = callback_node_status(cb_data, model)
        if status == MOI.CALLBACK_NODE_STATUS_FRACTIONAL
            println(" - Solution is integer infeasible!")
        elseif status == MOI.CALLBACK_NODE_STATUS_INTEGER
            println(" - Solution is integer feasible!")
        else
            @assert status == MOI.CALLBACK_NODE_STATUS_UNKNOWN
            println(" - I don't know if the solution is integer feasible :(")
        end
        if y_val - x_val > 1 + 1e-6
            con = @build_constraint(y - x <= 1)
            println("Adding $(con)")
            MOI.submit(model, MOI.LazyConstraint(cb_data), con)
        elseif y_val + x_val > 3 + 1e-6
            con = @build_constraint(y - x <= 1)
            println("Adding $(con)")
            MOI.submit(model, MOI.LazyConstraint(cb_data), con)
        end
    end
    MOI.set(model, MOI.LazyConstraintCallback(), my_callback_function)
    optimize!(model)
    println("Optimal solution (x, y) = ($(value(x)), $(value(y)))")
    return
end

example_lazy_constraint()
```

```
Called from (x, y) = (0.0, 2.0)
 - Solution is integer feasible!
Adding ScalarConstraint{AffExpr, MathOptInterface.LessThan{Float64}}(y - x, MathOptInterface.
    LessThan{Float64}(1.0))
Called from (x, y) = (1.0, 2.0)
 - Solution is integer feasible!
Optimal solution (x, y) = (1.0, 2.0)
```

User-cut

An example using a user-cut callback.

```
function example_user_cut_constraint()
    Random.seed!(1)
    N = 30
    item_weights, item_values = rand(N), rand(N)
    model = Model(GLPK.Optimizer)
    @variable(model, x[1:N], Bin)
    @constraint(model, sum(item_weights[i] * x[i] for i in 1:N) <= 10)
    @objective(model, Max, sum(item_values[i] * x[i] for i in 1:N))
    callback_called = false
    function my_callback_function(cb_data)
        callback_called = true
        x_vals = callback_value.(Ref(cb_data), x)
        accumulated = sum(item_weights[i] for i = 1:N if x_vals[i] > 1e-4)
        println("Called with accumulated = $(accumulated)")
        n_terms = sum(1 for i = 1:N if x_vals[i] > 1e-4)
```

```
        if accumulated > 10
            con = @build_constraint(
                sum(x[i] for i = 1:N if x_vals[i] > 0.5) <= n_terms - 1
            )
            println("Adding $(con)")
            MOI.submit(model, MOI.UserCut(cb_data), con)
        end
    end
    MOI.set(model, MOI.UserCutCallback(), my_callback_function)
    optimize!(model)
    @show callback_called
    return
end

example_user_cut_constraint()
```

```
Called with accumulated = 10.245300779183612
Adding ScalarConstraint{AffExpr, MathOptInterface.LessThan{Float64}}(x[1] + x[2] + x[4] + x[6] + x
     [7] + x[9] + x[10] + x[11] + x[12] + x[13] + x[14] + x[15] + x[16] + x[17] + x[18] + x[19] + x
     [20] + x[21] + x[22] + x[23] + x[24] + x[26] + x[29] + x[30], MathOptInterface.LessThan{Float64
     }(23.0))
Called with accumulated = 10.276817515233951
Adding ScalarConstraint{AffExpr, MathOptInterface.LessThan{Float64}}(x[1] + x[2] + x[4] + x[6] + x
     [7] + x[9] + x[10] + x[11] + x[12] + x[13] + x[14] + x[16] + x[17] + x[18] + x[19] + x[20] + x
     [21] + x[22] + x[23] + x[24] + x[26] + x[29] + x[30], MathOptInterface.LessThan{Float64}(23.0))
Called with accumulated = 10.812296027897915
Adding ScalarConstraint{AffExpr, MathOptInterface.LessThan{Float64}}(x[1] + x[2] + x[4] + x[6] + x
     [7] + x[9] + x[10] + x[11] + x[12] + x[13] + x[14] + x[16] + x[17] + x[18] + x[19] + x[20] + x
     [21] + x[22] + x[23] + x[24] + x[26] + x[29] + x[30], MathOptInterface.LessThan{Float64}(23.0))
callback_called = true
```

HeuristicCallback

An example using a heuristic solution callback.

```
function example_heuristic_solution()
    Random.seed!(1)
    N = 30
    item_weights, item_values = rand(N), rand(N)
    model = Model(GLPK.Optimizer)
    @variable(model, x[1:N], Bin)
    @constraint(model, sum(item_weights[i] * x[i] for i in 1:N) <= 10)
    @objective(model, Max, sum(item_values[i] * x[i] for i in 1:N))
    callback_called = false
    function my_callback_function(cb_data)
        callback_called = true
        x_vals = callback_value.(Ref(cb_data), x)
        ret =
            MOI.submit(model, MOI.HeuristicSolution(cb_data), x, floor.(x_vals))
        println("Heuristic solution status = $(ret)")
    end
    MOI.set(model, MOI.HeuristicCallback(), my_callback_function)
    optimize!(model)
    return
end
```

```
example_heuristic_solution()
```

```
Heuristic solution status = HEURISTIC_SOLUTION_ACCEPTED
Heuristic solution status = HEURISTIC_SOLUTION_ACCEPTED
Heuristic solution status = HEURISTIC_SOLUTION_REJECTED
Heuristic solution status = HEURISTIC_SOLUTION_REJECTED
```

GLPK solver-dependent callback

An example using GLPK's solver-dependent callback.

```
function example_solver_dependent_callback()
    model = Model(GLPK.Optimizer)
    @variable(model, 0 <= x <= 2.5, Int)
    @variable(model, 0 <= y <= 2.5, Int)
    @objective(model, Max, y)
    lazy_called = false
    function my_callback_function(cb_data)
        lazy_called = true
        reason = GLPK.glp_ios_reason(cb_data.tree)
        println("Called from reason = $(reason)")
        if reason != GLPK.GLP_IROWGEN
            return
        end
        x_val = callback_value(cb_data, x)
        y_val = callback_value(cb_data, y)
        if y_val - x_val > 1 + 1e-6
            con = @build_constraint(y - x <= 1)
            println("Adding $(con)")
            MOI.submit(model, MOI.LazyConstraint(cb_data), con)
        elseif y_val + x_val > 3 + 1e-6
            con = @build_constraint(y - x <= 1)
            println("Adding $(con)")
            MOI.submit(model, MOI.LazyConstraint(cb_data), con)
        end
    end
    MOI.set(model, GLPK.CallbackFunction(), my_callback_function)
    optimize!(model)
    return
end

example_solver_dependent_callback()
```

```
Called from reason = 6
Called from reason = 7
Called from reason = 1
Adding ScalarConstraint{AffExpr, MathOptInterface.LessThan{Float64}}(y - x, MathOptInterface.
    LessThan{Float64}(1.0))
Called from reason = 7
Called from reason = 1
Called from reason = 2
```

Tip

This tutorial was generated using Literate.jl. View the source `.jl` file on GitHub.

Chapter 5

Nonlinear programs

5.1 Tips and tricks

This example collates some tips and tricks you can use when formulating nonlinear programs. It uses the following packages:

```
using JuMP
import Ipopt
import Test
```

User-defined functions with vector outputs

A common situation is to have a user-defined function like the following that returns multiple outputs (we define `function_calls` to keep track of how many times we call this method):

```
function_calls = 0
function foo(x, y)
    global function_calls += 1
    common_term = x^2 + y^2
    term_1 = sqrt(1 + common_term)
    term_2 = common_term
    return term_1, term_2
end
```

```
foo (generic function with 1 method)
```

For example, the first term might be used in the objective, and the second term might be used in a constraint, and often they share share work that is expensive to evaluate.

This is a problem for JuMP, because it requires user-defined functions to return a single number. One option is to define two separate functions, the first returning the first argument, and the second returning the second argument.

```
foo_1(x, y) = foo(x, y)[1]
foo_2(x, y) = foo(x, y)[2]
```

```
foo_2 (generic function with 1 method)
```

However, if the common term is expensive to compute, this approach is wasteful because it will evaluate the expensive term twice. Let's have a look at how many times we evaluate x^2 + y^2 during a solve:

```
model = Model(Ipopt.Optimizer)
set_silent(model)
@variable(model, x[1:2] >= 0, start = 0.1)
register(model, :foo_1, 2, foo_1; autodiff = true)
register(model, :foo_2, 2, foo_2; autodiff = true)
@NLobjective(model, Max, foo_1(x[1], x[2]))
@NLconstraint(model, foo_2(x[1], x[2]) <= 2)
function_calls = 0
optimize!(model)
Test.@test objective_value(model) ≈ √3 atol = 1e-4
Test.@test value.(x) ≈ [1.0, 1.0] atol = 1e-4
println("Naive approach: function calls = $(function_calls)")
```

```
Naive approach: function calls = 40
```

An alternative approach is to use memoization, which uses a cache to store the result of function evaluations. We can write a memoization function as follows:

```
"""
    memoize(foo::Function, n_outputs::Int)

Take a function `foo` and return a vector of length `n_outputs`, where each
element is a function that returns the `i`'th output of `foo`.

To avoid duplication of work, cache the most-recent evaluations of `foo`.
Because `foo_i` is auto-differentiated with ForwardDiff, our cache needs to
work when `x` is a `Float64` and a `ForwardDiff.Dual`.
"""
function memoize(foo::Function, n_outputs::Int)
    last_x, last_f = nothing, nothing
    last_dx, last_dfdx = nothing, nothing
    function foo_i(i, x::T...) where {T<:Real}
        if T == Float64
            if x != last_x
                last_x, last_f = x, foo(x...)
            end
            return last_f[i]::T
        else
            if x != last_dx
                last_dx, last_dfdx = x, foo(x...)
            end
            return last_dfdx[i]::T
        end
    end
    return [(x...) -> foo_i(i, x...) for i in 1:n_outputs]
end
```

```
Main.memoize
```

Let's see how it works. First, construct the memoized versions of `foo`:

```
memoized_foo = memoize(foo, 2)
```

```
2-element Vector{Main.var"#4#7"{Int64, Main.var"#foo_i#5"{typeof(Main.foo)}}}:
 #4 (generic function with 1 method)
 #4 (generic function with 1 method)
```

Now try evaluating the first element of `memoized_foo`.

```
function_calls = 0
memoized_foo[1](1.0, 1.0)
println("function_calls = ", function_calls)
```

```
function_calls = 1
```

As expected, this evaluated the function once. However, if we call the function again, we hit the cache instead of needing to re-compute `foo` and `function_calls` is still 1!

```
memoized_foo[1](1.0, 1.0)
println("function_calls = ", function_calls)
```

```
function_calls = 1
```

Now let's see how this works during a real solve:

```
model = Model(Ipopt.Optimizer)
set_silent(model)
@variable(model, x[1:2] >= 0, start = 0.1)
register(model, :foo_1, 2, memoized_foo[1]; autodiff = true)
register(model, :foo_2, 2, memoized_foo[2]; autodiff = true)
@NLobjective(model, Max, foo_1(x[1], x[2]))
@NLconstraint(model, foo_2(x[1], x[2]) <= 2)
function_calls = 0
optimize!(model)
Test.@test objective_value(model) ≈ √3 atol = 1e-4
Test.@test value.(x) ≈ [1.0, 1.0] atol = 1e-4
println("Memoized approach: function_calls = $(function_calls)")
```

```
Memoized approach: function_calls = 20
```

Compared to the naive approach, the memoized approach requires half as many function evaluations!

Tip

This tutorial was generated using Literate.jl. View the source `.jl` file on GitHub.

5.2 Portfolio Optimization

Originally Contributed by: Arpit Bhatia

Optimization models play an increasingly important role in financial decisions. Many computational finance problems can be solved efficiently using modern optimization techniques.

This tutorial solves the famous Markowitz Portfolio Optimization problem with data from lecture notes from a course taught at Georgia Tech by Shabir Ahmed.

This tutorial uses the following packages

```
using JuMP
import Ipopt
import Statistics
```

Suppose we are considering investing 1000 dollars in three non-dividend paying stocks, IBM (IBM), Walmart (WMT), and Southern Electric (SEHI), for a one-month period.

This means we will use the money to buy shares of the three stocks at the current market prices, hold these for one month, and sell the shares off at the prevailing market prices at the end of the month.

As a rational investor, we hope to make some profit out of this endeavor, i.e., the return on our investment should be positive.

Suppose we bought a stock at p dollars per share in the beginning of the month, and sold it off at s dollars per share at the end of the month. Then the one-month return on a share of the stock is $\frac{s-p}{p}$.

Since the stock prices are quite uncertain, so is the end-of-month return on our investment. Our goal is to invest in such a way that the expected end-of-month return is at least \$50 or 5%. Furthermore, we want to make sure that the "risk" of not achieving our desired return is minimum.

Note that we are solving the problem under the following assumptions:

1. We can trade any continuum of shares.
2. No short-selling is allowed.
3. There are no transaction costs.

We model this problem by taking decision variables $x_i, i = 1, 2, 3$, denoting the dollars invested in each of the 3 stocks.

Let us denote by $\tilde{r}_i$ the random variable corresponding to the monthly return (increase in the stock price) per dollar for stock i.

Then, the return (or profit) on x_i dollars invested in stock i is $\tilde{r}_i x_i$, and the total (random) return on our investment is $\sum_{i=1}^{3} \tilde{r}_i x_i$. The expected return on our investment is then $\mathbb{E}\left[\sum_{i=1}^{3} \tilde{r}_i x_i\right] = \sum_{i=1}^{3} \overline{r}_i x_i$, where $\overline{r}_i$ is the expected value of the $\tilde{r}_i$.

Now we need to quantify the notion of "risk" in our investment.

Markowitz, in his Nobel prize winning work, showed that a rational investor's notion of minimizing risk can be closely approximated by minimizing the variance of the return of the investment portfolio. This variance is given by:

$$\text{Var}\left[\sum_{i=1}^{3} \tilde{r}_i x_i\right] = \sum_{i=1}^{3}\sum_{j=1}^{3} x_i x_j \sigma_{ij}$$

where σ_{ij} is the covariance of the return of stock i with stock j.

Note that the right hand side of the equation is the most reduced form of the expression and we have not shown the intermediate steps involved in getting to this form. We can also write this equation as:

$$\text{Var}\left[\sum_{i=1}^{3} \tilde{r}_i x_i\right] = x^T Q x$$

Where Q is the covariance matrix for the random vector $\tilde{r}$.

Finally, we can write the model as:

$$\begin{aligned} & \min x^T Q x \\ \text{s.t. } & \sum_{i=1}^{3} x_i \leq 1000.00 \\ & \overline{r}^T x \geq 50.00 \\ & x \geq 0 \end{aligned}$$

After that long discussion, lets now use JuMP to solve the portfolio optimization problem for the data given below.

Month	IBM	WMT	SEHI
November-00	93.043	51.826	1.063
December-00	84.585	52.823	0.938
January-01	111.453	56.477	1.000
February-01	99.525	49.805	0.938
March-01	95.819	50.287	1.438
April-01	114.708	51.521	1.700
May-01	111.515	51.531	2.540
June-01	113.211	48.664	2.390
July-01	104.942	55.744	3.120
August-01	99.827	47.916	2.980
September-01	91.607	49.438	1.900
October-01	107.937	51.336	1.750
November-01	115.590	55.081	1.800

```
stock_data = [
    93.043 51.826 1.063
    84.585 52.823 0.938
    111.453 56.477 1.000
    99.525 49.805 0.938
    95.819 50.287 1.438
```

```
    114.708 51.521 1.700
    111.515 51.531 2.540
    113.211 48.664 2.390
    104.942 55.744 3.120
    99.827 47.916 2.980
    91.607 49.438 1.900
    107.937 51.336 1.750
    115.590 55.081 1.800
]
```

```
13×3 Matrix{Float64}:
  93.043  51.826  1.063
  84.585  52.823  0.938
 111.453  56.477  1.0
  99.525  49.805  0.938
  95.819  50.287  1.438
 114.708  51.521  1.7
 111.515  51.531  2.54
 113.211  48.664  2.39
 104.942  55.744  3.12
  99.827  47.916  2.98
  91.607  49.438  1.9
 107.937  51.336  1.75
 115.59   55.081  1.8
```

Calculating stock returns

```
stock_returns = Array{Float64}(undef, 12, 3)
for i in 1:12
    stock_returns[i, :] =
        (stock_data[i+1, :] .- stock_data[i, :]) ./ stock_data[i, :]
end
stock_returns
```

```
12×3 Matrix{Float64}:
 -0.0909042   0.0192374    -0.117592
  0.317645    0.0691744     0.0660981
 -0.107023   -0.118137     -0.062
 -0.0372369   0.00967774    0.533049
  0.197132    0.0245391     0.182197
 -0.0278359   0.000194096   0.494118
  0.0152087  -0.0556364    -0.0590551
 -0.0730406   0.145487      0.305439
 -0.0487412  -0.140428     -0.0448718
 -0.0823425   0.0317639    -0.362416
  0.178261    0.0383915    -0.0789474
  0.0709025   0.0729508     0.0285714
```

Calculating the expected value of monthly return:

```
r = Statistics.mean(stock_returns, dims = 1)
```

```
1×3 Matrix{Float64}:
 0.0260022  0.00810132  0.0737159
```

Calculating the covariance matrix Q

```
Q = Statistics.cov(stock_returns)
```

```
3×3 Matrix{Float64}:
 0.018641    0.00359853  0.00130976
 0.00359853  0.00643694  0.00488727
 0.00130976  0.00488727  0.0686828
```

JuMP Model

```
portfolio = Model(Ipopt.Optimizer)
set_silent(portfolio)
@variable(portfolio, x[1:3] >= 0)
@objective(portfolio, Min, x' * Q * x)
@constraint(portfolio, sum(x) <= 1000)
@constraint(portfolio, sum(r[i] * x[i] for i in 1:3) >= 50)
optimize!(portfolio)

objective_value(portfolio)
```

```
22634.417849884143
```

```
value.(x)
```

```
3-element Vector{Float64}:
 497.0455298498641
   0.0
 502.95448015948085
```

Tip

This tutorial was generated using Literate.jl. View the source `.jl` file on GitHub.

5.3 Quadratically constrained programs

A simple quadratically constrained program based on an example from Gurobi.

```
using JuMP
import Ipopt
import Test

function example_qcp(; verbose = true)
    model = Model(Ipopt.Optimizer)
    set_silent(model)
    @variable(model, x)
    @variable(model, y >= 0)
    @variable(model, z >= 0)
    @objective(model, Max, x)
    @constraint(model, x + y + z == 1)
```

```
    @constraint(model, x * x + y * y - z * z <= 0)
    @constraint(model, x * x - y * z <= 0)
    optimize!(model)
    if verbose
        print(model)
        println("Objective value: ", objective_value(model))
        println("x = ", value(x))
        println("y = ", value(y))
    end
    Test.@test termination_status(model) == LOCALLY_SOLVED
    Test.@test primal_status(model) == FEASIBLE_POINT
    Test.@test objective_value(model) ≈ 0.32699 atol = 1e-5
    Test.@test value(x) ≈ 0.32699 atol = 1e-5
    Test.@test value(y) ≈ 0.25707 atol = 1e-5
    return
end

example_qcp()
```

```
Max x
Subject to
 x + y + z = 1.0
 x² + y² - z² ≤ 0.0
 x² - z*y ≤ 0.0
 y ≥ 0.0
 z ≥ 0.0
Objective value: 0.3269928349138724
x = 0.3269928349138724
y = 0.2570658388068964
```

Tip

This tutorial was generated using Literate.jl. View the source `.jl` file on GitHub.

5.4 Space Shuttle Reentry Trajectory

Originally Contributed by: Henrique Ferrolho

This tutorial demonstrates how to compute a reentry trajectory for the Space Shuttle, by formulating and solving a nonlinear programming problem. The problem was drawn from Chapter 6 of "Practical Methods for Optimal Control and Estimation Using Nonlinear Programming", by John T. Betts.

Tip

This tutorial is a more-complicated version of the Rocket Control example. If you are new to solving nonlinear programs in JuMP, you may want to start there instead.

The motion of the vehicle is defined by the following set of DAEs:

$$
\begin{aligned}
\dot{h} &= v \sin\gamma, \\
\dot{\phi} &= \frac{v}{r} \cos\gamma \sin\psi / \cos\theta, \\
\dot{\theta} &= \frac{v}{r} \cos\gamma \cos\psi, \\
\dot{v} &= -\frac{D}{m} - g \sin\gamma, \\
\dot{\gamma} &= \frac{L}{mv} \cos(\beta) + \cos\gamma \left(\frac{v}{r} - \frac{g}{v} \right), \\
\dot{\psi} &= \frac{1}{mv \cos\gamma} L \sin(\beta) + \frac{v}{r \cos\theta} \cos\gamma \sin\psi \sin\theta, \\
q &\leq q_U,
\end{aligned}
$$

where the aerodynamic heating on the vehicle wing leading edge is $q = q_a q_r$ and the dynamic variables are

h	altitude (ft),	γ	flight path angle (rad),
ϕ	longitude (rad),	ψ	azimuth (rad),
θ	latitude (rad),	α	angle of attack (rad),
v	velocity (ft/sec),	β	bank angle (rad).

The aerodynamic and atmospheric forces on the vehicle are specified by the following quantities (English units):

$$
\begin{aligned}
D &= \frac{1}{2} c_D S \rho v^2, & a_0 &= -0.20704, \\
L &= \frac{1}{2} c_L S \rho v^2, & a_1 &= 0.029244, \\
g &= \mu / r^2, & \mu &= 0.14076539 \times 10^{17}, \\
r &= R_e + h, & b_0 &= 0.07854, \\
\rho &= \rho_0 \exp[-h/h_r], & b_1 &= -0.61592 \times 10^{-2}, \\
\rho_0 &= 0.002378, & b_2 &= 0.621408 \times 10^{-3}, \\
h_r &= 23800, & q_r &= 17700 \sqrt{\rho} (0.0001 v)^{3.07}, \\
c_L &= a_0 + a_1 \hat{\alpha}, & q_a &= c_0 + c_1 \hat{\alpha} + c_2 \hat{\alpha}^2 + c_3 \hat{\alpha}^3, \\
c_D &= b_0 + b_1 \hat{\alpha} + b_2 \hat{\alpha}^2, & c_0 &= 1.0672181, \\
\hat{\alpha} &= 180 \alpha / \pi, & c_1 &= -0.19213774 \times 10^{-1}, \\
R_e &= 20902900, & c_2 &= 0.21286289 \times 10^{-3}, \\
S &= 2690, & c_3 &= -0.10117249 \times 10^{-5}.
\end{aligned}
$$

The reentry trajectory begins at an altitude where the aerodynamic forces are quite small with a weight of $w = 203000$ (lb) and mass $m = w/g_0$ (slug), where $g_0 = 32.174$ (ft/sec^2). The initial conditions are as follows:

$$
\begin{aligned}
h &= 260000 \text{ ft}, & v &= 25600 \text{ ft/sec}, \\
\phi &= 0 \text{ deg}, & \gamma &= -1 \text{ deg}, \\
\theta &= 0 \text{ deg}, & \psi &= 90 \text{ deg}.
\end{aligned}
$$

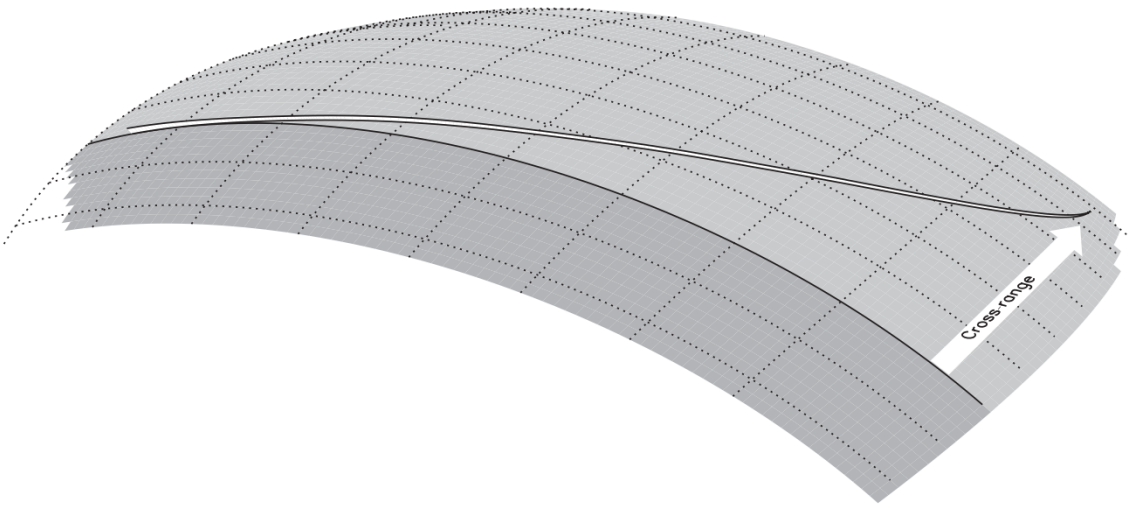


Figure 5.1: Max crossrange shuttle reentry

The final point on the reentry trajectory occurs at the unknown (free) time t_F, at the so-called terminal area energy management (TAEM) interface, which is defined by the conditions

$$h = 80000 \text{ ft}, \qquad v = 2500 \text{ ft/sec}, \qquad \gamma = -5 \text{ deg}.$$

As explained in the book, our goal is to maximize the final crossrange, which is equivalent to maximizing the final latitude of the vehicle, i.e., $J = \theta(t_F)$.

Approach

We will use a discretized model of time, with a fixed number of discretized points, n. The decision variables at each point are going to be the state of the vehicle and the controls commanded to it. In addition, we will also make each time step size Δt a decision variable; that way, we can either fix the time step size easily, or allow the solver to fine-tune the duration between each adjacent pair of points. Finally, in order to approximate the derivatives of the problem dynamics, we will use either rectangular or trapezoidal integration.

> **Warning**
>
> Do not try to actually land a Space Shuttle using this notebook! There's no mesh refinement going on, which can lead to unrealistic trajectories having position and velocity errors with orders of magnitude 10^4 ft and 10^2 ft/sec, respectively.

```
using JuMP
import Interpolations
import Ipopt

# Global variables
const w = 203000.0  # weight (lb)
const g₀ = 32.174    # acceleration (ft/sec^2)
const m = w / g₀    # mass (slug)
```

```
# Aerodynamic and atmospheric forces on the vehicle
const ρ₀ = 0.002378
const hᵣ = 23800.0
const Rₑ = 20902900.0
const μ = 0.14076539e17
const S = 2690.0
const a₀ = -0.20704
const a₁ = 0.029244
const b₀ = 0.07854
const b₁ = -0.61592e-2
const b₂ = 0.621408e-3
const c₀ = 1.0672181
const c₁ = -0.19213774e-1
const c₂ = 0.21286289e-3
const c₃ = -0.10117249e-5

# Initial conditions
const h_s = 2.6          # altitude (ft) / 1e5
const ϕ_s = deg2rad(0)   # longitude (rad)
const θ_s = deg2rad(0)   # latitude (rad)
const v_s = 2.56         # velocity (ft/sec) / 1e4
const γ_s = deg2rad(-1)  # flight path angle (rad)
const ψ_s = deg2rad(90)  # azimuth (rad)
const α_s = deg2rad(0)   # angle of attack (rad)
const β_s = deg2rad(0)   # bank angle (rad)
const t_s = 1.00         # time step (sec)

# Final conditions, the so-called Terminal Area Energy Management (TAEM)
const h_t = 0.8          # altitude (ft) / 1e5
const v_t = 0.25         # velocity (ft/sec) / 1e4
const γ_t = deg2rad(-5)  # flight path angle (rad)

# Number of mesh points (knots) to be used
const n = 503

# Integration scheme to be used for the dynamics
const integration_rule = "rectangular"
```

Choose a good linear solver

Picking a good linear solver is **extremely important** to maximize the performance of nonlinear solvers. For the best results, it is advised to experiment different linear solvers.

For example, the linear solver `MA27` is outdated and can be quite slow. `MA57` is a much better alternative, especially for highly-sparse problems (such as trajectory optimization problems).

```
# Uncomment the lines below to pass user options to the solver
user_options = (
# "mu_strategy" => "monotone",
# "linear_solver" => "ma27",
)

# Create JuMP model, using Ipopt as the solver
model = Model(optimizer_with_attributes(Ipopt.Optimizer, user_options...))
```

```
@variables(model, begin
    0 ≤ scaled_h[1:n]                # altitude (ft) / 1e5
    ϕ[1:n]                 # longitude (rad)
    deg2rad(-89) ≤ θ[1:n] ≤ deg2rad(89)  # latitude (rad)
    1e-4 ≤ scaled_v[1:n]                # velocity (ft/sec) / 1e4
    deg2rad(-89) ≤ γ[1:n] ≤ deg2rad(89)  # flight path angle (rad)
    ψ[1:n]                 # azimuth (rad)
    deg2rad(-90) ≤ α[1:n] ≤ deg2rad(90)  # angle of attack (rad)
    deg2rad(-89) ≤ β[1:n] ≤ deg2rad(1)  # bank angle (rad)
    #        3.5 ≤       Δt[1:n] ≤ 4.5          # time step (sec)
    Δt[1:n] == 4.0         # time step (sec)
end)
```

Info

Above you can find two alternatives for the Δt variables.

The first one, `3.5 ≤ Δt[1:n] ≤ 4.5` (currently commented), allows some wiggle room for the solver to adjust the time step size between pairs of mesh points. This is neat because it allows the solver to figure out which parts of the flight require more dense discretization than others. (Remember, the number of discretized points is fixed, and this example does not implement mesh refinement.) However, this makes the problem more complex to solve, and therefore leads to a longer computation time.

The second line, `Δt[1:n] == 4.0`, fixes the duration of every time step to exactly 4.0 seconds. This allows the problem to be solved faster. However, to do this we need to know beforehand that the close-to-optimal total duration of the flight is ~2009 seconds. Therefore, if we split the total duration in slices of 4.0 seconds, we know that we require `n = 503` knots to discretize the whole trajectory.

```
# Fix initial conditions
fix(scaled_h[1], h_s; force = true)
fix(ϕ[1], ϕ_s; force = true)
fix(θ[1], θ_s; force = true)
fix(scaled_v[1], v_s; force = true)
fix(γ[1], γ_s; force = true)
fix(ψ[1], ψ_s; force = true)

# Fix final conditions
fix(scaled_h[n], h_t; force = true)
fix(scaled_v[n], v_t; force = true)
fix(γ[n], γ_t; force = true)

# Initial guess: linear interpolation between boundary conditions
x_s = [h_s, ϕ_s, θ_s, v_s, γ_s, ψ_s, α_s, β_s, t_s]
x_t = [h_t, ϕ_s, θ_s, v_t, γ_t, ψ_s, α_s, β_s, t_s]
interp_linear = Interpolations.LinearInterpolation([1, n], [x_s, x_t])
initial_guess = mapreduce(transpose, vcat, interp_linear.(1:n))
set_start_value.(all_variables(model), vec(initial_guess))

# Functions to restore `h` and `v` to their true scale
@NLexpression(model, h[j = 1:n], scaled_h[j] * 1e5)
@NLexpression(model, v[j = 1:n], scaled_v[j] * 1e4)

# Helper functions
@NLexpression(model, c_L[j = 1:n], a₀ + a₁ * rad2deg(α[j]))
```

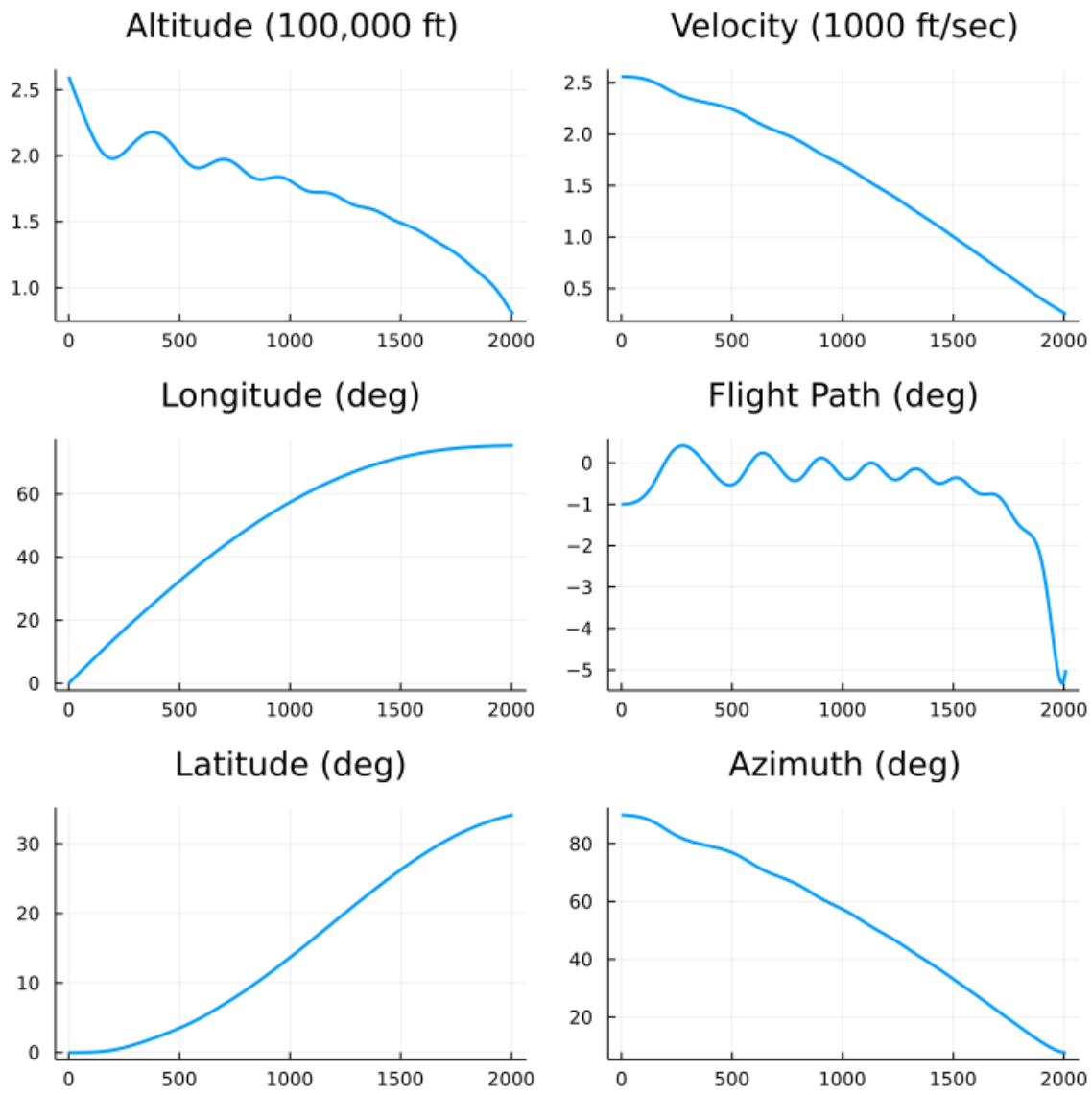
```
@NLexpression(
    model,
    c_D[j = 1:n],
    b₀ + b₁ * rad2deg(α[j]) + b₂ * rad2deg(α[j])^2
)
@NLexpression(model, ρ[j = 1:n], ρ₀ * exp(-h[j] / hᵣ))
@NLexpression(model, D[j = 1:n], 0.5 * c_D[j] * S * ρ[j] * v[j]^2)
@NLexpression(model, L[j = 1:n], 0.5 * c_L[j] * S * ρ[j] * v[j]^2)
@NLexpression(model, r[j = 1:n], Rₑ + h[j])
@NLexpression(model, g[j = 1:n], μ / r[j]^2)

# Motion of the vehicle as a differential-algebraic system of equations (DAEs)
@NLexpression(model, δh[j = 1:n], v[j] * sin(γ[j]))
@NLexpression(
    model,
    δϕ[j = 1:n],
    (v[j] / r[j]) * cos(γ[j]) * sin(ψ[j]) / cos(θ[j])
)
@NLexpression(model, δθ[j = 1:n], (v[j] / r[j]) * cos(γ[j]) * cos(ψ[j]))
@NLexpression(model, δv[j = 1:n], -(D[j] / m) - g[j] * sin(γ[j]))
@NLexpression(
    model,
    δγ[j = 1:n],
    (L[j] / (m * v[j])) * cos(β[j]) +
    cos(γ[j]) * ((v[j] / r[j]) - (g[j] / v[j]))
)
@NLexpression(
    model,
    δψ[j = 1:n],
    (1 / (m * v[j] * cos(γ[j]))) * L[j] * sin(β[j]) +
    (v[j] / (r[j] * cos(θ[j]))) * cos(γ[j]) * sin(ψ[j]) * sin(θ[j])
)

# System dynamics
for j in 2:n
    i = j - 1  # index of previous knot

    if integration_rule == "rectangular"
        # Rectangular integration
        @NLconstraint(model, h[j] == h[i] + Δt[i] * δh[i])
        @NLconstraint(model, ϕ[j] == ϕ[i] + Δt[i] * δϕ[i])
        @NLconstraint(model, θ[j] == θ[i] + Δt[i] * δθ[i])
        @NLconstraint(model, v[j] == v[i] + Δt[i] * δv[i])
        @NLconstraint(model, γ[j] == γ[i] + Δt[i] * δγ[i])
        @NLconstraint(model, ψ[j] == ψ[i] + Δt[i] * δψ[i])
    elseif integration_rule == "trapezoidal"
        # Trapezoidal integration
        @NLconstraint(model, h[j] == h[i] + 0.5 * Δt[i] * (δh[j] + δh[i]))
        @NLconstraint(model, ϕ[j] == ϕ[i] + 0.5 * Δt[i] * (δϕ[j] + δϕ[i]))
        @NLconstraint(model, θ[j] == θ[i] + 0.5 * Δt[i] * (δθ[j] + δθ[i]))
        @NLconstraint(model, v[j] == v[i] + 0.5 * Δt[i] * (δv[j] + δv[i]))
        @NLconstraint(model, γ[j] == γ[i] + 0.5 * Δt[i] * (δγ[j] + δγ[i]))
        @NLconstraint(model, ψ[j] == ψ[i] + 0.5 * Δt[i] * (δψ[j] + δψ[i]))
    else
        @error "Unexpected integration rule '$(integration_rule)'"
    end
```

```
end

# Objective: Maximize crossrange
@objective(model, Max, θ[n])

set_silent(model)  # Hide solver's verbose output
optimize!(model)  # Solve for the control and state
@assert termination_status(model) == LOCALLY_SOLVED

# Show final crossrange of the solution
println(
    "Final latitude θ = ",
    round(objective_value(model) |> rad2deg, digits = 2),
    "°",
)
```

```
Final latitude θ = 34.18°
```

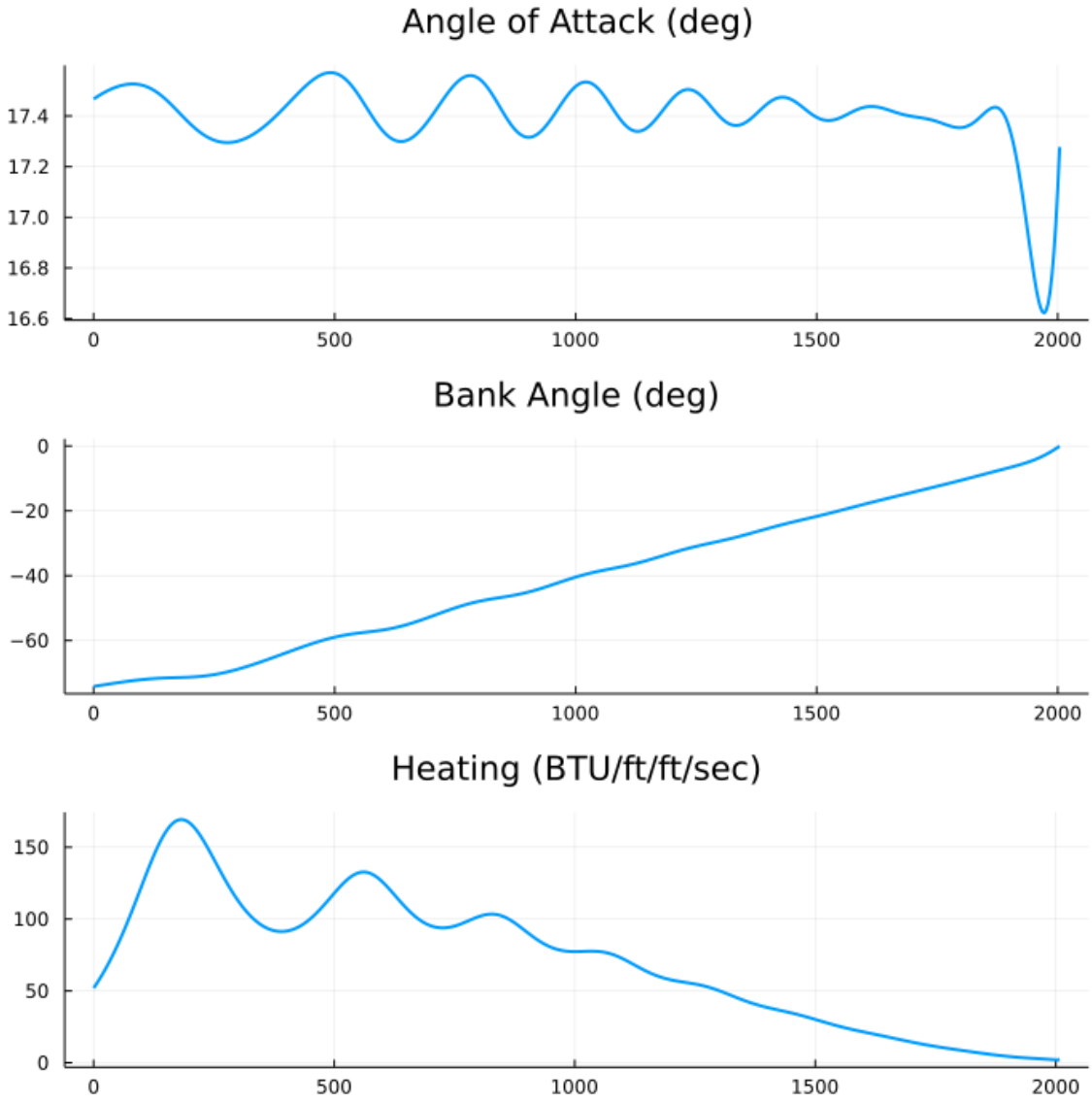
Plotting the results

```
using Plots

ts = cumsum([0; value.(Δt)])[1:end-1]

plt_altitude = plot(
    ts,
    value.(scaled_h),
    legend = nothing,
    title = "Altitude (100,000 ft)",
)
plt_longitude =
    plot(ts, rad2deg.(value.(ϕ)), legend = nothing, title = "Longitude (deg)")
plt_latitude =
    plot(ts, rad2deg.(value.(θ)), legend = nothing, title = "Latitude (deg)")
plt_velocity = plot(
    ts,
    value.(scaled_v),
    legend = nothing,
    title = "Velocity (1000 ft/sec)",
)
plt_flight_path =
    plot(ts, rad2deg.(value.(γ)), legend = nothing, title = "Flight Path (deg)")
plt_azimuth =
    plot(ts, rad2deg.(value.(ψ)), legend = nothing, title = "Azimuth (deg)")

plt = plot(
    plt_altitude,
    plt_velocity,
    plt_longitude,
    plt_flight_path,
    plt_latitude,
    plt_azimuth,
    layout = grid(3, 2),
    linewidth = 2,
    size = (700, 700),
)
```

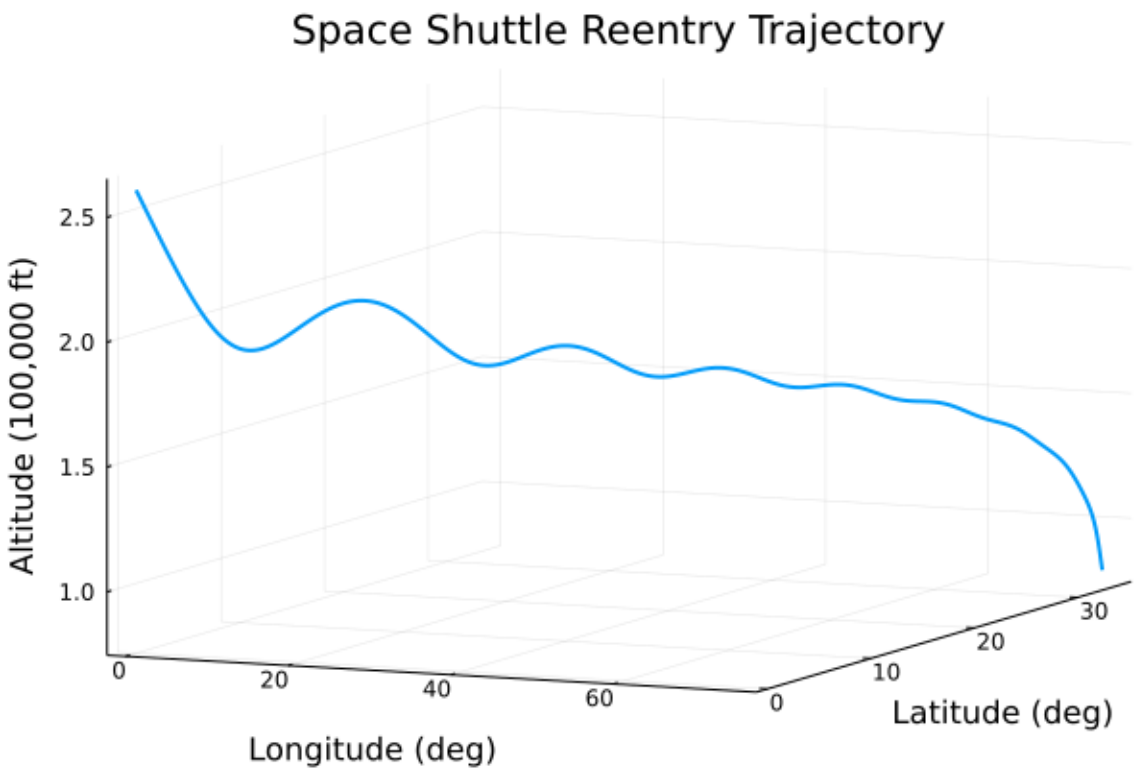



```
function q(h, v, a)
    ρ(h) = ρ₀ * exp(-h / hᵣ)
    qᵣ(h, v) = 17700 * √ρ(h) * (0.0001 * v)^3.07
    qₐ(a) = c₀ + c₁ * rad2deg(a) + c₂ * rad2deg(a)^2 + c₃ * rad2deg(a)^3
    # Aerodynamic heating on the vehicle wing leading edge
    return qₐ(a) * qᵣ(h, v)
end

plt_attack_angle = plot(
    ts[1:end-1],
    rad2deg.(value.(α)[1:end-1]),
    legend = nothing,
    title = "Angle of Attack (deg)",
)
plt_bank_angle = plot(
    ts[1:end-1],
```

```
    rad2deg.(value.(β)[1:end-1]),
    legend = nothing,
    title = "Bank Angle (deg)",
)
plt_heating = plot(
    ts,
    q.(value.(scaled_h) * 1e5, value.(scaled_v) * 1e4, value.(α)),
    legend = nothing,
    title = "Heating (BTU/ft/ft/sec)",
)

plt = plot(
    plt_attack_angle,
    plt_bank_angle,
    plt_heating,
    layout = grid(3, 1),
    linewidth = 2,
    size = (700, 700),
)
```




```
plt = plot(
    rad2deg.(value.(ϕ)),
    rad2deg.(value.(θ)),
    value.(scaled_h),
    linewidth = 2,
    legend = nothing,
    title = "Space Shuttle Reentry Trajectory",
    xlabel = "Longitude (deg)",
    ylabel = "Latitude (deg)",
    zlabel = "Altitude (100,000 ft)",
)
```



Tip

This tutorial was generated using Literate.jl. View the source `.jl` file on GitHub.

5.5 Rocket Control

Originally Contributed by: Iain Dunning

This tutorial shows how to solve a nonlinear rocketry control problem. The problem was drawn from the COPS3 benchmark.

Our goal is to maximize the final altitude of a vertically launched rocket.

We can control the thrust of the rocket, and must take account of the rocket mass, fuel consumption rate, gravity, and aerodynamic drag.

Let us consider the basic description of the model (for the full description, including parameters for the rocket, see the COPS3 PDF)

Overview

We will use a discretized model of time, with a fixed number of time steps, n.

We will make the time step size Δt, and thus the final time $t_f = n \cdot \Delta t$, a variable in the problem. To approximate the derivatives in the problem we will use the trapezoidal rule.

State and Control

We will have three state variables:

- Velocity, v
- Altitude, h
- Mass of rocket and remaining fuel, m

and a single control variable, thrust T.

Our goal is thus to maximize $h(t_f)$.

Each of these corresponds to a JuMP variable indexed by the time step.

Dynamics

We have three equations that control the dynamics of the rocket:

Rate of ascent: $h' = v$ Acceleration: $v' = \frac{T - D(h,v)}{m} - g(h)$ Rate of mass loss: $m' = -\frac{T}{c}$

where drag $D(h, v)$ is a function of altitude and velocity, and gravity $g(h)$ is a function of altitude.

These forces are defined as

$$D(h, v) = D_c v^2 exp\left(-h_c\left(\frac{h - h(0)}{h(0)}\right)\right)$$

and $g(h) = g_0\left(\frac{h(0)}{h}\right)^2$

The three rate equations correspond to JuMP constraints, and for convenience we will represent the forces with nonlinear expressions.

```
using JuMP
import Ipopt
import Plots
```

Create JuMP model, using Ipopt as the solver

```
rocket = Model(Ipopt.Optimizer)
set_silent(rocket)
```

Constants

Note that all parameters in the model have been normalized to be dimensionless. See the COPS3 paper for more info.

```
h_0 = 1    # Initial height
v_0 = 0    # Initial velocity
m_0 = 1    # Initial mass
g_0 = 1    # Gravity at the surface
```

```
T_c = 3.5  # Used for thrust
h_c = 500  # Used for drag
v_c = 620  # Used for drag
m_c = 0.6  # Fraction of initial mass left at end

c = 0.5 * sqrt(g_0 * h_0)  # Thrust-to-fuel mass
m_f = m_c * m_0              # Final mass
D_c = 0.5 * v_c * m_0 / g_0  # Drag scaling
T_max = T_c * g_0 * m_0        # Maximum thrust

n = 800    # Time steps
```

```
800
```

Decision variables

```
@variables(rocket, begin
    Δt ≥ 0, (start = 1 / n) # Time step
    # State variables
    v[1:n] ≥ 0            # Velocity
    h[1:n] ≥ h_0          # Height
    m_f ≤ m[1:n] ≤ m_0    # Mass
    # Control variables
    0 ≤ T[1:n] ≤ T_max    # Thrust
end)
```

Objective

The objective is to maximize altitude at end of time of flight.

```
@objective(rocket, Max, h[n])
```

$$h_{800}$$

Initial conditions

```
fix(v[1], v_0; force = true)
fix(h[1], h_0; force = true)
fix(m[1], m_0; force = true)
fix(m[n], m_f; force = true)
```

Forces

```
@NLexpressions(
    rocket,
    begin
        # Drag(h,v) = Dc v^2 exp( -hc * (h - h0) / h0 )
        drag[j = 1:n], D_c * (v[j]^2) * exp(-h_c * (h[j] - h_0) / h_0)
        # Grav(h)   = go * (h0 / h)^2
        grav[j = 1:n], g_0 * (h_0 / h[j])^2
        # Time of flight
        t_f, Δt * n
    end
)
```

Dynamics

```
for j in 2:n
    # h' = v
    # Rectangular integration
    # @NLconstraint(rocket, h[j] == h[j - 1] + Δt * v[j - 1])
    # Trapezoidal integration
    @NLconstraint(rocket, h[j] == h[j-1] + 0.5 * Δt * (v[j] + v[j-1]))
    # v' = (T-D(h,v))/m - g(h)
    # Rectangular integration
    # @NLconstraint(
    #     rocket,
    #     v[j] == v[j - 1] + Δt *((T[j - 1] - drag[j - 1]) / m[j - 1] - grav[j - 1])
    # )
    # Trapezoidal integration
    @NLconstraint(
        rocket,
        v[j] ==
        v[j-1] +
        0.5 *
        Δt *
        (
            (T[j] - drag[j] - m[j] * grav[j]) / m[j] +
            (T[j-1] - drag[j-1] - m[j-1] * grav[j-1]) / m[j-1]
        )
    )
    # m' = -T/c
    # Rectangular integration
    # @NLconstraint(rocket, m[j] == m[j - 1] - Δt * T[j - 1] / c)
    # Trapezoidal integration
    @NLconstraint(rocket, m[j] == m[j-1] - 0.5 * Δt * (T[j] + T[j-1]) / c)
end
```

Solve for the control and state

```
println("Solving...")
optimize!(rocket)
solution_summary(rocket)
```

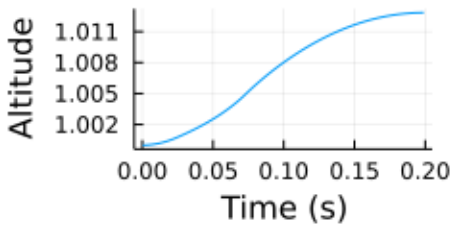
```
* Solver : Ipopt

* Status
  Termination status : LOCALLY_SOLVED
  Primal status      : FEASIBLE_POINT
  Dual status        : FEASIBLE_POINT
  Message from the solver:
  "Solve_Succeeded"

* Candidate solution
  Objective value      : 1.0128340648308019

* Work counters
  Solve time (sec)   : 1.01076
```

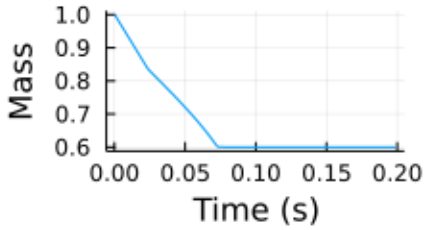
Display results

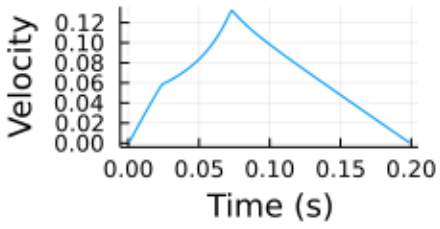
```
println("Max height: ", objective_value(rocket))
```

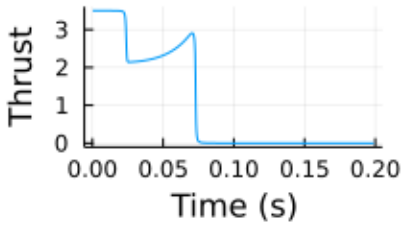
```
Max height: 1.0128340648308019
```

```
function my_plot(y, ylabel)
    return Plots.plot(
        (1:n) * value.(Δt),
        value.(y)[:];
        xlabel = "Time (s)",
        ylabel = ylabel,
    )
end

Plots.plot(
    my_plot(h, "Altitude"),
    my_plot(m, "Mass"),
    my_plot(v, "Velocity"),
    my_plot(T, "Thrust");
    layout = (2, 2),
    legend = false,
    margin = 1Plots.cm,
)
```









Tip

This tutorial was generated using Literate.jl. View the source `.jl` file on GitHub.

5.6 The Rosenbrock function

A nonlinear example of the classical Rosenbrock function.

```
using JuMP
import Ipopt
import Test

function example_rosenbrock()
    model = Model(Ipopt.Optimizer)
    set_silent(model)
    @variable(model, x)
    @variable(model, y)
    @NLobjective(model, Min, (1 - x)^2 + 100 * (y - x^2)^2)
    optimize!(model)

    Test.@test termination_status(model) == LOCALLY_SOLVED
    Test.@test primal_status(model) == FEASIBLE_POINT
    Test.@test objective_value(model) ≈ 0.0 atol = 1e-10
    Test.@test value(x) ≈ 1.0
    Test.@test value(y) ≈ 1.0
    return
end

example_rosenbrock()
```

Tip

This tutorial was generated using Literate.jl. View the source `.jl` file on GitHub.

5.7 Maximum likelihood estimation

Use nonlinear optimization to compute the maximum likelihood estimate (MLE) of the parameters of a normal distribution, a.k.a., the sample mean and variance.

```
using JuMP
import Ipopt
import Random
import Statistics
import Test

function example_mle(; verbose = true)
    n = 1_000
    Random.seed!(1234)
    data = randn(n)
    model = Model(Ipopt.Optimizer)
    set_silent(model)
    @variable(model, μ, start = 0.0)
```

```
    @variable(model, σ >= 0.0, start = 1.0)
    @NLobjective(
        model,
        Max,
        n / 2 * log(1 / (2 * π * σ^2)) -
        sum((data[i] - μ)^2 for i in 1:n) / (2 * σ^2)
    )
    optimize!(model)
    if verbose
        println("μ             = ", value(μ))
        println("mean(data)    = ", Statistics.mean(data))
        println("σ^2           = ", value(σ)^2)
        println("var(data)     = ", Statistics.var(data))
        println("MLE objective = ", objective_value(model))
    end
    Test.@test value(μ) ≈ Statistics.mean(data) atol = 1e-3
    Test.@test value(σ)^2 ≈ Statistics.var(data) atol = 1e-2
    # You can even do constrained MLE!
    @NLconstraint(model, μ == σ^2)
    optimize!(model)
    Test.@test value(μ) ≈ value(σ)^2
    if verbose
        println()
        println("With constraint μ == σ^2:")
        println("μ                          = ", value(μ))
        println("σ^2                        = ", value(σ)^2)
        println("Constrained MLE objective = ", objective_value(model))
    end
    return
end

example_mle()
```

```
μ
             = -0.022943864572027958
mean(data)    = -0.022943864572027916σ
^2           = 1.0096978289461431
var(data)     = 1.0107085374810936
MLE objective = -1423.76408661786

With constraint μ == σ^2:μ
                          = 0.6225971004178991σ
^2                        = 0.6225971004178991
Constrained MLE objective = -1827.5516590930729
```

Tip

This tutorial was generated using Literate.jl. View the source `.jl` file on GitHub.

5.8 The clnlbeam problem

Based on an AMPL model by Hande Y. Benson

Source: H. Maurer and H.D. Mittelman, "The non-linear beam via optimal control with bound state variables", Optimal Control Applications and Methods 12, pp. 19-31, 1991.

```
using JuMP
import Ipopt

function example_clnlbeam()
    N = 1000
    h = 1 / N
    alpha = 350
    model = Model(Ipopt.Optimizer)
    @variables(model, begin
        -1 <= t[1:(N+1)] <= 1
        -0.05 <= x[1:(N+1)] <= 0.05
        u[1:(N+1)]
    end)
    @NLobjective(
        model,
        Min,
        sum(
            0.5 * h * (u[i+1]^2 + u[i]^2) +
            0.5 * alpha * h * (cos(t[i+1]) + cos(t[i])) for i in 1:N
        ),
    )
    @NLconstraint(
        model,
        [i = 1:N],
        x[i+1] - x[i] - 0.5 * h * (sin(t[i+1]) + sin(t[i])) == 0,
    )
    @constraint(
        model,
        [i = 1:N],
        t[i+1] - t[i] - 0.5 * h * u[i+1] - 0.5 * h * u[i] == 0,
    )
    optimize!(model)
    println("""
    termination_status = $(termination_status(model))
    primal_status      = $(primal_status(model))
    objective_value    = $(objective_value(model))
    """)
    return
end

example_clnlbeam()
```

```
This is Ipopt version 3.13.4, running with linear solver mumps.
NOTE: Other linear solvers might be more efficient (see Ipopt documentation).

Number of nonzeros in equality constraint Jacobian...:     8000
Number of nonzeros in inequality constraint Jacobian.:        0
Number of nonzeros in Lagrangian Hessian.............:     4002
```

```
Total number of variables............................:     3003
                     variables with only lower bounds:        0
                variables with lower and upper bounds:     2002
                     variables with only upper bounds:        0
Total number of equality constraints.................:     2000
Total number of inequality constraints...............:        0
        inequality constraints with only lower bounds:        0
   inequality constraints with lower and upper bounds:        0
        inequality constraints with only upper bounds:        0

iter    objective    inf_pr   inf_du lg(mu)  ||d||  lg(rg) alpha_du alpha_pr  ls
   0  3.5000000e+02 0.00e+00 0.00e+00  -1.0 0.00e+00    -  0.00e+00 0.00e+00   0
   1  3.5000000e+02 0.00e+00 0.00e+00  -1.7 0.00e+00    -  1.00e+00 1.00e+00   0
   2  3.5000000e+02 0.00e+00 0.00e+00  -3.8 0.00e+00  -2.0 1.00e+00 1.00e+00T  0
   3  3.5000000e+02 0.00e+00 0.00e+00  -5.7 0.00e+00   0.2 1.00e+00 1.00e+00T  0
   4  3.5000000e+02 0.00e+00 0.00e+00  -8.6 0.00e+00  -0.2 1.00e+00 1.00e+00T  0

Number of Iterations....: 4

                                   (scaled)                 (unscaled)
Objective...............:   3.5000000000000318e+02    3.5000000000000318e+02
Dual infeasibility......:   0.0000000000000000e+00    0.0000000000000000e+00
Constraint violation....:   0.0000000000000000e+00    0.0000000000000000e+00
Complementarity.........:   2.5059035596802450e-09    2.5059035596802450e-09
Overall NLP error.......:   2.5059035596802450e-09    2.5059035596802450e-09


Number of objective function evaluations             = 5
Number of objective gradient evaluations             = 5
Number of equality constraint evaluations            = 5
Number of inequality constraint evaluations          = 0
Number of equality constraint Jacobian evaluations   = 5
Number of inequality constraint Jacobian evaluations = 0
Number of Lagrangian Hessian evaluations             = 4
Total CPU secs in IPOPT (w/o function evaluations)   =      0.123
Total CPU secs in NLP function evaluations           =      0.007

EXIT: Optimal Solution Found.
termination_status = LOCALLY_SOLVED
primal_status      = FEASIBLE_POINT
objective_value    = 350.0000000000032
```

Tip

This tutorial was generated using Literate.jl. View the source `.jl` file on GitHub.

Chapter 6

Conic programs

6.1 Tips and Tricks

Originally Contributed by: Arpit Bhatia

This tutorial is aimed at providing a simplistic introduction to conic programming using JuMP.

It uses the following packages:

```
using JuMP
import SCS
import LinearAlgebra
```

> **Tip**
>
> A good resource for learning more about functions which can be modeled using cones is the MOSEK Modeling Cookbook.

What is a cone?

A subset C of a vector space V is a cone if $\forall x \in C$ and positive scalars $\lambda > 0$, the product $\lambda x \in C$.

A cone C is a convex cone if $\lambda x + (1 - \lambda)y \in C$, for any $\lambda \in [0, 1]$, and any $x, y \in C$.

What is a conic program?

Conic programming problems are convex optimization problems in which a convex function is minimized over the intersection of an affine subspace and a convex cone. An example of a conic-form minimization problems, in the primal form is:

$$\begin{aligned} \min_{x \in \mathbb{R}^n} \quad & a_0^T x + b_0 \\ \text{s.t.} \quad & A_i x + b_i \in \mathcal{C}_i \quad i = 1 \dots m \end{aligned}$$

The corresponding dual problem is:

$$\begin{aligned} \max_{y_1,\ldots,y_m} \quad & -\sum_{i=1}^{m} b_i^T y_i + b_0 \\ \text{s.t.} \quad & a_0 - \sum_{i=1}^{m} A_i^T y_i = 0 \\ & y_i \in \mathcal{C}_i^* \quad i = 1 \ldots m \end{aligned}$$

where each $\mathcal{C}_i$ is a closed convex cone and $\mathcal{C}_i^*$ is its dual cone.

Second-Order Cone

The Second-Order Cone (or Lorentz Cone) of dimension n is of the form:

$$Q^n = \{(t, x) \in \mathbb{R}^n : t \geq ||x||_2\}$$

Example

Minimize the L2 norm of a vector x.

```
model = Model()
@variable(model, x[1:3])
@variable(model, norm_x)
@constraint(model, [norm_x; x] in SecondOrderCone())
@objective(model, Min, norm_x)
```

$$norm\_x$$

Rotated Second-Order Cone

A Second-Order Cone rotated by $\pi/4$ in the (x_1, x_2) plane is called a Rotated Second-Order Cone. It is of the form:

$$Q_r^n = \{(t, u, x) \in \mathbb{R}^n : 2tu \geq ||x||_2^2, t, u \geq 0\}$$

Example

Given a set of predictors x, and observations y, find the parameter θ that minimizes the sum of squares loss between y_i and θx_i.

```
x = [1.0, 2.0, 3.0, 4.0]
y = [0.45, 1.04, 1.51, 1.97]
model = Model()
@variable(model, θ)
@variable(model, loss)
@constraint(model, [loss; 0.5; θ .* x .- y] in RotatedSecondOrderCone())
@objective(model, Min, loss)
```

$$loss$$

Exponential Cone

An Exponential Cone is a set of the form:

$$K_{exp} = \{(x, y, z) \in \mathbb{R}^3 : y \exp(x/y) \leq z, y > 0\}$$

```
model = Model()
@variable(model, x[1:3] >= 0)
@constraint(model, x in MOI.ExponentialCone())
@objective(model, Min, x[3])
```

$$x_3$$

Example: Entropy Maximization

The entropy maximization problem consists of maximizing the entropy function, $H(x) = -x \log x$ subject to linear inequality constraints.

$$\begin{aligned} \max \quad & -\sum_{i=1}^{n} x_i \log x_i \\ \text{s.t.} \quad & \mathbf{1}'x = 1 \\ & Ax \leq b \end{aligned}$$

We can model this problem using an exponential cone by using the following transformation:

$$t \leq -x \log x \iff t \leq x \log(1/x) \iff (t, x, 1) \in K_{exp}$$

Thus, our problem becomes,

$$\begin{aligned} \max \quad & 1^T t \\ \text{s.t.} \quad & Ax \leq b \\ & 1^T x = 1 \\ & (t_i, x_i, 1) \in K_{exp} \quad \forall i = 1 \dots n \end{aligned}$$

```
n = 15
m = 10
A = randn(m, n)
b = rand(m, 1)

model = Model(SCS.Optimizer)
set_silent(model)
@variable(model, t[1:n])
@variable(model, x[1:n])
@objective(model, Max, sum(t))
```

```
@constraint(model, sum(x) == 1)
@constraint(model, A * x .<= b)
@constraint(model, con[i = 1:n], [t[i], x[i], 1] in MOI.ExponentialCone())
optimize!(model)
```

```
objective_value(model)
```

```
2.7080842234725484
```

Positive Semidefinite Cone

The set of positive semidefinite matrices (PSD) of dimension n form a cone in $\mathbb{R}^n$. We write this set mathematically as:

$$\mathcal{S}_+^n = \{X \in \mathcal{S}^n \mid z^T X z \geq 0, \forall z \in \mathbb{R}^n\}.$$

A PSD cone is represented in JuMP using the MOI sets `PositiveSemidefiniteConeTriangle` (for upper triangle of a PSD matrix) and `PositiveSemidefiniteConeSquare` (for a complete PSD matrix). However, it is preferable to use the PSDCone shortcut as illustrated below.

Example: largest eigenvalue of a symmetric matrix Suppose A has eigenvalues $\lambda_1 \geq \lambda_2 \ldots \geq \lambda_n$. Then the matrix $tI - A$ has eigenvalues $t - \lambda_1, t - \lambda_2, \ldots, t - \lambda_n$. Note that $tI - A$ is PSD exactly when all these eigenvalues are non-negative, and this happens for values $t \geq \lambda_1$. Thus, we can model the problem of finding the largest eigenvalue of a symmetric matrix as:

$$\begin{aligned} \lambda_1 = \min\, & t \\ \text{s.t. } & tI - A \succeq 0 \end{aligned}$$

```
A = [3 2 4; 2 0 2; 4 2 3]
I = Matrix{Float64}(LinearAlgebra.I, 3, 3)
model = Model(SCS.Optimizer)
set_silent(model)
@variable(model, t)
@objective(model, Min, t)
@constraint(model, t .* I - A in PSDCone())

optimize!(model)
```

```
objective_value(model)
```

```
8.000000000891703
```

Other Cones and Functions

For other cones supported by JuMP, check out the MathOptInterface Manual.

Tip

This tutorial was generated using Literate.jl. View the source `.jl` file on GitHub.

6.2 Logistic Regression

Originally Contributed by: François Pacaud

This tutorial shows how to solve a logistic regression problem with JuMP. Logistic regression is a well known method in machine learning, useful when we want to classify binary variables with the help of a given set of features. To this goal, we find the optimal combination of features maximizing the (log)-likelihood onto a training set. From a modern optimization glance, the resulting problem is convex and differentiable. On a modern optimization glance, it is even conic representable.

Formulating the logistic regression problem

Suppose we have a set of training data-point $i = 1, \cdots, n$, where for each i we have a vector of features $x_i \in \mathbb{R}^p$ and a categorical observation $y_i \in \{-1, 1\}$.

The log-likelihood is given by

$$l(\theta) = \sum_{i=1}^{n} \log(\frac{1}{1 + \exp(-y_i \theta^\top x_i)})$$

and the optimal θ minimizes the logistic loss function:

$$\min_{\theta} \sum_{i=1}^{n} \log(1 + \exp(-y_i \theta^\top x_i)).$$

Most of the time, instead of solving directly the previous optimization problem, we prefer to add a regularization term:

$$\min_{\theta} \sum_{i=1}^{n} \log(1 + \exp(-y_i \theta^\top x_i)) + \lambda \|\theta\|$$

with $\lambda \in \mathbb{R}_+$ a penalty and $\|.\|$ a norm function. By adding such a regularization term, we avoid overfitting on the training set and usually achieve a greater score in cross-validation.

Reformulation as a conic optimization problem

By introducing auxiliary variables $t_1, \cdots, t_n$ and r, the optimization problem is equivalent to

$$\begin{aligned} \min_{t,r,\theta} \quad & \sum_{i=1}^{n} t_i + \lambda r \\ \text{subject to} \quad & t_i \geq \log(1 + \exp(-y_i \theta^\top x_i)) \\ & r \geq \|\theta\| \end{aligned}$$

Now, the trick is to reformulate the constraints $t_i \geq \log(1 + \exp(-y_i \theta^\top x_i))$ with the help of the exponential cone

$$K_{exp} = \{(x, y, z) \in \mathbb{R}^3 : \ y \exp(x/y) \leq z\}.$$

Indeed, by passing to the exponential, we see that for all $i = 1, \cdots, n$, the constraint $t_i \geq \log(1+\exp(-y_i \theta^\top x_i))$ is equivalent to

$$\exp(-t_i) + \exp(u_i - t_i) \leq 1$$

with $u_i = -y_i \theta^\top x_i$. Then, by adding two auxiliary variables z_{i1} and z_{i2} such that $z_{i1} \geq \exp(u_i - t_i)$ and $z_{i2} \geq \exp(-t_i)$, we get the equivalent formulation

$$\begin{cases} (u_i - t_i, 1, z_{i1}) \in K_{exp} \\ (-t_i, 1, z_{i2}) \in K_{exp} \\ z_{i1} + z_{i2} \leq 1 \end{cases}$$

In this setting, the conic version of the logistic regression problems writes out

$$\begin{aligned} \min_{t,z,r,\theta} \ & \sum_{i=1}^{n} t_i + \lambda r \\ \text{subject to} \quad & (u_i - t_i, 1, z_{i1}) \in K_{exp} \\ & (-t_i, 1, z_{i2}) \in K_{exp} \\ & z_{i1} + z_{i2} \leq 1 \\ & u_i = -y_i x_i^\top \theta \\ & r \geq \|\theta\| \end{aligned}$$

and thus encompasses $3n + p + 1$ variables and $3n + 1$ constraints ($u_i = -y_i \theta^\top x_i$ is only a virtual constraint used to clarify the notation). Thus, if $n \gg 1$, we get a large number of variables and constraints.

Fitting logistic regression with a conic solver

It is now time to pass to the implementation. We choose SCS as a conic solver.

```
using JuMP
import Random
import SCS

Random.seed!(2713);
```

We start by implementing a function to generate a fake dataset, and where we could tune the correlation between the feature variables. The function is a direct transcription of the one used in this blog post.

```
function generate_dataset(n_samples = 100, n_features = 10; shift = 0.0)
    X = randn(n_samples, n_features)
    w = randn(n_features)
```

```
    y = sign.(X * w)
    X .+= 0.8 * randn(n_samples, n_features) # add noise
    X .+= shift # shift the points in the feature space
    X = hcat(X, ones(n_samples, 1))
    return X, y
end
```

```
generate_dataset (generic function with 3 methods)
```

We write a `softplus` function to formulate each constraint $t \geq \log(1 + \exp(u))$ with two exponential cones.

```
function softplus(model, t, u)
    z = @variable(model, [1:2], lower_bound = 0.0)
    @constraint(model, sum(z) <= 1.0)
    @constraint(model, [u - t, 1, z[1]] in MOI.ExponentialCone())
    @constraint(model, [-t, 1, z[2]] in MOI.ExponentialCone())
end
```

```
softplus (generic function with 1 method)
```

ℓ_2 regularized logistic regression

Then, with the help of the `softplus` function, we could write our optimization model. In the ℓ_2 regularization case, the constraint $r \geq \|\theta\|_2$ rewrites as a second order cone constraint.

```
function build_logit_model(X, y, λ)
    n, p = size(X)
    model = Model()
    @variable(model, θ[1:p])
    @variable(model, t[1:n])
    for i in 1:n
        u = -(X[i, :]' * θ) * y[i]
        softplus(model, t[i], u)
    end
    # Add 2 regularization
    @variable(model, 0.0 <= reg)
    @constraint(model, [reg; θ] in MOI.SecondOrderCone(p + 1))
    # Define objective
    @objective(model, Min, sum(t) + λ * reg)
    return model
end
```

```
build_logit_model (generic function with 1 method)
```

We generate the dataset.

> **Warning**
>
> Be careful here, for large n and p SCS could fail to converge!

```
n, p = 200, 10
X, y = generate_dataset(n, p, shift = 10.0);
```

```
# We could now solve the logistic regression problem
λ = 10.0
model = build_logit_model(X, y, λ)
set_optimizer(model, SCS.Optimizer)
set_silent(model)
JuMP.optimize!(model)
```

```
θ♯ = JuMP.value.(model[:θ])
```

```
11-element Vector{Float64}:
  0.0015739398744061455
  0.6238311180575344
 -0.36068059716387135
  0.16711699701725066
  0.24900929787887122
 -0.49972930044500624
 -0.46482760905401316
  0.4218953014106394
 -0.14975373461583785
  0.027573785297299116
 -0.12513807956813522
```

It appears that the speed of convergence is not that impacted by the correlation of the dataset, nor by the penalty λ.

ℓ_1 regularized logistic regression

We now formulate the logistic problem with a ℓ_1 regularization term. The ℓ_1 regularization ensures sparsity in the optimal solution of the resulting optimization problem. Luckily, the ℓ_1 norm is implemented as a set in `MathOptInterface`. Thus, we could easily formulate the sparse logistic regression problem with the help of a `MOI.NormOneCone` set.

```
function build_sparse_logit_model(X, y, λ)
    n, p = size(X)
    model = Model()
    @variable(model, θ[1:p])
    @variable(model, t[1:n])
    for i in 1:n
        u = -(X[i, :]' * θ) * y[i]
        softplus(model, t[i], u)
    end
    # Add 1 regularization
    @variable(model, 0.0 <= reg)
    @constraint(model, [reg; θ] in MOI.NormOneCone(p + 1))
    # Define objective
    @objective(model, Min, sum(t) + λ * reg)
    return model
end

# Auxiliary function to count non-null components:
count_nonzero(v::Vector; tol = 1e-6) = sum(abs.(v) .>= tol)

# We solve the sparse logistic regression problem on the same dataset as before.
λ = 10.0
sparse_model = build_sparse_logit_model(X, y, λ)
```

```
set_optimizer(sparse_model, SCS.Optimizer)
set_silent(sparse_model)
JuMP.optimize!(sparse_model)
```

```
θ♯ = JuMP.value.(sparse_model[:θ])
println(
    "Number of non-zero components: ",
    count_nonzero(θ♯),
    " (out of ",
    p,
    " features)",
)
```

```
Number of non-zero components: 9 (out of 10 features)
```

Extensions

A direct extension would be to consider the sparse logistic regression with hard thresholding, which, on contrary to the soft version using a ℓ_1 regularization, adds an explicit cardinality constraint in its formulation:

$$\begin{aligned} \min_{\theta} \quad & \sum_{i=1}^{n} \log(1 + \exp(-y_i \theta^\top x_i)) + \lambda \|\theta\|_2^2 \\ \text{subject to} \quad & \|\theta\|_0 <= k \end{aligned}$$

where k is the maximum number of non-zero components in the vector θ, and $\|.\|_0$ is the ℓ_0 pseudo-norm:

$$\|x\|_0 = \#\{i : \ x_i \neq 0\}$$

The cardinality constraint $\|\theta\|_0 \leq k$ could be reformulated with binary variables. Thus the hard sparse regression problem could be solved by any solver supporting mixed integer conic problems.

> **Tip**
>
> This tutorial was generated using Literate.jl. View the source `.jl` file on GitHub.

6.3 K-means clustering via SDP

From "Approximating K-means-type clustering via semidefinite programming" By Jiming Peng and Yu Wei.

Given a set of points $a_1, \ldots, a_m$ in R_n, allocate them to k clusters.

```
using JuMP
import LinearAlgebra
import SCS

function example_cluster(; verbose = true)
    # Data points
```

```
    n = 2
    m = 6
    a = Any[
        [2.0, 2.0],
        [2.5, 2.1],
        [7.0, 7.0],
        [2.2, 2.3],
        [6.8, 7.0],
        [7.2, 7.5],
    ]
    k = 2
    # Weight matrix
    W = zeros(m, m)
    for i in 1:m
        for j in i+1:m
            W[i, j] = W[j, i] = exp(-LinearAlgebra.norm(a[i] - a[j]) / 1.0)
        end
    end
    model = Model(SCS.Optimizer)
    set_silent(model)
    # Z >= 0, PSD
    @variable(model, Z[1:m, 1:m], PSD)
    @constraint(model, Z .>= 0)
    # min Tr(W(I-Z))
    I = Matrix(1.0 * LinearAlgebra.I, m, m)
    @objective(model, Min, LinearAlgebra.tr(W * (I - Z)))
    # Z e = e
    @constraint(model, Z * ones(m) .== ones(m))
    # Tr(Z) = k
    @constraint(model, LinearAlgebra.tr(Z) == k)
    optimize!(model)
    Z_val = value.(Z)
    # A simple rounding scheme
    which_cluster = zeros(Int, m)
    num_clusters = 0
    for i in 1:m
        if Z_val[i, i] <= 1e-3
            continue
        elseif which_cluster[i] == 0
            num_clusters += 1
            which_cluster[i] = num_clusters
            for j in i+1:m
                if LinearAlgebra.norm(Z_val[i, j] - Z_val[i, i]) <= 1e-3
                    which_cluster[j] = num_clusters
                end
            end
        end
    end
    if verbose
        # Print results
        for cluster in 1:k
            println("Cluster $cluster")
            for i in 1:m
                if which_cluster[i] == cluster
                    println(a[i])
                end
```

```
            end
        end
    end
    return
end

example_cluster()
```

```
Cluster 1
[2.0, 2.0]
[2.5, 2.1]
[2.2, 2.3]
Cluster 2
[7.0, 7.0]
[6.8, 7.0]
[7.2, 7.5]
```

Tip

This tutorial was generated using Literate.jl. View the source `.jl` file on GitHub.

6.4 The correlation problem

Given three random variables A, B, C and given bounds on two of the three correlation coefficients:

```
-0.2 <= ρ_AB <= -0.1
0.4 <= ρ_BC <=  0.5
```

We can use the following property of the correlations to determine bounds on ρ_AC by solving a SDP:

```
|  1    ρ_AB  ρ_AC |
| ρ_AB   1    ρ_BC |  ⪰ 0
| ρ_AC  ρ_BC   1   |
```

```
using JuMP
import SCS

function example_corr_sdp()
    model = Model(SCS.Optimizer)
    set_silent(model)
    @variable(model, X[1:3, 1:3], PSD)
    # Diagonal is 1s
    @constraint(model, X[1, 1] == 1)
    @constraint(model, X[2, 2] == 1)
    @constraint(model, X[3, 3] == 1)
    # Bounds on the known correlations
    @constraint(model, X[1, 2] >= -0.2)
    @constraint(model, X[1, 2] <= -0.1)
    @constraint(model, X[2, 3] >= 0.4)
    @constraint(model, X[2, 3] <= 0.5)
    # Find upper bound
    @objective(model, Max, X[1, 3])
    optimize!(model)
```

```
    println("An upper bound for X[1, 3] is $(value(X[1, 3]))")
    # Find lower bound
    @objective(model, Min, X[1, 3])
    optimize!(model)
    println("A lower bound for X[1, 3] is $(value(X[1, 3]))")
    return
end

example_corr_sdp()
```

```
An upper bound for X[1, 3] is 0.8719210492775256
A lower bound for X[1, 3] is -0.9779977729240896
```

Tip

This tutorial was generated using Literate.jl. View the source `.jl` file on GitHub.

6.5 Experiment Design

Originally Contributed by: Arpit Bhatia, Chris Coey

This tutorial covers experiment design examples (D-optimal, A-optimal, and E-optimal) from section 7.5 of the book Convex Optimization by Boyd and Vandenberghe.

The tutorial uses the following packages

```
using JuMP
import SCS
import LinearAlgebra
import Random
```

We set a seed so the random numbers are repeatable:

```
Random.seed!(1234)
```

```
MersenneTwister(1234)
```

Relaxed Experiment Design Problem

The basic experiment design problem is as follows.

Given the menu of possible choices for experiments, $v_1, \ldots, v_p$, and the total number m of experiments to be carried out, choose the numbers of each type of experiment, i.e., $m_1, \ldots, m_p$ to make the error covariance E small (in some sense).

The variables $m_1, \ldots, m_p$ must, of course, be integers and sum to m the given total number of experiments. This leads to the optimization problem:

$$\begin{aligned}\min\left(\text{w.r.t.}\mathbf{S}_+^n\right) E = &\left(\sum_{j=1}^p m_j v_j v_j^T\right)^{-1}\\ \text{subject to} & m_i \geq 0\\ & \sum_{i=1}^p m_i = m\\ & m_i \in \mathbb{Z}, \quad i = 1, \ldots, p\end{aligned}$$

The basic experiment design problem can be a hard combinatorial problem when m, the total number of experiments, is comparable to n, since in this case the m_i are all small integers.

In the case when m is large compared to n, however, a good approximate solution can be found by ignoring, or relaxing, the constraint that the m_i are integers.

Let $\lambda_i = m_i/m$, which is the fraction of the total number of experiments for which $a_j = v_i$, or the relative frequency of experiment i. We can express the error covariance in terms of λ_i as:

$$E = \frac{1}{m}\left(\sum_{i=1}^p \lambda_i v_i v_i^T\right)^{-1}$$

The vector $\lambda \in \mathbf{R}^p$ satisfies $\lambda \succeq 0, \mathbf{1}^T\lambda = 1$, and also, each λ_i is an integer multiple of $1/m$. By ignoring this last constraint, we arrive at the problem:

$$\begin{aligned}\min\left(\text{w.r.t.}\mathbf{S}_+^n\right) E = &(1/m)\left(\sum_{i=1}^p \lambda_i v_i v_i^T\right)^{-1}\\ \text{subject to:} \quad & \lambda \succeq 0\\ & \mathbf{1}^T\lambda = 1\end{aligned}$$

Several scalarizations have been proposed for the experiment design problem, which is a vector optimization problem over the positive semidefinite cone.

```
q = 4 # dimension of estimate space
p = 8 # number of experimental vectors
nmax = 3 # upper bound on lambda
n = 12

V = randn(q, p)

eye = Matrix{Float64}(LinearAlgebra.I, q, q);
```

A-optimal design

In A-optimal experiment design, we minimize tr E, the trace of the covariance matrix. This objective is simply the mean of the norm of the error squared:

$$\mathbf{E}\|e\|_2^2 = \mathbf{E}\,\mathrm{tr}\left(ee^T\right) = \mathrm{tr}\,E$$

The A-optimal experiment design problem in SDP form is

$$\begin{aligned} &\min \mathbf{1}^T u \\ &\text{subject to} \begin{bmatrix} \sum_{i=1}^{p} \lambda_i v_i v_i^T e_k & \\ & e_k^T u_k \end{bmatrix} \succeq 0, \quad k = 1, \ldots, n \\ &\lambda \succeq 0 \\ &\mathbf{1}^T \lambda = 1 \end{aligned}$$

```
aOpt = Model(SCS.Optimizer)
set_silent(aOpt)
@variable(aOpt, np[1:p], lower_bound = 0, upper_bound = nmax)
@variable(aOpt, u[1:q], lower_bound = 0)
@constraint(aOpt, sum(np) <= n)
for i in 1:q
    matrix = [
        V*LinearAlgebra.diagm(0 => np ./ n)*V' eye[:, i]
        eye[i, :]' u[i]
    ]
    @constraint(aOpt, matrix >= 0, PSDCone())
end
@objective(aOpt, Min, sum(u))
optimize!(aOpt)
objective_value(aOpt)
```

```
5.041251236819586
```

```
value.(np)
```

```
8-element Vector{Float64}:
  1.7479403177344883
  1.1153179223668486
 -1.922234258858022e-6
  1.661952880720603
  3.0000000604448136
  0.8414252139364004
  1.3825638732306254
  2.25080085111483
```

E-optimal design

In E -optimal design, we minimize the norm of the error covariance matrix, i.e. the maximum eigenvalue of E.

Since the diameter (twice the longest semi-axis) of the confidence ellipsoid $\mathcal{E}$ is proportional to $\|E\|_2^{1/2}$, minimizing $\|E\|_2$ can be interpreted geometrically as minimizing the diameter of the confidence ellipsoid.

E-optimal design can also be interpreted as minimizing the maximum variance of $q^T e$, over all q with $\|q\|_2 = 1$. The E-optimal experiment design problem in SDP form is:

$$\begin{aligned} &\min t \\ \text{subject to } &\sum_{i=1}^{p} \lambda_i v_i v_i^T \succeq tI \\ &\lambda \succeq 0 \\ &\mathbf{1}^T \lambda = 1 \end{aligned}$$

```
eOpt = Model(SCS.Optimizer)
set_silent(eOpt)
@variable(eOpt, 0 <= np[1:p] <= nmax)
@variable(eOpt, t)
@constraint(
    eOpt,
    V * LinearAlgebra.diagm(0 => np ./ n) * V' - (t .* eye) >= 0,
    PSDCone(),
)
@constraint(eOpt, sum(np) <= n)
@objective(eOpt, Max, t)
optimize!(eOpt)
objective_value(eOpt)
```

```
0.44894117787000026
```

```
value.(np)
```

```
8-element Vector{Float64}:
  2.9999995561186084
  0.6751761426314662
 -1.6786021921057114e-6
  1.0458391996148781
  3.0000003496179763
  1.7870006242470347
  0.30151437460805924
  2.1904663013064627
```

D-optimal design

The most widely used scalarization is called D -optimal design, in which we minimize the determinant of the error covariance matrix E. This corresponds to designing the experiment to minimize the volume of the resulting confidence ellipsoid (for a fixed confidence level). Ignoring the constant factor $1/m$ in E, and taking the logarithm of the objective, we can pose this problem as convex optimization problem:

$$\begin{aligned} &\min \log\det \left(\sum_{i=1}^{p} \lambda_i v_i v_i^T \right)^{-1} \\ \text{subject to} &\lambda \succeq 0 \\ &\mathbf{1}^T \lambda = 1 \end{aligned}$$

```
dOpt = Model(SCS.Optimizer)
set_silent(dOpt)
@variable(dOpt, np[1:p], lower_bound = 0, upper_bound = nmax)
@variable(dOpt, t)
@objective(dOpt, Max, t)
@constraint(dOpt, sum(np) <= n)
E = V * LinearAlgebra.diagm(0 => np ./ n) * V'
@constraint(
    dOpt,
    [t, 1, (E[i, j] for i in 1:q for j in 1:i)...] in MOI.LogDetConeTriangle(q)
)
optimize!(dOpt)
objective_value(dOpt)
```

```
0.19015529059725084
```

```
value.(np)
```

```
8-element Vector{Float64}:
 -5.15168097403274e-7
  2.5674126002773345
 -4.203669608920717e-9
  0.2627487637701729
  2.943555996956865
  2.3925058556597834
  2.836940978143653
  0.9968446352783984
```

Tip

This tutorial was generated using Literate.jl. View the source `.jl` file on GitHub.

6.6 SDP relaxations: max-cut

Solves a semidefinite programming relaxation of the MAXCUT graph problem:

```
max   0.25 * •LX
s.t.  diag(X) == e
      X ⪰ 0
```

Where L is the weighted graph Laplacian. Uses this relaxation to generate a solution to the original MAXCUT problem using the method from the paper:

Goemans, M. X., & Williamson, D. P. (1995). Improved approximation algorithms for maximum cut and satisfiability problems using semidefinite programming. Journal of the ACM (JACM), 42(6), 1115-1145.

```
using JuMP
import LinearAlgebra
import Random
import SCS
import Test
```

```
function solve_max_cut_sdp(num_vertex, weights)
    # Calculate the (weighted) Lapacian of the graph: L = D - W.
    laplacian = LinearAlgebra.diagm(0 => weights * ones(num_vertex)) - weights
    # Solve the SDP relaxation
    model = Model(SCS.Optimizer)
    set_silent(model)
    @variable(model, X[1:num_vertex, 1:num_vertex], PSD)
    @objective(model, Max, 1 / 4 * LinearAlgebra.dot(laplacian, X))
    @constraint(model, LinearAlgebra.diag(X) .== 1)
    optimize!(model)
    # Compute the Cholesky factorization of X, i.e., X = V^T V.
    opt_X = LinearAlgebra.Hermitian(value.(X), :U)  # Tell Julia its PSD.
    factorization = LinearAlgebra.cholesky(opt_X, Val(true); check = false)
    V = (factorization.P * factorization.L)'
    # Normalize columns.
    for i in 1:num_vertex
        V[:, i] ./= LinearAlgebra.norm(V[:, i])
    end
    # Generate random vector on unit sphere.
    Random.seed!(num_vertex)
    r = rand(num_vertex)
    r /= LinearAlgebra.norm(r)
    # Iterate over vertices, and assign each vertex to a side of cut.
    cut = ones(num_vertex)
    for i in 1:num_vertex
        if LinearAlgebra.dot(r, V[:, i]) <= 0
            cut[i] = -1
        end
    end

    return cut, 0.25 * sum(laplacian .* (cut * cut'))
end

function example_max_cut_sdp()
    #   [1] --- 5 --- [2]
    #
    # Solution:
    #  (S, S′)  = ({1}, {2})
    cut, cutval = solve_max_cut_sdp(2, [0.0 5.0; 5.0 0.0])
    Test.@test cut[1] != cut[2]
    #   [1] --- 5 --- [2]
    #    |  \          |
    #    |    \        |
    #    7      6      1
    #    |        \    |
    #    |          \  |
    #   [3] --- 1 --- [4]
    #
    # Solution:
    #  (S, S′)  = ({1}, {2, 3, 4})
    W = [
        0.0 5.0 7.0 6.0
        5.0 0.0 0.0 1.0
        7.0 0.0 0.0 1.0
        6.0 1.0 1.0 0.0
    ]
```

```
    cut, cutval = solve_max_cut_sdp(4, W)
    Test.@test cut[1] != cut[2]
    Test.@test cut[2] == cut[3] == cut[4]
    #   [1] --- 1 --- [2]
    #    |             |
    #    |             |
    #    5             9
    #    |             |
    #    |             |
    #   [3] --- 2 --- [4]
    #
    # Solution:
    #  (S, S′)  = ({1, 4}, {2, 3})
    W = [
        0.0 1.0 5.0 0.0
        1.0 0.0 0.0 9.0
        5.0 0.0 0.0 2.0
        0.0 9.0 2.0 0.0
    ]
    cut, cutval = solve_max_cut_sdp(4, W)
    Test.@test cut[1] == cut[4]
    Test.@test cut[2] == cut[3]
    Test.@test cut[1] != cut[2]
    return
end

example_max_cut_sdp()
```

Tip

This tutorial was generated using Literate.jl. View the source `.jl` file on GitHub.

6.7 The minimum distortion problem

This example arises from computational geometry, in particular the problem of embedding a general finite metric space into a euclidean space.

It is known that the 4-point metric space defined by the star graph:

```
x
 \\
  x — x
 /
x
```

where distances are computed by length of the shortest path between vertices, cannot be exactly embedded into a euclidean space of any dimension.

Here we will formulate and solve an SDP to compute the best possible embedding, that is, the embedding f() that minimizes the distortion c such that

```
(1 / c) * D(a, b) ≤ ||f(a) - f(b)|| ≤ D(a, b)
```

for all points (a, b), where D(a, b) is the distance in the metric space.

Any embedding can be characterized by its Gram matrix Q, which is PSD, and

```
||f(a) - f(b)||^2 = Q[a, a] + Q[b, b] - 2 * Q[a, b]
```

We can therefore constrain

```
D[i, j]^2 ≤ Q[i, i] + Q[j, j] - 2 * Q[i, j] ≤ c^2 * D[i, j]^2
```

and minimize c^2, which gives us the SDP formulation below.

For more detail, see "Lectures on discrete geometry" by J. Matoušek, Springer, 2002, pp. 378-379.

```
using JuMP
import SCS
import Test

function example_min_distortion()
    model = Model(SCS.Optimizer)
    set_silent(model)
    D = [
        0.0 1.0 1.0 1.0
        1.0 0.0 2.0 2.0
        1.0 2.0 0.0 2.0
        1.0 2.0 2.0 0.0
    ]
    @variable(model, c² >= 1.0)
    @variable(model, Q[1:4, 1:4], PSD)
    for i in 1:4
        for j in (i+1):4
            @constraint(model, D[i, j]^2 <= Q[i, i] + Q[j, j] - 2 * Q[i, j])
            @constraint(
                model,
                Q[i, i] + Q[j, j] - 2 * Q[i, j] <= c² * D[i, j]^2
            )
        end
    end
    @objective(model, Min, c²)
    optimize!(model)
    Test.@test termination_status(model) == OPTIMAL
    Test.@test primal_status(model) == FEASIBLE_POINT
    Test.@test objective_value(model) ≈ 4 / 3 atol = 1e-4
    return
end

example_min_distortion()
```

Tip

This tutorial was generated using Literate.jl. View the source `.jl` file on GitHub.

6.8 Minimum ellipses

This example is from the Boyd & Vandenberghe book "Convex Optimization". Given a set of ellipses centered on the origin

```
E(A) = { u | u^T inv(A) u <= 1 }
```

find a "minimal" ellipse that contains the provided ellipses.

We can formulate this as an SDP:

```
minimize    trace(WX)
subject to  X >= A_i,    i = 1,...,m
            X PSD
```

where W is a PD matrix of weights to choose between different solutions.

```
using JuMP
import LinearAlgebra
import SCS
import Test

function example_min_ellipse()
    # We will use three ellipses: two "simple" ones, and a random one.
    As = [
        [2.0 0.0; 0.0 1.0],
        [1.0 0.0; 0.0 3.0],
        [2.86715 1.60645; 1.60645 1.12639],
    ]
    # We change the weights to see different solutions, if they exist
    weights = [1.0 0.0; 0.0 1.0]
    model = Model(SCS.Optimizer)
    set_silent(model)
    @variable(model, X[i = 1:2, j = 1:2], PSD)
    @objective(model, Min, LinearAlgebra.tr(weights * X))
    @constraint(model, [As_i in As], X >= As_i, PSDCone())
    optimize!(model)
    Test.@test termination_status(model) == OPTIMAL
    Test.@test primal_status(model) == FEASIBLE_POINT
    Test.@test objective_value(model) ≈ 6.46233 atol = 1e-5
    Test.@test value.(X) ≈ [3.1651 0.8022; 0.8022 3.2972] atol = 1e-4
    return
end

example_min_ellipse()
```

Tip

This tutorial was generated using Literate.jl. View the source `.jl` file on GitHub.

6.9 Robust uncertainty sets

Computes the Value at Risk for a data-driven uncertainty set; see "Data-Driven Robust Optimization" (Bertsimas 2013), section 6.1 for details. Closed-form expressions for the optimal value are available.

```
using JuMP
import SCS
import LinearAlgebra
import Test

```

```
function example_robust_uncertainty()
    R = 1
    d = 3
     = 0.05
    ε = 0.05
    N = ceil((2 + 2 * log(2 / ))^2) + 1
    c = randn(d)
    μhat = rand(d)
    M = rand(d, d)
    Σhat = 1 / (d - 1) * (M - ones(d) * μhat')' * (M - ones(d) * μhat')
    Γ1(, N) = R / sqrt(N) * (2 + sqrt(2 * log(1 / )))
    Γ2(, N) = 2 * R^2 / sqrt(N) * (2 + sqrt(2 * log(2 / )))
    model = Model(SCS.Optimizer)
    set_silent(model)
    @variable(model, Σ[1:d, 1:d], PSD)
    @variable(model, u[1:d])
    @variable(model, μ[1:d])
    @constraint(model, [Γ1( / 2, N); μ - μhat] in SecondOrderCone())
    @constraint(model, [Γ2( / 2, N); vec(Σ - Σhat)] in SecondOrderCone())
    @constraint(model, [((1-ε)/ε) (u - μ)'; (u-μ) Σ] in PSDCone())
    @objective(model, Max, LinearAlgebra.dot(c, u))
    optimize!(model)
    I = Matrix(1.0 * LinearAlgebra.I, d, d)
    exact =
        LinearAlgebra.dot(μhat, c) +
        Γ1( / 2, N) * LinearAlgebra.norm(c) +
        sqrt((1 - ε) / ε) *
        sqrt(LinearAlgebra.dot(c, (Σhat + Γ2( / 2, N) * I) * c))
    Test.@test objective_value(model) ≈ exact atol = 1e-3
    return
end

example_robust_uncertainty()
```

Tip

This tutorial was generated using Literate.jl. View the source `.jl` file on GitHub.

Chapter 7

Algorithms

7.1 Benders Decomposition

Originally Contributed by: Shuvomoy Das Gupta

This notebook describes how to implement Benders decomposition in JuMP, which is a large scale optimization scheme.

We only discuss the classical approach (using loops) here. The approach using lazy constraints is showed in the corresponding tutorial.

To illustrate an implementation of the Benders decomposition in JuMP, we apply it to the following general mixed integer problem:

$$\begin{aligned} \text{maximize} \quad & c_1^T x + c_2^T v \\ \text{subject to} \quad & A_1 x + A_2 v \preceq b \\ & x \succeq 0, x \in \mathbb{Z}^n \\ & v \succeq 0, v \in \mathbb{R}^p \end{aligned}$$

where $b \in \mathbb{R}^m$, $A_1 \in \mathbb{R}^{m \times n}$, $A_2 \in \mathbb{R}^{m \times p}$ and $\mathbb{Z}$ is the set of integers.

Here the symbol $\succeq$ ($\preceq$) stands for element-wise greater (less) than or equal to. Any mixed integer programming problem can be written in the form above.

We want to write the Benders decomposition algorithm for the problem above. Consider the polyhedron $\{u \in \mathbb{R}^m | A_2^T u \succeq 0, u \succeq 0\}$. Assume the set of vertices and extreme rays of the polyhedron is denoted by P and Q respectively.

Assume on the kth iteration the subset of vertices of the polyhedron mentioned is denoted by $T(k)$ and the subset of extreme rays are denoted by $Q(k)$, which will be generated by the Benders decomposition problem below.

Benders decomposition algorithm

Step 1 (Initialization)

We start with $T(1) = Q(1) = \emptyset$. Let $f_m^{(1)}$ be arbitrarily large and $x^{(1)}$ be any non-negative integer vector and go to Step 2.

Step 2 (Solving the master problem)

Solve the master problem, $f_{\mathsf{m}}^{(k)}$ =

$$\begin{array}{lll} \text{maximize} & t & \\ \text{subject to} & \forall \bar{u} \in T(k) & t + (A_1^T \bar{u} - c_1)^T x \leq b^T \bar{u} \\ & \forall \bar{y} \in Q(k) & (A_1^T \bar{y})^T x \leq b^T \bar{y} \\ & & x \succeq 0, x \in \mathbb{Z}^n \end{array}$$

Let the maximizer corresponding to the objective value $f_{\mathsf{m}}^{(k)}$ be denoted by $x^{(k)}$. Now there are three possibilities:

- If $f_{\mathsf{m}}^{(k)} = -\infty$, i.e., the master problem is infeasible, then the original proble is infeasible and sadly, we are done.
- If $f_{\mathsf{m}}^{(k)} = \infty$, i.e. the master problem is unbounded above, then we take $f_{\mathsf{m}}^{(k)}$ to be arbitrarily large and $x^{(k)}$ to be a corresponding feasible solution. Go to Step 3.
- If $f_{\mathsf{m}}^{(k)}$ is finite, then we collect $t^{(k)}$ and $x^{(k)}$ and go to Step 3.

Step 3 (Solving the subproblem and add Benders cut when needed)

Solve the subproblem, $f_s(x^{(k)})$ =

$$\begin{array}{lll} c_1^T x^{(k)} + \text{minimize} & (b - A_1 x^{(k)})^T u \\ \text{subject to} & A_2^T u \succeq c_2 \\ & u \succeq 0, u \in \mathbb{R}^m \end{array}$$

Let the minimizer corresponding to the objective value $f_s(x^{(k)})$ be denoted by $u^{(k)}$. There are three possibilities:

- If $f_s(x^{(k)}) = \infty$, the original problem is either infeasible or unbounded. We quit from Benders algorithm and use special purpose algorithm to find a feasible solution if there exists one.
- If $f_s(x^{(k)}) = -\infty$, we arrive at an extreme ray $y^{(k)}$. We add the Benders cut corresponding to this extreme ray $(A_1^T y^{(k)})^T x \leq b^T y^{(k)}$ to the master problem, i.e., $Q(k+1) := Q(k) \cup \{y^{(k)}\}$. Take $k := k + 1$ and go to Step 3.
- If $f_s(x^{(k)})$ is finite, then
- If $f_s(x^{(k)}) = f_m^{(k)}$ we arrive at the optimal solution. The optimum objective value of the original problem is $f_s(x^{(k)}) = f_m^{(k)}$, an optimal x is $x^{(k)}$ and an optimal v is the dual values for the second constraints of the subproblem. We are happily done!
- If $f_s(x^{(k)}) < f_m^{(k)}$ we get an suboptimal vertex $u^{(k)}$. We add the corresponding Benders cut $u_0 + (A_1^T u^{(k)} - c_1)^T x \leq b^T u^{(k)}$ to the master problem, i.e., $T(k+1) := T(k) \cup \{u^{(k)}\}$. Take $k := k + 1$ and go to Step 3.

For a more general approach to Bender's Decomposition you can have a look at Mathieu Besançon's blog.

Data for the problem

The input data is from page 139, Integer programming by Garfinkel and Nemhauser[1].

```
c1 = [-1; -4]
c2 = [-2; -3]

dim_x = length(c1)
dim_u = length(c2)

b = [-2; -3]

A1 = [
    1 -3
    -1 -3
]
A2 = [
    1 -2
    -1 -1
]

M = 1000;
```

How to implement the Benders decomposition algorithm in JuMP

There are two ways we can implement Benders decomposition in JuMP:

- Classical approach: Adding the Benders cuts in a loop, and
- Modern approach: Adding the Benders cuts as lazy constraints.

The classical approach might be inferior to the modern one, as the solver

- might revisit previously eliminated solution, and
- might discard the optimal solution to the original problem in favor of a better but ultimately infeasible solution to the relaxed one.

For more details on the comparison between the two approaches, see Paul Rubin's blog on Benders Decomposition.

Classical Approach: Adding the Benders Cuts in a Loop

Let's describe the master problem first. Note that there are no constraints, which we will added later using Benders decomposition.

Loading the necessary packages

```
using JuMP
import GLPK
import Test
```

Master Problem Description

```
master_problem_model = Model(GLPK.Optimizer);
```

Variable Definition

```
@variable(master_problem_model, 0 <= x[1:dim_x] <= 1e6, Int)
@variable(master_problem_model, t <= 1e6)
```

$$t$$

Objective Setting

```
@objective(master_problem_model, Max, t)
global iter_num = 1

print(master_problem_model)
```

```
Max t
Subject to
 x[1] ≥ 0.0
 x[2] ≥ 0.0
 x[1] ≤ 1.0e6
 x[2] ≤ 1.0e6
 t ≤ 1.0e6
 x[1] integer
 x[2] integer
```

Here is the loop that checks the status of the master problem and the subproblem and then adds necessary Benders cuts accordingly.

```
iter_num = 1

while true
    println("\n----------------------")
    println("Iteration number = ", iter_num)
    println("----------------------\n")
    println("The current master problem is")
    print(master_problem_model)

    optimize!(master_problem_model)

    t_status = termination_status(master_problem_model)
    p_status = primal_status(master_problem_model)

    if p_status == INFEASIBLE_POINT
        println("The problem is infeasible :-(")
        break
    end

    (fm_current, x_current) = if t_status == INFEASIBLE_OR_UNBOUNDED
        (M, M * ones(dim_x))
    elseif p_status == FEASIBLE_POINT
        (value(t), value.(x))
    else
        error("Unexpected status: $((t_status, p_status))")
    end
```

```
    println(
        "Status of the master problem is ",
        t_status,
        "\nwith fm_current = ",
        fm_current,
        "\nx_current = ",
        x_current,
    )

    sub_problem_model = Model(GLPK.Optimizer)

    c_sub = b - A1 * x_current

    local u = @variable(sub_problem_model, u[1:dim_u] >= 0)

    @constraint(
        sub_problem_model,
        constr_ref_subproblem[j = 1:size(A2, 2)],
        A2[:, j]' * u >= c2[j],
    )
    # The second argument of @constraint macro,
    # constr_ref_subproblem[j=1:size(A2,2)] means that the j-th constraint is
    # referenced by constr_ref_subproblem[j].

    @objective(sub_problem_model, Min, c1' * x_current + c_sub' * u)

    print("\nThe current subproblem model is \n")
    print(sub_problem_model)

    optimize!(sub_problem_model)

    t_status_sub = termination_status(sub_problem_model)
    p_status_sub = primal_status(sub_problem_model)

    fs_x_current = objective_value(sub_problem_model)

    u_current = value.(u)

    γ = b' * u_current

    println(
        "Status of the subproblem is ",
        t_status_sub,
        "\nwith fs_x_current = ",
        fs_x_current,
        "\nand fm_current = ",
        fm_current,
    )

    if p_status_sub == FEASIBLE_POINT && fs_x_current == fm_current # we are done
        println("\n################################################")
        println("Optimal solution of the original problem found")
        println("The optimal objective value t is ", fm_current)
        println("The optimal x is ", x_current)
        println("The optimal v is ", dual.(constr_ref_subproblem))
        println("################################################\n")
```

```
        break
    end

    if p_status_sub == FEASIBLE_POINT && fs_x_current < fm_current
        println(
            "\nThere is a suboptimal vertex, add the corresponding constraint",
        )
        cv = A1' * u_current - c1
        @constraint(master_problem_model, t + cv' * x <= γ)
        println("t + ", cv, "ᵀ x <= ", γ)
    end

    if t_status_sub == INFEASIBLE_OR_UNBOUNDED
        println(
            "\nThere is an  extreme ray, adding the corresponding constraint",
        )
        ce = A1' * u_current
        @constraint(master_problem_model, ce' * x <= γ)
        println(ce, "ᵀ x <= ", γ)
    end

    global iter_num += 1
end
```

```
-----------------------
Iteration number = 1
-----------------------

The current master problem is
Max t
Subject to
 x[1] ≥ 0.0
 x[2] ≥ 0.0
 x[1] ≤ 1.0e6
 x[2] ≤ 1.0e6
 t ≤ 1.0e6
 x[1] integer
 x[2] integer
Status of the master problem is OPTIMAL
with fm_current = 1.0e6
x_current = [0.0, 0.0]

The current subproblem model is
Min -2 u[1] - 3 u[2]
Subject to
 constr_ref_subproblem[1] : u[1] - u[2] ≥ -2.0
 constr_ref_subproblem[2] : -2 u[1] - u[2] ≥ -3.0
 u[1] ≥ 0.0
 u[2] ≥ 0.0
Status of the subproblem is OPTIMAL
with fs_x_current = -7.666666666666667
and fm_current = 1.0e6

There is a suboptimal vertex, add the corresponding constraint
t + [-1.0, ᵀ-4.0] x <= -7.666666666666667
```

```
----------------------
Iteration number = 2
----------------------

The current master problem is
Max t
Subject to
 -x[1] - 4 x[2] + t ≤ -7.666666666666667
 x[1] ≥ 0.0
 x[2] ≥ 0.0
 x[1] ≤ 1.0e6
 x[2] ≤ 1.0e6
 t ≤ 1.0e6
 x[1] integer
 x[2] integer
Status of the master problem is OPTIMAL
with fm_current = 1.0e6
x_current = [0.0, 250002.0]

The current subproblem model is
Min 750004 u[1] + 750003 u[2] - 1.000008e6
Subject to
 constr_ref_subproblem[1] : u[1] - u[2] ≥ -2.0
 constr_ref_subproblem[2] : -2 u[1] - u[2] ≥ -3.0
 u[1] ≥ 0.0
 u[2] ≥ 0.0
Status of the subproblem is OPTIMAL
with fs_x_current = -1.000008e6
and fm_current = 1.0e6

There is a suboptimal vertex, add the corresponding constraint
t + [1.0, ᵀ4.0] x <= 0.0

----------------------
Iteration number = 3
----------------------

The current master problem is
Max t
Subject to
 -x[1] - 4 x[2] + t ≤ -7.666666666666667
 x[1] + 4 x[2] + t ≤ 0.0
 x[1] ≥ 0.0
 x[2] ≥ 0.0
 x[1] ≤ 1.0e6
 x[2] ≤ 1.0e6
 t ≤ 1.0e6
 x[1] integer
 x[2] integer
Status of the master problem is OPTIMAL
with fm_current = -4.0
x_current = [4.0, 0.0]

The current subproblem model is
Min -6 u[1] + u[2] - 4
```

```
Subject to
 constr_ref_subproblem[1] : u[1] - u[2] ≥ -2.0
 constr_ref_subproblem[2] : -2 u[1] - u[2] ≥ -3.0
 u[1] ≥ 0.0
 u[2] ≥ 0.0
Status of the subproblem is OPTIMAL
with fs_x_current = -13.0
and fm_current = -4.0

There is a suboptimal vertex, add the corresponding constraint
t + [2.5, ᵀ-0.5] x <= -3.0

----------------------
Iteration number = 4
----------------------

The current master problem is
Max t
Subject to
 -x[1] - 4 x[2] + t ≤ -7.666666666666667
 x[1] + 4 x[2] + t ≤ 0.0
 2.5 x[1] - 0.5 x[2] + t ≤ -3.0
 x[1] ≥ 0.0
 x[2] ≥ 0.0
 x[1] ≤ 1.0e6
 x[2] ≤ 1.0e6
 t ≤ 1.0e6
 x[1] integer
 x[2] integer
Status of the master problem is OPTIMAL
with fm_current = -4.0
x_current = [0.0, 1.0]

The current subproblem model is
Min u[1] - 4
Subject to
 constr_ref_subproblem[1] : u[1] - u[2] ≥ -2.0
 constr_ref_subproblem[2] : -2 u[1] - u[2] ≥ -3.0
 u[1] ≥ 0.0
 u[2] ≥ 0.0
Status of the subproblem is OPTIMAL
with fs_x_current = -4.0
and fm_current = -4.0

################################################
Optimal solution of the original problem found
The optimal objective value t is -4.0
The optimal x is [0.0, 1.0]
The optimal v is [0.0, 0.0]
################################################
```

References

1. Garfinkel, R. & Nemhauser, G. L. Integer programming. (Wiley, 1972).

Tip

This tutorial was generated using Literate.jl. View the source `.jl` file on GitHub.

7.2 Benders Decomposition (Lazy Constraints)

Originally Contributed by: Mathieu Besançon

This notebook describes how to implement the Benders decomposition in JuMP using lazy constraints. We keep the same notation and problem form as the first notebook Benders decomposition.

$$\begin{aligned} \text{maximize} \quad & c_1^T x + c_2^T v \\ \text{subject to} \quad & A_1 x + A_2 v \preceq b \\ & x \succeq 0, x \in \mathbb{Z}^n \\ & v \succeq 0, v \in \mathbb{R}^p \end{aligned}$$

where $b \in \mathbb{R}^m$, $A_1 \in \mathbb{R}^{m \times n}$, $A_2 \in \mathbb{R}^{m \times p}$ and $\mathbb{Z}$ is the set of integers.

Here the symbol $\succeq$ ($\preceq$) stands for element-wise greater (less) than or equal to. Any mixed integer programming problem can be written in the form above.

For a detailed explanation on the Benders decomposition algorithm, see the introduction notebook.

Lazy constraints

Some optimization solvers allow users to interact with them during the solution process by providing user-defined functions which are triggered under certain conditions.

The generic term for these functions is **callback**.

In integer optimization, the main callback types are lazy constraints, user-cuts and heuristic solutions. See the Callbacks section for an introduction on how to use them.

Some callbacks define a new constraint which is only activated when necessary, i.e., when a current solution does not respect them. It can avoid building an optimization model with too many constraints up-front.

This is the case for Benders decomposition, since the sub-problem defines an exponential number of primal vertices and therefore dual cuts.

A detailed explanation on the distinction between user-cuts and lazy constraints is also available on Paul Rubin's blog. He also describes this approach to Benders Decomposition.

We use the data from the original notebook and change the solution algorithm to leverage lazy constraints:

- Step 1 (Initialization)
- Step 2 (defining the subproblem model)
- Step 3 (registering the lazy constraint of the subproblem)

Data for the problem

The input data is from page 139, Integer programming by Garfinkel and Nemhauser[1].

```
c1 = [-1; -4]
c2 = [-2; -3]

dim_x = length(c1)
dim_u = length(c2)

b = [-2; -3]

A1 = [
    1 -3
    -1 -3
]
A2 = [
    1 -2
    -1 -1
]

M = 1000;
```

Loading the necessary packages

```
using JuMP
import GLPK
import Test
```

Subproblem creation

```
function build_subproblem()
    sub_problem_model = Model(GLPK.Optimizer)
    @variable(sub_problem_model, u[1:dim_u] >= 0)
    @constraint(
        sub_problem_model,
        constr_ref_subproblem[j = 1:size(A2, 2)],
        A2[:, j]' * u >= c2[j],
    )
    return (sub_problem_model, u)
end
```

```
build_subproblem (generic function with 1 method)
```

Master Problem Description

```
master_problem_model = Model(GLPK.Optimizer);

(sub_problem_model, u) = build_subproblem();
```

Variable Definition

```
@variable(master_problem_model, 0 <= x[1:dim_x] <= 1e6, Int)
@variable(master_problem_model, t <= 1e6)
```

$$t$$

Objective Setting

```
@objective(master_problem_model, Max, t)

print(master_problem_model)
```

```
Max t
Subject to
 x[1] ≥ 0.0
 x[2] ≥ 0.0
 x[1] ≤ 1.0e6
 x[2] ≤ 1.0e6
 t ≤ 1.0e6
 x[1] integer
 x[2] integer
```

Track the calls to the callback

```
iter_num = 0
```

```
0
```

Define lazy constraints

```
function benders_lazy_constraint_callback(cb_data)
    global iter_num
    iter_num += 1
    println("Iteration number = ", iter_num)

    x_current = callback_value.(Ref(cb_data), x)
    fm_current = callback_value(cb_data, t)

    c_sub = b - A1 * x_current
    @objective(sub_problem_model, Min, c1' * x_current + c_sub' * u)
    optimize!(sub_problem_model)

    print("\nThe current subproblem model is \n")
    print(sub_problem_model)

    t_status_sub = termination_status(sub_problem_model)
    p_status_sub = primal_status(sub_problem_model)

    fs_x_current = objective_value(sub_problem_model)

    u_current = value.(u)

    γ = b' * u_current

    if p_status_sub == FEASIBLE_POINT && fs_x_current ≈ fm_current # we are done
        @info("No additional constraint from the subproblem")
    end
```

```
    if p_status_sub == FEASIBLE_POINT && fs_x_current < fm_current
        println(
            "\nThere is a suboptimal vertex, add the corresponding constraint",
        )
        cv = A1' * u_current - c1
        new_optimality_cons = @build_constraint(t + cv' * x <= γ)
        MOI.submit(
            master_problem_model,
            MOI.LazyConstraint(cb_data),
            new_optimality_cons,
        )
    end

    if t_status_sub == INFEASIBLE_OR_UNBOUNDED
        println(
            "\nThere is an  extreme ray, adding the corresponding constraint",
        )
        ce = A1' * u_current
        new_feasibility_cons = @build_constraint(dot(ce, x) <= γ)
        MOI.submit(
            master_problem_model,
            MOI.LazyConstraint(cb_data),
            new_feasibility_cons,
        )
    end
end

MOI.set(
    master_problem_model,
    MOI.LazyConstraintCallback(),
    benders_lazy_constraint_callback,
)

optimize!(master_problem_model)

t_status = termination_status(master_problem_model)
p_status = primal_status(master_problem_model)

if p_status == INFEASIBLE_POINT
    println("The problem is infeasible :-(")
end

if t_status == INFEASIBLE_OR_UNBOUNDED
    fm_current = M
    x_current = M * ones(dim_x)
end

if p_status == FEASIBLE_POINT
    fm_current = value(t)
    x_current = value.(x)
end


println(
    "Status of the master problem is ",
```

```
    t_status,
    "\nwith fm_current = ",
    fm_current,
    "\nx_current = ",
    x_current,
)
```

```
Iteration number = 1

The current subproblem model is
Min -2 u[1] - 3 u[2]
Subject to
 constr_ref_subproblem[1] : u[1] - u[2] ≥ -2.0
 constr_ref_subproblem[2] : -2 u[1] - u[2] ≥ -3.0
 u[1] ≥ 0.0
 u[2] ≥ 0.0

There is a suboptimal vertex, add the corresponding constraint
Iteration number = 2

The current subproblem model is
Min 750003.75 u[1] + 750002.75 u[2] - 1.0000076666666666e6
Subject to
 constr_ref_subproblem[1] : u[1] - u[2] ≥ -2.0
 constr_ref_subproblem[2] : -2 u[1] - u[2] ≥ -3.0
 u[1] ≥ 0.0
 u[2] ≥ 0.0

There is a suboptimal vertex, add the corresponding constraint
Iteration number = 3

The current subproblem model is
Min 0.875 u[1] - 0.125 u[2] - 3.8333333333333335
Subject to
 constr_ref_subproblem[1] : u[1] - u[2] ≥ -2.0
 constr_ref_subproblem[2] : -2 u[1] - u[2] ≥ -3.0
 u[1] ≥ 0.0
 u[2] ≥ 0.0

There is a suboptimal vertex, add the corresponding constraint
Iteration number = 4

The current subproblem model is
Min -3.8333333333333335
Subject to
 constr_ref_subproblem[1] : u[1] - u[2] ≥ -2.0
 constr_ref_subproblem[2] : -2 u[1] - u[2] ≥ -3.0
 u[1] ≥ 0.0
 u[2] ≥ 0.0
[ Info: No additional constraint from the subproblem
Iteration number = 5

The current subproblem model is
Min -0.875 u[1] + 0.125 u[2] - 3.8333333333333335
Subject to
 constr_ref_subproblem[1] : u[1] - u[2] ≥ -2.0
```

```
 constr_ref_subproblem[2] : -2 u[1] - u[2] ≥ -3.0
 u[1] ≥ 0.0
 u[2] ≥ 0.0

There is a suboptimal vertex, add the corresponding constraint
Iteration number = 6

The current subproblem model is
Min u[2] - 5
Subject to
 constr_ref_subproblem[1] : u[1] - u[2] ≥ -2.0
 constr_ref_subproblem[2] : -2 u[1] - u[2] ≥ -3.0
 u[1] ≥ 0.0
 u[2] ≥ 0.0
[ Info: No additional constraint from the subproblem
Iteration number = 7

The current subproblem model is
Min u[1] - 4
Subject to
 constr_ref_subproblem[1] : u[1] - u[2] ≥ -2.0
 constr_ref_subproblem[2] : -2 u[1] - u[2] ≥ -3.0
 u[1] ≥ 0.0
 u[2] ≥ 0.0
[ Info: No additional constraint from the subproblem
Status of the master problem is OPTIMAL
with fm_current = -4.0
x_current = [0.0, 1.0]
```

References

1. Garfinkel, R. & Nemhauser, G. L. Integer programming. (Wiley, 1972).

Tip

This tutorial was generated using Literate.jl. View the source `.jl` file on GitHub.

7.3 Column Generation

This example solves the cutting stock problem (sometimes also called the cutting rod problem) using a column-generation technique. It is based on https://doi.org/10.5281/zenodo.3329388.

Intuitively, this problem is about cutting large rolls of paper into smaller pieces. There is an exact demand of pieces to meet, and all rolls have the same size. The goal is to meet the demand while maximizing the profits (each paper roll has a fixed cost, each sold piece allows earning some money), which is roughly equivalent to using the smallest amount of rolls to cut (or, equivalently, to minimize the amount of paper waste).

This function takes five parameters:

- `maxwidth`: the maximum width of a roll (or length of a rod)
- `widths`: an array of the requested widths

- `rollcost`: the cost of a complete roll
- `demand`: the demand, in number of pieces, for each width
- `prices`: the selling price for each width

Mathematically, this problem might be formulated with two variables:

- `x[i, j] ∈ ℕ`: the number of times the width `i` is cut out of the roll `j`
- `y[j] ∈ `: whether the roll `j` is used

Several constraints are needed:

- the demand must be satisfied, for each width `i`: ∑j x[i, j] = demand[i]
- the roll size cannot be exceed, for each roll `j` that is used: ∑i x[i, j] width[i] ≤ maxwidth y[j]

If you want to implement this naïve model, you will need an upper bound on the number of rolls to use: the simplest one is to consider that each required width is cut from its own roll, i.e. `j` varies from 1 to ∑i demand[i].

This example prefers a more advanced technique to solve this problem: column generation.

It considers a different set of variables: patterns of width to cut a roll. The decisions then become the number of times each pattern is used (i.e. the number of rolls that are cut following this pattern).

The intelligence comes from the way these patterns are chosen: not all of them are considered, but only the "interesting" ones, within the master problem.

A "pricing" problem is used to decide whether a new pattern should be generated or not (it is implemented in the function `solve_pricing`). "Interesting" means, for a pattern, that the optimal solution may use this cutting pattern.

In more detail, the solving process is the following. First, a series of dumb patterns are generated (just one width per roll, repeated until the roll is completely cut). Then, the master problem is solved with these first patterns and its dual solution is passed on to the pricing problem. The latter decides if there is a new pattern to include in the formulation or not; if so, it returns it to the master problem. The master is solved again, the new dual variables are given to the pricing problem, until there is no more pattern to generate from the pricing problem: all "interesting" patterns have been generated, and the master can take its optimal decision.

In the implementation, the variables deciding how many times a pattern is chosen are called θ.

For more information on column-generation techniques applied on the cutting stock problem, you can see:

- Integer programming column generation strategies for the cutting stock problem and its variants
- Tackling the cutting stock problem

This example uses the following packages:

```
using JuMP
import GLPK
import SparseArrays
```

The function `solve_pricing` implements the pricing problem for the function `example_cutting_stock`.

It takes, as input, the dual solution from the master problem and the cutting stock instance.

It outputs either a new cutting pattern, or `nothing` if no pattern could improve the current cost.

```
function solve_pricing(
    dual_demand_satisfaction,
    maxwidth,
    widths,
    rollcost,
    demand,
    prices,
)
    reduced_costs = dual_demand_satisfaction + prices
    n = length(reduced_costs)
    # The actual pricing model.
    submodel = Model(GLPK.Optimizer)
    set_silent(submodel)
    @variable(submodel, xs[1:n] >= 0, Int)
    @constraint(submodel, sum(xs .* widths) <= maxwidth)
    @objective(submodel, Max, sum(xs .* reduced_costs))
    optimize!(submodel)
    new_pattern = round.(Int, value.(xs))
    net_cost =
        rollcost - sum(new_pattern .* (dual_demand_satisfaction .+ prices))
    # If the net cost of this new pattern is nonnegative, no more patterns to add.
    return net_cost >= 0 ? nothing : new_pattern
end

function example_cutting_stock(; max_gen_cols::Int = 5_000)
    maxwidth = 100.0
    rollcost = 500.0
    prices = [
        167.0,
        197.0,
        281.0,
        212.0,
        225.0,
        111.0,
        93.0,
        129.0,
        108.0,
        106.0,
        55.0,
        85.0,
        66.0,
        44.0,
        47.0,
        15.0,
        24.0,
        13.0,
        16.0,
        14.0,
    ]
    widths = [
        75.0,
```

```
        75.0,
        75.0,
        75.0,
        75.0,
        53.8,
        53.0,
        51.0,
        50.2,
        32.2,
        30.8,
        29.8,
        20.1,
        16.2,
        14.5,
        11.0,
        8.6,
        8.2,
        6.6,
        5.1,
    ]
    demand = [
        38,
        44,
        30,
        41,
        36,
        33,
        36,
        41,
        35,
        37,
        44,
        49,
        37,
        36,
        42,
        33,
        47,
        35,
        49,
        42,
    ]
    nwidths = length(prices)
    n = length(widths)
    ncols = length(widths)
    # Initial set of patterns (stored in a sparse matrix: a pattern won't
    # include many different cuts).
    patterns = SparseArrays.spzeros(UInt16, n, ncols)
    for i in 1:n
        patterns[i, i] =
            min(floor(Int, maxwidth / widths[i]), round(Int, demand[i]))
    end
    # Write the master problem with this "reduced" set of patterns.
    # Not yet integer variables: otherwise, the dual values may make no sense
    # (actually, GLPK will yell at you if you're trying to get duals for
    # integer problems).
```

```
    m = Model(GLPK.Optimizer)
    set_silent(m)
    @variable(m, θ[1:ncols] >= 0)
    @objective(
        m,
        Min,
        sum(
            θ[p] * (rollcost - sum(patterns[j, p] * prices[j] for j in 1:n)) for
            p in 1:ncols
        )
    )
    @constraint(
        m,
        demand_satisfaction[j = 1:n],
        sum(patterns[j, p] * θ[p] for p in 1:ncols) >= demand[j]
    )
    # First solve of the master problem.
    optimize!(m)
    if termination_status(m) != OPTIMAL
        warn("Master not optimal ($ncols patterns so far)")
    end
    # Then, generate new patterns, based on the dual information.
    while ncols - n <= max_gen_cols ## Generate at most max_gen_cols columns.
        if !has_duals(m)
            break
        end
        new_pattern = solve_pricing(
            dual.(demand_satisfaction),
            maxwidth,
            widths,
            rollcost,
            demand,
            prices,
        )
        # No new pattern to add to the formulation: done!
        if new_pattern === nothing
            break
        end
        # Otherwise, add the new pattern to the master problem, recompute the
        # duals, and go on waltzing one more time with the pricing problem.
        ncols += 1
        patterns = hcat(patterns, new_pattern)
        # One new variable.
        push!(θ, @variable(m, base_name = "θ", lower_bound = 0))
        # Update the objective function.
        set_objective_coefficient(
            m,
            θ[ncols],
            rollcost - sum(patterns[j, ncols] * prices[j] for j in 1:n),
        )
        # Update the constraint number j if the new pattern impacts this production.
        for j in 1:n
            if new_pattern[j] > 0
                set_normalized_coefficient(
                    demand_satisfaction[j],
                    θ[ncols],
```

```
                    new_pattern[j],
                )
            end
        end
        # Solve the new master problem to update the dual variables.
        optimize!(m)
        if termination_status(m) != OPTIMAL
            @warn("Master not optimal ($ncols patterns so far)")
        end
    end
    # Finally, impose the master variables to be integer and resolve.
    # To be exact, at each node in the branch-and-bound tree, we would need to
    # restart the column generation process (just in case a new column would be
    # interesting to add). This way, we only get an upper bound (a feasible
    # solution).
    set_integer.(θ)
    optimize!(m)
    if termination_status(m) != OPTIMAL
        @warn("Final master not optimal ($ncols patterns)")
        return
    end
    println("Final solution:")
    for i in 1:length(θ)
        if value(θ[i]) > 0.5
            println("$(round(Int, value(θ[i]))) units of pattern $(i)")
        end
    end
    return
end

example_cutting_stock()
```

```
Final solution:
15 units of pattern 21
26 units of pattern 22
10 units of pattern 27
33 units of pattern 28
30 units of pattern 29
44 units of pattern 30
30 units of pattern 31
26 units of pattern 32
7 units of pattern 35
23 units of pattern 36
1 units of pattern 39
34 units of pattern 41
23 units of pattern 42
2 units of pattern 43
13 units of pattern 44
9 units of pattern 45
5 units of pattern 46
3 units of pattern 47
```

Tip

This tutorial was generated using Literate.jl. View the source `.jl` file on GitHub.

Chapter 8

Applications

8.1 Power Systems

Originally Contributed by: Yury Dvorkin and Miles Lubin

This tutorial demonstrates how to formulate basic power systems engineering models in JuMP.

We will consider basic "economic dispatch" and "unit commitment" models without taking into account transmission constraints.

For this tutorial, we use the following packages:

```
using JuMP
import DataFrames
import GLPK
import Plots
import StatsPlots
```

Economic dispatch

Economic dispatch (ED) is an optimization problem that minimizes the cost of supplying energy demand subject to operational constraints on power system assets. In its simplest modification, ED is an LP problem solved for an aggregated load and wind forecast and for a single infinitesimal moment.

Mathematically, the ED problem can be written as follows:

$$\min \sum_{i \in I} c_i^g \cdot g_i + c^w \cdot w,$$

where c_i and g_i are the incremental cost (\$/MWh) and power output (MW) of the i^{th} generator, respectively, and c^w and w are the incremental cost (\$/MWh) and wind power injection (MW), respectively.

Subject to the constraints:

- Minimum ($g^{\min}$) and maximum ($g^{\max}$) limits on power outputs of generators: $g_i^{\min} \leq g_i \leq g_i^{\max}$.
- Constraint on the wind power injection: $0 \leq w \leq w^f$, where w and w^f are the wind power injection and wind power forecast, respectively.
- Power balance constraint: $\sum_{i \in I} g_i + w = d^f$, where d^f is the demand forecast.

Further reading on ED models can be found in A. J. Wood, B. F. Wollenberg, and G. B. Sheblé, "Power Generation, Operation and Control", Wiley, 2013.

Define some input data about the test system.

We define some thermal generators:

```
function ThermalGenerator(
    min::Float64,
    max::Float64,
    fixed_cost::Float64,
    variable_cost::Float64,
)
    return (
        min = min,
        max = max,
        fixed_cost = fixed_cost,
        variable_cost = variable_cost,
    )
end

generators = [
    ThermalGenerator(0.0, 1000.0, 1000.0, 50.0),
    ThermalGenerator(300.0, 1000.0, 0.0, 100.0),
]
```

```
2-element Vector{NamedTuple{(:min, :max, :fixed_cost, :variable_cost), NTuple{4, Float64}}}:
 (min = 0.0, max = 1000.0, fixed_cost = 1000.0, variable_cost = 50.0)
 (min = 300.0, max = 1000.0, fixed_cost = 0.0, variable_cost = 100.0)
```

A wind generator

```
WindGenerator(variable_cost::Float64) = (variable_cost = variable_cost,)

wind_generator = WindGenerator(50.0)
```

```
(variable_cost = 50.0,)
```

And a scenario

```
function Scenario(demand::Float64, wind::Float64)
    return (demand = demand, wind = wind)
end

scenario = Scenario(1500.0, 200.0)
```

```
(demand = 1500.0, wind = 200.0)
```

Create a function `solve_ed`, which solves the economic dispatch problem for a given set of input parameters.

```
function solve_ed(generators::Vector, wind, scenario)
    # Define the economic dispatch (ED) model
    ed = Model(GLPK.Optimizer)
    # Define decision variables
```

```
    # power output of generators
    N = length(generators)
    @variable(ed, generators[i].min <= g[i = 1:N] <= generators[i].max)
    # wind power injection
    @variable(ed, 0 <= w <= scenario.wind)
    # Define the objective function
    @objective(
        ed,
        Min,
        sum(generators[i].variable_cost * g[i] for i in 1:N) +
        wind.variable_cost * w,
    )
    # Define the power balance constraint
    @constraint(ed, sum(g[i] for i in 1:N) + w == scenario.demand)
    # Solve statement
    optimize!(ed)
    # return the optimal value of the objective function and its minimizers
    return (
        g = value.(g),
        w = value(w),
        wind_spill = scenario.wind - value(w),
        total_cost = objective_value(ed),
    )
end
```

```
solve_ed (generic function with 1 method)
```

Solve the economic dispatch problem

```
solution = solve_ed(generators, wind_generator, scenario);

println("Dispatch of Generators: ", solution.g, " MW")
println("Dispatch of Wind: ", solution.w, " MW")
println("Wind spillage: ", solution.wind_spill, " MW")
println("Total cost: \$", solution.total_cost)
```

```
Dispatch of Generators: [1000.0, 300.0] MW
Dispatch of Wind: 200.0 MW
Wind spillage: 0.0 MW
Total cost: $90000.0
```

Economic dispatch with adjustable incremental costs

In the following exercise we adjust the incremental cost of generator G1 and observe its impact on the total cost.

```
function scale_generator_cost(g, scale)
    return ThermalGenerator(g.min, g.max, g.fixed_cost, scale * g.variable_cost)
end

start = time()
c_g_scale_df = DataFrames.DataFrame(
    # Scale factor
    scale = Float64[],
```

```
    # Dispatch of Generator 1 [MW]
    dispatch_G1 = Float64[],
    # Dispatch of Generator 2 [MW]
    dispatch_G2 = Float64[],
    # Dispatch of Wind [MW]
    dispatch_wind = Float64[],
    # Spillage of Wind [MW]
    spillage_wind = Float64[],
    # Total cost [$]
    total_cost = Float64[],
)
for c_g1_scale in 0.5:0.1:3.0
    # Update the incremental cost of the first generator at every iteration.
    new_generators = scale_generator_cost.(generators, [c_g1_scale, 1.0])
    # Solve the ed problem with the updated incremental cost
    sol = solve_ed(new_generators, wind_generator, scenario)
    push!(
        c_g_scale_df,
        (c_g1_scale, sol.g[1], sol.g[2], sol.w, sol.wind_spill, sol.total_cost),
    )
end
print(string("elapsed time: ", time() - start, " seconds"))
```

```
elapsed time: 0.15984010696411133 seconds
```

```
c_g_scale_df
```

	scale	dispatch_G1	dispatch_G2	dispatch_wind	spillage_wind	total_cost
	Float64	Float64	Float64	Float64	Float64	Float64
1	0.5	1000.0	300.0	200.0	0.0	65000.0
2	0.6	1000.0	300.0	200.0	0.0	70000.0
3	0.7	1000.0	300.0	200.0	0.0	75000.0
4	0.8	1000.0	300.0	200.0	0.0	80000.0
5	0.9	1000.0	300.0	200.0	0.0	85000.0
6	1.0	1000.0	300.0	200.0	0.0	90000.0
7	1.1	1000.0	300.0	200.0	0.0	95000.0
8	1.2	1000.0	300.0	200.0	0.0	100000.0
9	1.3	1000.0	300.0	200.0	0.0	105000.0
10	1.4	1000.0	300.0	200.0	0.0	110000.0
11	1.5	1000.0	300.0	200.0	0.0	115000.0
12	1.6	1000.0	300.0	200.0	0.0	120000.0
13	1.7	1000.0	300.0	200.0	0.0	125000.0
14	1.8	1000.0	300.0	200.0	0.0	130000.0
15	1.9	1000.0	300.0	200.0	0.0	135000.0
16	2.0	1000.0	300.0	200.0	0.0	140000.0
17	2.1	300.0	1000.0	200.0	0.0	141500.0
18	2.2	300.0	1000.0	200.0	0.0	143000.0
19	2.3	300.0	1000.0	200.0	0.0	144500.0
20	2.4	300.0	1000.0	200.0	0.0	146000.0
21	2.5	300.0	1000.0	200.0	0.0	147500.0
22	2.6	300.0	1000.0	200.0	0.0	149000.0
23	2.7	300.0	1000.0	200.0	0.0	150500.0
24	2.8	300.0	1000.0	200.0	0.0	152000.0
25	2.9	300.0	1000.0	200.0	0.0	153500.0
26	3.0	300.0	1000.0	200.0	0.0	155000.0

Modifying the JuMP model in-place

Note that in the previous exercise we entirely rebuilt the optimization model at every iteration of the internal loop, which incurs an additional computational burden. This burden can be alleviated if instead of re-building the entire model, we modify a specific constraint(s) or the objective function, as it shown in the example below.

Compare the computing time in case of the above and below models.

```
function solve_ed_inplace(
    generators::Vector,
    wind,
    scenario,
    scale::AbstractVector{Float64},
)
    obj_out = Float64[]
    w_out = Float64[]
    g1_out = Float64[]
    g2_out = Float64[]
    # This function only works for two generators
    @assert length(generators) == 2
    ed = Model(GLPK.Optimizer)
    N = length(generators)
    @variable(ed, generators[i].min <= g[i = 1:N] <= generators[i].max)
    @variable(ed, 0 <= w <= scenario.wind)
    @objective(
```

```
        ed,
        Min,
        sum(generators[i].variable_cost * g[i] for i in 1:N) +
        wind.variable_cost * w,
    )
    @constraint(ed, sum(g[i] for i in 1:N) + w == scenario.demand)
    for c_g1_scale in scale
        @objective(
            ed,
            Min,
            c_g1_scale * generators[1].variable_cost * g[1] +
            generators[2].variable_cost * g[2] +
            wind.variable_cost * w,
        )
        optimize!(ed)
        push!(obj_out, objective_value(ed))
        push!(w_out, value(w))
        push!(g1_out, value(g[1]))
        push!(g2_out, value(g[2]))
    end
    df = DataFrames.DataFrame(
        scale = scale,
        dispatch_G1 = g1_out,
        dispatch_G2 = g2_out,
        dispatch_wind = w_out,
        spillage_wind = scenario.wind .- w_out,
        total_cost = obj_out,
    )
    return df
end

start = time()
inplace_df = solve_ed_inplace(generators, wind_generator, scenario, 0.5:0.1:3.0)
print(string("elapsed time: ", time() - start, " seconds"))
```

```
elapsed time: 0.20675301551818848 seconds
```

Adjusting specific constraints and/or the objective function is faster than re-building the entire model.

```
inplace_df
```

	scale	dispatch_G1	dispatch_G2	dispatch_wind	spillage_wind	total_cost
	Float64	Float64	Float64	Float64	Float64	Float64
1	0.5	1000.0	300.0	200.0	0.0	65000.0
2	0.6	1000.0	300.0	200.0	0.0	70000.0
3	0.7	1000.0	300.0	200.0	0.0	75000.0
4	0.8	1000.0	300.0	200.0	0.0	80000.0
5	0.9	1000.0	300.0	200.0	0.0	85000.0
6	1.0	1000.0	300.0	200.0	0.0	90000.0
7	1.1	1000.0	300.0	200.0	0.0	95000.0
8	1.2	1000.0	300.0	200.0	0.0	100000.0
9	1.3	1000.0	300.0	200.0	0.0	105000.0
10	1.4	1000.0	300.0	200.0	0.0	110000.0
11	1.5	1000.0	300.0	200.0	0.0	115000.0
12	1.6	1000.0	300.0	200.0	0.0	120000.0
13	1.7	1000.0	300.0	200.0	0.0	125000.0
14	1.8	1000.0	300.0	200.0	0.0	130000.0
15	1.9	1000.0	300.0	200.0	0.0	135000.0
16	2.0	1000.0	300.0	200.0	0.0	140000.0
17	2.1	300.0	1000.0	200.0	0.0	141500.0
18	2.2	300.0	1000.0	200.0	0.0	143000.0
19	2.3	300.0	1000.0	200.0	0.0	144500.0
20	2.4	300.0	1000.0	200.0	0.0	146000.0
21	2.5	300.0	1000.0	200.0	0.0	147500.0
22	2.6	300.0	1000.0	200.0	0.0	149000.0
23	2.7	300.0	1000.0	200.0	0.0	150500.0
24	2.8	300.0	1000.0	200.0	0.0	152000.0
25	2.9	300.0	1000.0	200.0	0.0	153500.0
26	3.0	300.0	1000.0	200.0	0.0	155000.0

Inefficient usage of wind generators

The economic dispatch problem does not perform commitment decisions and, thus, assumes that all generators must be dispatched at least at their minimum power output limit. This approach is not cost efficient and may lead to absurd decisions. For example, if $d = \sum_{i \in I} g_i^{\min}$, the wind power injection must be zero, i.e. all available wind generation is spilled, to meet the minimum power output constraints on generators.

In the following example, we adjust the total demand and observed how it affects wind spillage.

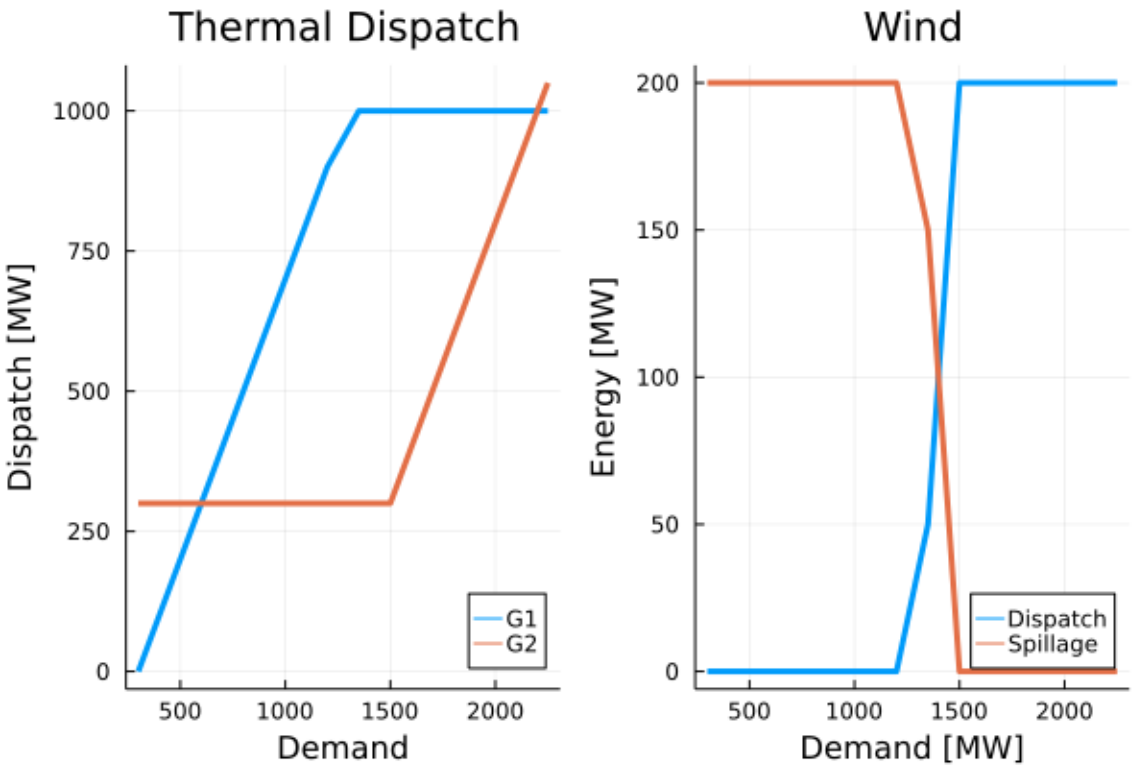
```
demand_scale_df = DataFrames.DataFrame(
    demand = Float64[],
    dispatch_G1 = Float64[],
    dispatch_G2 = Float64[],
    dispatch_wind = Float64[],
    spillage_wind = Float64[],
    total_cost = Float64[],
)

function scale_demand(scenario, scale)
    return Scenario(scale * scenario.demand, scenario.wind)
end

for demand_scale in 0.2:0.1:1.5
    new_scenario = scale_demand(scenario, demand_scale)
    sol = solve_ed(generators, wind_generator, new_scenario)
```

```
    push!(
        demand_scale_df,
        (
            new_scenario.demand,
            sol.g[1],
            sol.g[2],
            sol.w,
            sol.wind_spill,
            sol.total_cost,
        ),
    )
end

demand_scale_df
```

	demand	dispatch_G1	dispatch_G2	dispatch_wind	spillage_wind	total_cost
	Float64	Float64	Float64	Float64	Float64	Float64
1	300.0	0.0	300.0	0.0	200.0	30000.0
2	450.0	150.0	300.0	0.0	200.0	37500.0
3	600.0	300.0	300.0	0.0	200.0	45000.0
4	750.0	450.0	300.0	0.0	200.0	52500.0
5	900.0	600.0	300.0	0.0	200.0	60000.0
6	1050.0	750.0	300.0	0.0	200.0	67500.0
7	1200.0	900.0	300.0	0.0	200.0	75000.0
8	1350.0	1000.0	300.0	50.0	150.0	82500.0
9	1500.0	1000.0	300.0	200.0	0.0	90000.0
10	1650.0	1000.0	450.0	200.0	0.0	105000.0
11	1800.0	1000.0	600.0	200.0	0.0	120000.0
12	1950.0	1000.0	750.0	200.0	0.0	135000.0
13	2100.0	1000.0	900.0	200.0	0.0	150000.0
14	2250.0	1000.0	1050.0	200.0	0.0	165000.0

```
dispatch_plot = StatsPlots.@df(
    demand_scale_df,
    Plots.plot(
        :demand,
        [:dispatch_G1, :dispatch_G2],
        labels = ["G1" "G2"],
        title = "Thermal Dispatch",
        legend = :bottomright,
        linewidth = 3,
        xlabel = "Demand",
        ylabel = "Dispatch [MW]",
    ),
)

wind_plot = StatsPlots.@df(
    demand_scale_df,
    Plots.plot(
        :demand,
        [:dispatch_wind, :spillage_wind],
        labels = ["Dispatch" "Spillage"],
        title = "Wind",
        legend = :bottomright,
```

```
        linewidth = 3,
        xlabel = "Demand [MW]",
        ylabel = "Energy [MW]",
    ),
)

Plots.plot(dispatch_plot, wind_plot)
```



This particular drawback can be overcome by introducing binary decisions on the "on/off" status of generators. This model is called unit commitment and considered later in these notes.

For further reading on the interplay between wind generation and the minimum power output constraints of generators, we refer interested readers to R. Baldick, "Wind and Energy Markets: A Case Study of Texas," IEEE Systems Journal, vol. 6, pp. 27-34, 2012.

Unit commitment

The Unit Commitment (UC) model can be obtained from ED model by introducing binary variable associated with each generator. This binary variable can attain two values: if it is "1", the generator is synchronized and, thus, can be dispatched, otherwise, i.e. if the binary variable is "0", that generator is not synchronized and its power output is set to 0.

To obtain the mathematical formulation of the UC model, we will modify the constraints of the ED model as follows:

$$g_i^{\min} \cdot u_{t,i} \leq g_i \leq g_i^{\max} \cdot u_{t,i},$$

where $u_i \in \{0, 1\}$. In this constraint, if $u_i = 0$, then $g_i = 0$. On the other hand, if $u_i = 1$, then $g_i^{min} \leq g_i \leq g_i^{max}$.

For further reading on the UC problem we refer interested readers to G. Morales-Espana, J. M. Latorre, and A. Ramos, "Tight and Compact MILP Formulation for the Thermal Unit Commitment Problem," IEEE Transactions on Power Systems, vol. 28, pp. 4897-4908, 2013.

In the following example we convert the ED model explained above to the UC model.

```
function solve_uc(generators::Vector, wind, scenario)
    uc = Model(GLPK.Optimizer)
    N = length(generators)
    @variable(uc, generators[i].min <= g[i = 1:N] <= generators[i].max)
    @variable(uc, 0 <= w <= scenario.wind)
    @constraint(uc, sum(g[i] for i in 1:N) + w == scenario.demand)
    # !!! New: add binary on-off variables for each generator
    @variable(uc, u[i = 1:N], Bin)
    @constraint(uc, [i = 1:N], g[i] <= generators[i].max * u[i])
    @constraint(uc, [i = 1:N], g[i] >= generators[i].min * u[i])
    @objective(
        uc,
        Min,
        sum(generators[i].variable_cost * g[i] for i in 1:N) +
        wind.variable_cost * w +
        # !!! new
        sum(generators[i].fixed_cost * u[i] for i in 1:N)
    )
    optimize!(uc)
    status = termination_status(uc)
    if status != OPTIMAL
        return (status = status,)
    end
    return (
        status = status,
        g = value.(g),
        w = value(w),
        wind_spill = scenario.wind - value(w),
        u = value.(u),
        total_cost = objective_value(uc),
    )
end
```

```
solve_uc (generic function with 1 method)
```

Solve the economic dispatch problem

```
solution = solve_uc(generators, wind_generator, scenario)

println("Dispatch of Generators: ", solution.g, " MW")
println("Commitments of Generators: ", solution.u)
println("Dispatch of Wind: ", solution.w, " MW")
println("Wind spillage: ", solution.wind_spill, " MW")
println("Total cost: \$", solution.total_cost)
```

```
Dispatch of Generators: [1000.0, 300.0] MW
Commitments of Generators: [1.0, 1.0]
Dispatch of Wind: 200.0 MW
Wind spillage: 0.0 MW
Total cost: $91000.0
```

Unit Commitment as a function of demand

After implementing the UC model, we can now assess the interplay between the minimum power output constraints on generators and wind generation.

```
uc_df = DataFrames.DataFrame(
    demand = Float64[],
    commitment_G1 = Float64[],
    commitment_G2 = Float64[],
    dispatch_G1 = Float64[],
    dispatch_G2 = Float64[],
    dispatch_wind = Float64[],
    spillage_wind = Float64[],
    total_cost = Float64[],
)

for demand_scale in 0.2:0.1:1.5
    new_scenario = scale_demand(scenario, demand_scale)
    sol = solve_uc(generators, wind_generator, new_scenario)
    if sol.status == OPTIMAL
        push!(
            uc_df,
            (
                new_scenario.demand,
                sol.u[1],
                sol.u[2],
                sol.g[1],
                sol.g[2],
                sol.w,
                sol.wind_spill,
                sol.total_cost,
            ),
        )
    end
    println("Status: $(sol.status) for demand_scale = $(demand_scale)")
end
```

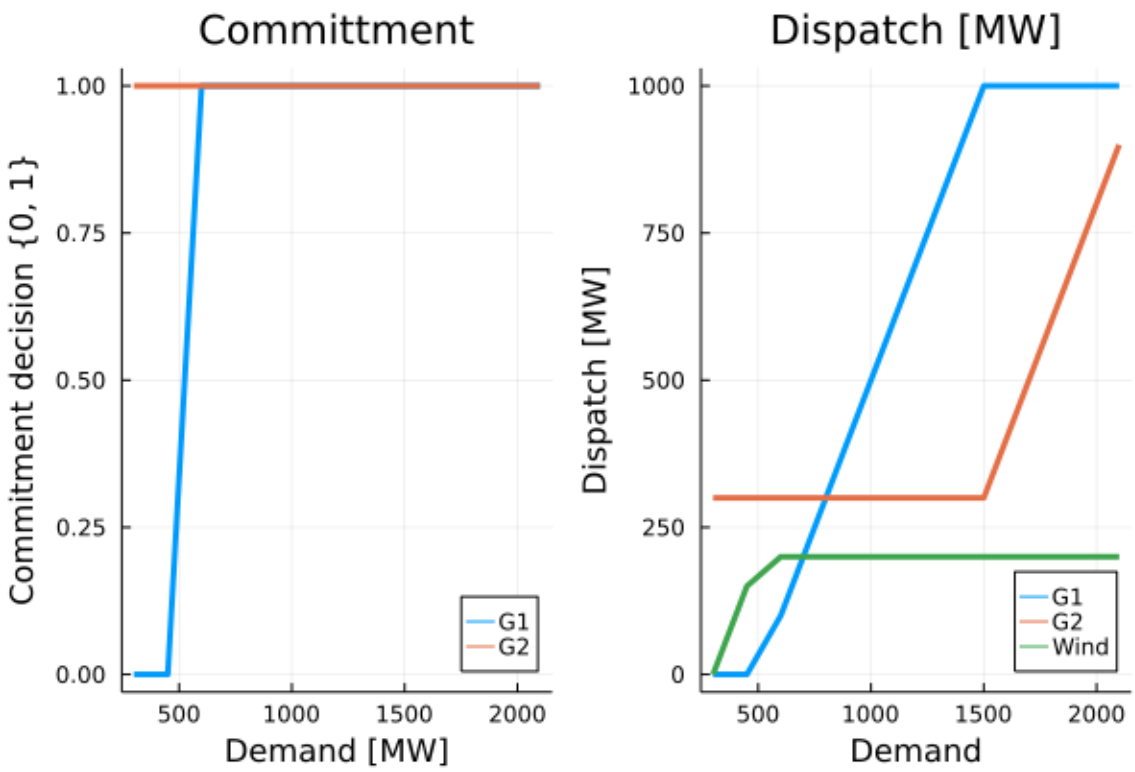
```
Status: OPTIMAL for demand_scale = 0.2
Status: OPTIMAL for demand_scale = 0.3
Status: OPTIMAL for demand_scale = 0.4
Status: OPTIMAL for demand_scale = 0.5
Status: OPTIMAL for demand_scale = 0.6
Status: OPTIMAL for demand_scale = 0.7
Status: OPTIMAL for demand_scale = 0.8
Status: OPTIMAL for demand_scale = 0.9
Status: OPTIMAL for demand_scale = 1.0
Status: OPTIMAL for demand_scale = 1.1
Status: OPTIMAL for demand_scale = 1.2
Status: OPTIMAL for demand_scale = 1.3
```

```
Status: OPTIMAL for demand_scale = 1.4
Status: INFEASIBLE for demand_scale = 1.5
```

```
uc_df
```

	demand	commitment_G1	commitment_G2	dispatch_G1	dispatch_G2	dispatch_wind	spillage_wind	total_c
	Float64	Float64	Float64	Float64	Float64	Float64	Float64	Float
1	300.0	0.0	1.0	0.0	300.0	0.0	200.0	3000
2	450.0	0.0	1.0	0.0	300.0	150.0	50.0	3750
3	600.0	1.0	1.0	100.0	300.0	200.0	0.0	4600
4	750.0	1.0	1.0	250.0	300.0	200.0	0.0	5350
5	900.0	1.0	1.0	400.0	300.0	200.0	0.0	6100
6	1050.0	1.0	1.0	550.0	300.0	200.0	0.0	6850
7	1200.0	1.0	1.0	700.0	300.0	200.0	0.0	7600
8	1350.0	1.0	1.0	850.0	300.0	200.0	0.0	8350
9	1500.0	1.0	1.0	1000.0	300.0	200.0	0.0	9100
10	1650.0	1.0	1.0	1000.0	450.0	200.0	0.0	10600
11	1800.0	1.0	1.0	1000.0	600.0	200.0	0.0	12100
12	1950.0	1.0	1.0	1000.0	750.0	200.0	0.0	13600
13	2100.0	1.0	1.0	1000.0	900.0	200.0	0.0	15100

```
commitment_plot = StatsPlots.@df(
    uc_df,
    Plots.plot(
        :demand,
        [:commitment_G1, :commitment_G2],
        labels = ["G1" "G2"],
        title = "Committment",
        legend = :bottomright,
        linewidth = 3,
        xlabel = "Demand [MW]",
        ylabel = "Commitment decision {0, 1}",
    ),
)

dispatch_plot = StatsPlots.@df(
    uc_df,
    Plots.plot(
        :demand,
        [:dispatch_G1, :dispatch_G2, :dispatch_wind],
        labels = ["G1" "G2" "Wind"],
        title = "Dispatch [MW]",
        legend = :bottomright,
        linewidth = 3,
        xlabel = "Demand",
        ylabel = "Dispatch [MW]",
    ),
)

Plots.plot(commitment_plot, dispatch_plot)
```



Nonlinear economic dispatch

As a final example, we modify our economic dispatch problem in two ways:

- The thermal cost function is user-defined
- The output of the wind is only the square-root of the dispatch

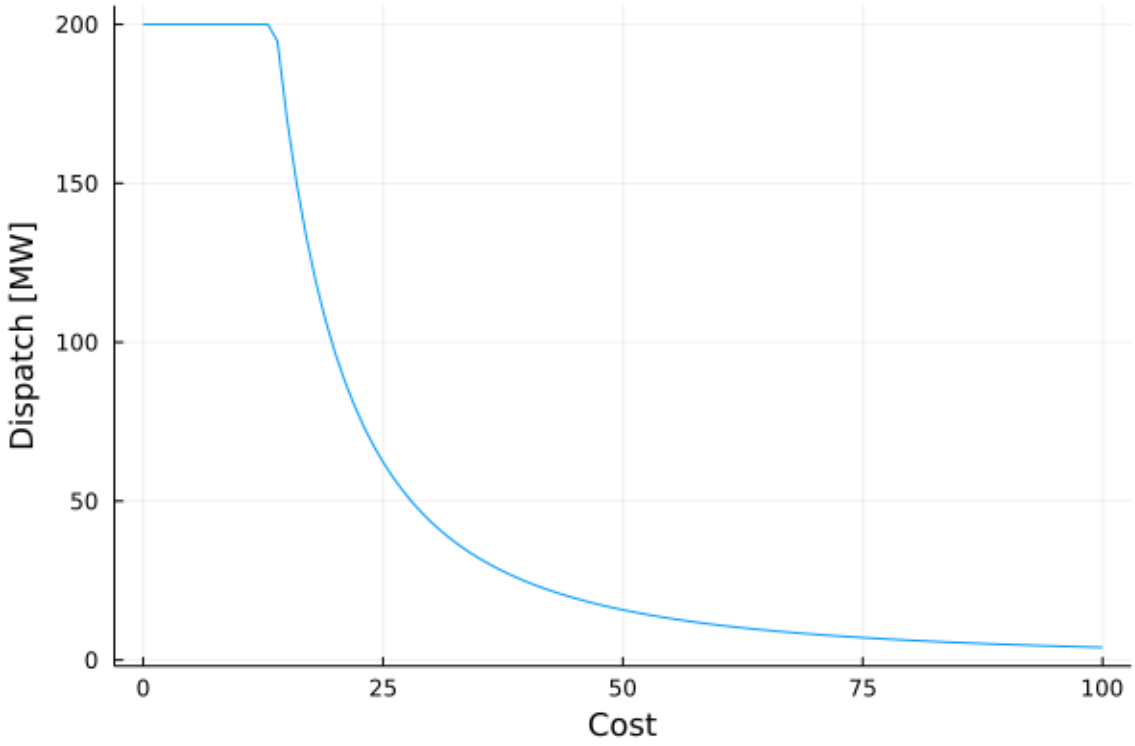
```
import Ipopt

"""
    thermal_cost_function(g)

A user-defined thermal cost function in pure-Julia! You can include
nonlinearities, and even things like control flow.

!!! warning
    It's still up to you to make sure that the function has a meaningful
    derivative.
"""
function thermal_cost_function(g)
    if g <= 500
        return g
    else
        return g + 1e-2 * (g - 500)^2
    end
end
```

```
function solve_nonlinear_ed(
    generators::Vector,
    wind,
    scenario;
    silent::Bool = false,
)
    model = Model(Ipopt.Optimizer)
    if silent
        set_silent(model)
    end
    register(model, :tcf, 1, thermal_cost_function; autodiff = true)
    N = length(generators)
    @variable(model, generators[i].min <= g[i = 1:N] <= generators[i].max)
    @variable(model, 0 <= w <= scenario.wind)
    @NLobjective(
        model,
        Min,
        sum(generators[i].variable_cost * tcf(g[i]) for i in 1:N) +
        wind.variable_cost * w,
    )
    @NLconstraint(model, sum(g[i] for i in 1:N) + sqrt(w) == scenario.demand)
    optimize!(model)
    return (
        g = value.(g),
        w = value(w),
        wind_spill = scenario.wind - value(w),
        total_cost = objective_value(model),
    )
end

solution = solve_nonlinear_ed(generators, wind_generator, scenario)
```

```
(g = [847.3509933774712, 648.6754966887423], w = 15.788781193899027, wind_spill = 184.211218806101,
     total_cost = 190455.298013245)
```

Now let's see how the wind is dispatched as a function of the cost:

```
wind_cost = 0.0:1:100
wind_dispatch = Float64[]
for c in wind_cost
    sol = solve_nonlinear_ed(
        generators,
        WindGenerator(c),
        scenario;
        silent = true,
    )
    push!(wind_dispatch, sol.w)
end

Plots.plot(
    wind_cost,
    wind_dispatch,
    xlabel = "Cost",
    ylabel = "Dispatch [MW]",
    label = false,
)
```



Tip

This tutorial was generated using Literate.jl. View the source `.jl` file on GitHub.

Part III

Manual

Chapter 9

Models

> **Info**
>
> JuMP uses "optimizer" as a synonym for "solver." Our convention is to use "solver" to refer to the underlying software, and use "optimizer" to refer to the Julia object that wraps the solver. For example, GLPK is a solver, and `GLPK.Optimizer` is an optimizer.

9.1 Create a model

Create a model by passing an optimizer to `Model`:

```
julia> model = Model(GLPK.Optimizer)
A JuMP Model
Feasibility problem with:
Variables: 0
Model mode: AUTOMATIC
CachingOptimizer state: EMPTY_OPTIMIZER
Solver name: GLPK
```

or by calling `set_optimizer` on an empty `Model`:

```
julia> model = Model()
A JuMP Model
Feasibility problem with:
Variables: 0
Model mode: AUTOMATIC
CachingOptimizer state: NO_OPTIMIZER
Solver name: No optimizer attached.

julia> set_optimizer(model, GLPK.Optimizer)
```

> **Tip**
>
> Don't know what the fields `Model mode`, `CachingOptimizer state` mean? Read the Backends section.

What is the difference?

For most models, there is no difference between passing the optimizer to `Model`, and calling `set_optimizer`.

However, if an optimizer does not support a constraint in the model, the timing of when an error will be thrown can differ:

- If you pass an optimizer, an error will be thrown when you try to add the constraint.
- If you call `set_optimizer`, an error will be thrown when you try to solve the model via `optimize!`.

Therefore, most users should pass an optimizer to `Model` because it provides the earliest warning that your solver is not suitable for the model you are trying to build. However, if you are modifying a problem by adding and deleting different constraint types, you may need to use `set_optimizer`. See Switching optimizer for the relaxed problem for an example of when this is useful.

Reducing time-to-first-solve latency

By default, JuMP uses bridges to reformulate the model you wrote into an equivalent model supported by the solver.

However, if your model is already supported by the solver, bridges add latency (read The "time-to-first-solve" issue). This is particularly noticeable for small models.

To reduce the "time-to-first-solve", try passing `add_bridges = false`.

```
julia> model = Model(GLPK.Optimizer; add_bridges = false);
```

or

```
julia> model = Model();

julia> set_optimizer(model, GLPK.Optimizer; add_bridges = false)
```

However, be wary! If your model and solver combination needs bridges, an error will be thrown:

```
julia> model = Model(SCS.Optimizer; add_bridges = false);


julia> @variable(model, x)
x

julia> @constraint(model, 2x <= 1)
ERROR: Constraints of type
↪  MathOptInterface.ScalarAffineFunction{Float64}-in-MathOptInterface.LessThan{Float64} are not
↪  supported by the solver.

If you expected the solver to support your problem, you may have an error in our formulation.
↪  Otherwise, consider using a different solver.

The list of available solvers, along with the problem types they support, is available at
↪  https://jump.dev/JuMP.jl/stable/installation/#Supported-solvers.
[...]
```

9.2 Solver options

Use `optimizer_with_attributes` to create an optimizer with some attributes initialized:

```
julia> model = Model(optimizer_with_attributes(GLPK.Optimizer, "msg_lev" => 0))
A JuMP Model
Feasibility problem with:
Variables: 0
Model mode: AUTOMATIC
CachingOptimizer state: EMPTY_OPTIMIZER
Solver name: GLPK
```

Alternatively, you can create a function which takes no arguments and returns an initialized `Optimizer` object:

```
julia> function my_optimizer()
           model = GLPK.Optimizer()
           MOI.set(model, MOI.RawOptimizerAttribute("msg_lev"), 0)
           return model
       end
my_optimizer (generic function with 1 method)

julia> model = Model(my_optimizer)
A JuMP Model
Feasibility problem with:
Variables: 0
Model mode: AUTOMATIC
CachingOptimizer state: EMPTY_OPTIMIZER
Solver name: GLPK
```

A third option is to use `set_optimizer_attribute`:

```
julia> model = Model(GLPK.Optimizer);

julia> set_optimizer_attribute(model, "msg_lev", 0)

julia> get_optimizer_attribute(model, "msg_lev")
0
```

9.3 Print the model

By default, `show(model)` will print a summary of the problem.

```
julia> model = Model(); @variable(model, x >= 0); @objective(model, Max, x);

julia> model
A JuMP Model
Maximization problem with:
Variable: 1
Objective function type: VariableRef
`VariableRef`-in-`MathOptInterface.GreaterThan{Float64}`: 1 constraint
Model mode: AUTOMATIC
CachingOptimizer state: NO_OPTIMIZER
Solver name: No optimizer attached.
Names registered in the model: x
```

Use `print` to print the formulation of the model (in IJulia, this will render as LaTeX.

```
julia> print(model)
Max x
Subject to
 x ≥ 0.0
```

Warning

This format is specific to JuMP. To write the model to a file, use `write_to_file` instead.

Use `latex_formulation` to display the model in LaTeX form.

```
julia> latex_formulation(model)
$$ \begin{aligned}
\max\quad & x\\
\text{Subject to} \quad & x \geq 0.0\\
\end{aligned} $$
```

In IJulia (and Documenter), ending a cell in with `latex_formulation` will render the model in LaTeX!

```
latex_formulation(model)
```

$$\begin{aligned}
\max\quad & x\\
\text{Subject to} \quad & x \geq 0.0\\
\end{aligned}$$

9.4 Turn off output

Use `set_silent` and `unset_silent` to disable or enable printing output from the solver.

```
julia> model = Model(GLPK.Optimizer);

julia> set_silent(model)


julia> unset_silent(model)
```

9.5 Set a time limit

Use `set_time_limit_sec`, `unset_time_limit_sec`, and `time_limit_sec` to manage time limits.

```
julia> model = Model(GLPK.Optimizer);

julia> set_time_limit_sec(model, 60.0)


julia> time_limit_sec(model)
60.0

julia> unset_time_limit_sec(model)

julia> time_limit_sec(model)
2.147483647e6
```

9.6 Write a model to file

JuMP can write models to a variety of file-formats using `write_to_file` and `Base.write`.

```
julia> write_to_file(model, "model.mps")

julia> write(io, model; format = MOI.FileFormats.FORMAT_MPS)
```

Info

The supported file formats are defined by the `MOI.FileFormats.FileFormat` enum.

9.7 Read a model from file

JuMP models can be created from file formats using `read_from_file` and `Base.read`.

```
julia> model = read_from_file("model.mps")
A JuMP Model
Minimization problem with:
Variables: 0
Objective function type: AffExpr
Model mode: AUTOMATIC
CachingOptimizer state: NO_OPTIMIZER
Solver name: No optimizer attached.

julia> seekstart(io);

julia> model2 = read(io, Model; format = MOI.FileFormats.FORMAT_MPS)
A JuMP Model
Minimization problem with:
Variables: 0
Objective function type: AffExpr
Model mode: AUTOMATIC
CachingOptimizer state: NO_OPTIMIZER
Solver name: No optimizer attached.
```

Note

Because file formats do not serialize the containers of JuMP variables and constraints, the names in the model will not be registered. Therefore, you cannot access named variables and constraints via `model[:x]`. Instead, use `variable_by_name` or `constraint_by_name` to access specific variables or constraints.

9.8 Relax integrality

Use `relax_integrality` to remove any integrality constraints from the model, such as integer and binary restrictions on variables. `relax_integrality` returns a function that can be later called with zero arguments in order to re-add the removed constraints:

```
julia> model = Model();

julia> @variable(model, x, Int)
x
```

```
julia> num_constraints(model, VariableRef, MOI.Integer)
1

julia> undo = relax_integrality(model);

julia> num_constraints(model, VariableRef, MOI.Integer)
0

julia> undo()

julia> num_constraints(model, VariableRef, MOI.Integer)
1
```

Switching optimizer for the relaxed problem

A common reason for relaxing integrality is to compute dual variables of the relaxed problem. However, some mixed-integer linear solvers (e.g., Cbc) do not return dual solutions, even if the problem does not have integrality restrictions.

Therefore, after `relax_integrality` you should call `set_optimizer` with a solver that does support dual solutions, such as Clp. For example:

```
using JuMP, Cbc
model = Model(Cbc.Optimizer)
@variable(model, x, Int)
undo = relax_integrality(model)
optimize!(model)
reduced_cost(x)  # Errors
```

```
using JuMP, Cbc, Clp
model = Model(Cbc.Optimizer)
@variable(model, x, Int)
undo = relax_integrality(model)

# Bad
optimize!(model)
has_duals(model)  # false

# Good
set_optimizer(model, Clp.Optimizer)
optimize!(model)
has_duals(model)  # true
```

9.9 Backends

A JuMP `Model` is a thin layer around a backend of type `MOI.ModelLike` that stores the optimization problem and acts as the optimization solver.

From JuMP, the MOI backend can be accessed using the `backend` function. Let's see what the `backend` of a JuMP `Model` is:

```
julia> model = Model(GLPK.Optimizer)
A JuMP Model
```

```
Feasibility problem with:
Variables: 0
Model mode: AUTOMATIC
CachingOptimizer state: EMPTY_OPTIMIZER
Solver name: GLPK

julia> b = backend(model)
MOIU.CachingOptimizer{MOIB.LazyBridgeOptimizer{GLPK.Optimizer},
↪  MOIU.UniversalFallback{MOIU.Model{Float64}}}
in state EMPTY_OPTIMIZER
in mode AUTOMATIC
with model cache MOIU.UniversalFallback{MOIU.Model{Float64}}
  fallback for MOIU.Model{Float64}
with optimizer MOIB.LazyBridgeOptimizer{GLPK.Optimizer}
  with 0 variable bridges
  with 0 constraint bridges
  with 0 objective bridges
  with inner model A GLPK model
```

The backend is a `MOIU.CachingOptimizer` in the state `EMPTY_OPTIMIZER` and mode `AUTOMATIC`.

Alternatively, use `unsafe_backend` to access the innermost `GLPK.Optimizer` object:

```
julia> unsafe_backend(model)
A GLPK model
```

> **Warning**
>
> `backend` and `unsafe_backend` are advanced routines. Read their docstrings to understand the caveats of their usage. You should only call them if you wish to access low-level solver-specific functions.

CachingOptimizer

A `MOIU.CachingOptimizer` is an MOI layer that abstracts the difference between solvers that support incremental modification (e.g., they support adding variables one-by-one), and solvers that require the entire problem in a single API call (e.g., they only accept the `A`, `b` and `c` matrices of a linear program).

It has two parts:

1. A cache, where the model can be built and modified incrementally

   ```
   julia> b.model_cache
   MOIU.UniversalFallback{MOIU.Model{Float64}}
   fallback for MOIU.Model{Float64}
   ```

2. An optimizer, which is used to solve the problem

   ```
   julia> b.optimizer
   MOIB.LazyBridgeOptimizer{GLPK.Optimizer}
   with 0 variable bridges
   with 0 constraint bridges
   with 0 objective bridges
   with inner model A GLPK model
   ```

Info

The LazyBridgeOptimizer section explains what a `LazyBridgeOptimizer` is.

The `CachingOptimizer` has logic to decide when to copy the problem from the cache to the optimizer, and when it can efficiently update the optimizer in-place.

A `CachingOptimizer` may be in one of three possible states:

- `NO_OPTIMIZER`: The CachingOptimizer does not have any optimizer.
- `EMPTY_OPTIMIZER`: The CachingOptimizer has an empty optimizer, and it is not synchronized with the cached model.
- `ATTACHED_OPTIMIZER`: The CachingOptimizer has an optimizer, and it is synchronized with the cached model.

A `CachingOptimizer` has two modes of operation:

- `AUTOMATIC`: The `CachingOptimizer` changes its state when necessary. For example, `optimize!` will automatically call `attach_optimizer` (an optimizer must have been previously set). Attempting to add a constraint or perform a modification not supported by the optimizer results in a drop to `EMPTY_OPTIMIZER` mode.
- `MANUAL`: The user must change the state of the `CachingOptimizer` using `MOIU.reset_optimizer(::JuMP.Model)`, `MOIU.drop_optimizer(::JuMP.Model)`, and `MOIU.attach_optimizer(::JuMP.Model)`. Attempting to perform an operation in the incorrect state results in an error.

By default `Model` will create a `CachingOptimizer` in `AUTOMATIC` mode.

LazyBridgeOptimizer

The second layer that JuMP applies automatically is a `LazyBridgeOptimizer`. A `LazyBridgeOptimizer` is an MOI layer that attempts to transform constraints added by the user into constraints supported by the solver. This may involve adding new variables and constraints to the optimizer. The transformations are selected from a set of known recipes called bridges.

A common example of a bridge is one that splits an interval constrait like `@constraint(model, 1 <= x + y <= 2)` into two constraints, `@constraint(model, x + y >= 1)` and `@constraint(model, x + y <= 2)`.

Use the `add_bridges = false` keyword to remove the bridging layer:

```
julia> model = Model(GLPK.Optimizer; add_bridges = false)
A JuMP Model
Feasibility problem with:
Variables: 0
Model mode: AUTOMATIC
CachingOptimizer state: EMPTY_OPTIMIZER
Solver name: GLPK

julia> backend(model)
MOIU.CachingOptimizer{GLPK.Optimizer, MOIU.UniversalFallback{MOIU.Model{Float64}}}
in state EMPTY_OPTIMIZER
in mode AUTOMATIC
with model cache MOIU.UniversalFallback{MOIU.Model{Float64}}
  fallback for MOIU.Model{Float64}
with optimizer A GLPK model
```

9.10 Direct mode

Using a `CachingOptimizer` results in an additional copy of the model being stored by JuMP in the `.model_cache` field. To avoid this overhead, create a JuMP model using `direct_model`:

```
julia> model = direct_model(GLPK.Optimizer())
A JuMP Model
Feasibility problem with:
Variables: 0
Model mode: DIRECT
Solver name: GLPK
```

> **Warning**
>
> Solvers that do not support incremental modification do not support `direct_model`. An error will be thrown, telling you to use a `CachingOptimizer` instead.

The benefit of using `direct_model` is that there are no extra layers (e.g., `Cachingoptimizer` or `LazyBridgeOptimizer`) between `model` and the provided optimizer:

```
julia> typeof(backend(model))
GLPK.Optimizer
```

A downside of direct mode is that there is no bridging layer. Therefore, only constraints which are natively supported by the solver are supported. For example, `GLPK.jl` does not implement constraints of the form `l <= a' x <= u`.

```
julia> @variable(model, x[1:2]);

julia> @constraint(model, 1 <= x[1] + x[2] <= 2)
ERROR: Constraints of type
↪  MathOptInterface.ScalarAffineFunction{Float64}-in-MathOptInterface.Interval{Float64} are not
↪  supported by the solver.

If you expected the solver to support your problem, you may have an error in our formulation.
↪  Otherwise, consider using a different solver.

The list of available solvers, along with the problem types they support, is available at
↪  https://jump.dev/JuMP.jl/stable/installation/#Supported-solvers.
[...]
```

Chapter 10

Variables

10.1 What is a JuMP variable?

The term variable in mathematical optimization has many meanings. Here, we distinguish between the following three types of variables:

1. optimization variables, which are the mathematical x in the problem $\max\{f_0(x)|f_i(x) \in S_i\}$.
2. Julia variables, which are bindings between a name and a value, for example `x = 1`. (See here for the Julia docs.)
3. JuMP variables, which are instances of the `VariableRef` struct defined by JuMP that contains a reference to an optimization variable in a model. (Extra for experts: the `VariableRef` struct is a thin wrapper around a `MOI.VariableIndex`, and also contains a reference to the JuMP model.)

To illustrate these three types of variables, consider the following JuMP code (the full syntax is explained below):

```
julia> model = Model()
A JuMP Model
Feasibility problem with:
Variables: 0
Model mode: AUTOMATIC
CachingOptimizer state: NO_OPTIMIZER
Solver name: No optimizer attached.

julia> @variable(model, x[1:2])
2-element Vector{VariableRef}:
 x[1]
 x[2]
```

This code does three things:

1. It adds two optimization variables to `model`.
2. It creates two JuMP variables that act as references to those optimization variables.
3. It binds those JuMP variables as a vector with two elements to the Julia variable `x`.

To reduce confusion, we will attempt, where possible, to always refer to variables with their corresponding prefix.

JuMP variables can have attributes, such as names or an initial primal start value. We illustrate the name attribute in the following example:

```
julia> @variable(model, y, base_name="decision variable")
decision variable
```

This code does four things:

1. It adds one optimization variable to `model`.
2. It creates one JuMP variable that acts as a reference to that optimization variable.
3. It binds the JuMP variable to the Julia variable `y`.
4. It tells JuMP that the name attribute of this JuMP variable is "decision variable". JuMP uses the value of `base_name` when it has to print the variable as a string.

For example, when we print `y` at the REPL we get:

```
julia> y
decision variable
```

Because `y` is a Julia variable, we can bind it to a different value. For example, if we write:

```
julia> y = 1
1
```

`y` is no longer a binding to a JuMP variable. This does not mean that the JuMP variable has been destroyed. It still exists and is still a reference to the same optimization variable. The binding can be reset by querying the model for the symbol as it was written in the `@variable` macro. For example:

```
julia> model[:y]
decision variable
```

This act of looking up the JuMP variable by using the symbol is most useful when composing JuMP models across multiple functions, as illustrated by the following example:

```
function add_component_to_model(model::JuMP.Model)
    x = model[:x]
    # ... code that uses x
end
function build_model()
    model = Model()
    @variable(model, x)
    add_component_to_model(model)
end
```

Now that we understand the difference between optimization, JuMP, and Julia variables, we can introduce more of the functionality of the `@variable` macro.

10.2 Variable bounds

We have already seen the basic usage of the `@variable` macro. The next extension is to add lower- and upper-bounds to each optimization variable. This can be done as follows:

```
julia> @variable(model, x_free)
x_free

julia> @variable(model, x_lower >= 0)
x_lower

julia> @variable(model, x_upper <= 1)
x_upper

julia> @variable(model, 2 <= x_interval <= 3)
x_interval

julia> @variable(model, x_fixed == 4)
x_fixed
```

In the above examples, `x_free` represents an unbounded optimization variable, `x_lower` represents an optimization variable with a lower bound and so forth.

> **Warning**
>
> When creating a variable with only a lower-bound or an upper-bound, and the value of the bound is not a numeric literal (e.g., `1` or `1.0`), the name of the variable must appear on the left-hand side. Putting the name on the right-hand side will result in an error. For example to create a variable `x`:
>
> ```
> a = 1
> @variable(model, x >= 1)      # ✓ Okay
> @variable(model, 1.0 <= x)    # ✓ Okay
> @variable(model, x >= a)      # ✓ Okay
> @variable(model, a <= x)      # × Not okay
> @variable(model, x >= 1 / 2)  # ✓ Okay
> @variable(model, 1 / 2 <= x)  # × Not okay
> ```

Check if a variable bound exists

We can query whether an optimization variable has a lower- or upper-bound via the `has_lower_bound` and `has_upper_bound` functions. For example:

```
julia> has_lower_bound(x_free)
false

julia> has_upper_bound(x_upper)
true
```

Query a variable bound

If a variable has a lower or upper bound, we can query the value of it via the `lower_bound` and `upper_bound` functions. For example:

```
julia> lower_bound(x_interval)
2.0

julia> upper_bound(x_interval)
3.0
```

Querying the value of a bound that does not exist will result in an error.

Set variable bounds via keyword

Instead of using the <= and >= syntax, we can also use the `lower_bound` and `upper_bound` keyword arguments. For example:

```
julia> @variable(model, x, lower_bound=1, upper_bound=2)
x

julia> lower_bound(x)
1.0
```

Set variable bounds bounds via functions

Another option is to use the `set_lower_bound` and `set_upper_bound` functions. These can also be used to modify an existing variable bound. For example:

```
julia> @variable(model, x >= 1)
x

julia> lower_bound(x)
1.0

julia> set_lower_bound(x, 2)

julia> lower_bound(x)
2.0
```

Delete a variable bound

We can delete variable bounds using `delete_lower_bound` and `delete_upper_bound`:

```
julia> @variable(model, 1 <= x <= 2)
x

julia> lower_bound(x)
1.0

julia> delete_lower_bound(x)

julia> has_lower_bound(x)
false

julia> upper_bound(x)
2.0

julia> delete_upper_bound(x)
```

```
julia> has_upper_bound(x)
false
```

Create a fixed variable

In addition to upper and lower bounds, JuMP variables can also be fixed to a value using `fix`. See also `is_fixed`, `fix_value`, and `unfix`.

```
julia> @variable(model, x == 1)
x

julia> is_fixed(x)
true

julia> fix_value(x)
1.0

julia> unfix(x)

julia> is_fixed(x)
false
```

Fixing a variable with existing bounds will throw an error. To delete the bounds prior to fixing, use `fix(variable, value; force = true)`.

```
julia> @variable(model, x >= 1)
x

julia> fix(x, 2)
ERROR: Unable to fix x to 2 because it has existing variable bounds. Consider calling
↪  `JuMP.fix(variable, value; force=true)` which will delete existing bounds before fixing the
↪  variable.

julia> fix(x, 2; force = true)


julia> fix_value(x)
2.0
```

> **Tip**
>
> Use `fix` instead of `@constraint(model, x == 2)`. The former modifies variable bounds, while the latter adds a new linear constraint to the problem.

10.3 Variable names

The name, i.e. the value of the `MOI.VariableName` attribute, of a variable can be obtained by `JuMP.name(::JuMP.VariableRef)` and set by `JuMP.set_name(::JuMP.VariableRef, ::String)`.

```
julia> model = Model();

julia> @variable(model, x)
```

```
x

julia> name(x)
"x"

julia> set_name(x, "my_x_name")

julia> x
my_x_name
```

Specify a name in the macro via `base_name`:

```
julia> model = Model();

julia> x = @variable(model, [i=1:2], base_name = "my_var")
2-element Vector{VariableRef}:
 my_var[1]
 my_var[2]
```

Note that names apply to each element of the container, not to the container of variables:

```
julia> name(x[1])
"my_var[1]"

julia> set_name(x[1], "my_x")

julia> x
2-element Vector{VariableRef}:
 my_x
 my_var[2]
```

Retrieve a variable by name

Retrieve a variable from a model using `variable_by_name`:

```
julia> variable_by_name(model, "my_x")
my_x
```

If the name is not present, `nothing` will be returned:

```
julia> variable_by_name(model, "bad_name")
```

You can only look up invididual variables using `variable_by_name`. Something like this will not work:

```
julia> model = Model();

julia> @variable(model, [i = 1:2], base_name = "my_var")
2-element Vector{VariableRef}:
 my_var[1]
 my_var[2]

julia> variable_by_name(model, "my_var")
```

To look up a collection of variables, do not use `variable_by_name`. Instead, register them using the `model[:key] = value` syntax:

```
julia> model = Model();

julia> model[:x] = @variable(model, [i = 1:2], base_name = "my_var")
2-element Vector{VariableRef}:
 my_var[1]
 my_var[2]

julia> model[:x]
2-element Vector{VariableRef}:
 my_var[1]
 my_var[2]
```

10.4 Variable containers

In the examples above, we have mostly created scalar variables. By scalar, we mean that the Julia variable is bound to exactly one JuMP variable. However, it is often useful to create collections of JuMP variables inside more complicated data structures.

JuMP provides a mechanism for creating three types of these data structures, which we refer to as containers. The three types are `Array`s, `DenseAxisArray`s, and `SparseAxisArray`s. We explain each of these in the following.

> **Tip**
>
> You can read more about containers in the Containers section.

Arrays

We have already seen the creation of an array of JuMP variables with the `x[1:2]` syntax. This can naturally be extended to create multi-dimensional arrays of JuMP variables. For example:

```
julia> @variable(model, x[1:2, 1:2])
2×2 Matrix{VariableRef}:
 x[1,1]  x[1,2]
 x[2,1]  x[2,2]
```

Arrays of JuMP variables can be indexed and sliced as follows:

```
julia> x[1, 2]
x[1,2]

julia> x[2, :]
2-element Vector{VariableRef}:
 x[2,1]
 x[2,2]
```

Variable bounds can depend upon the indices:

```
julia> @variable(model, x[i=1:2, j=1:2] >= 2i + j)
2×2 Matrix{VariableRef}:
 x[1,1]  x[1,2]
 x[2,1]  x[2,2]

julia> lower_bound.(x)
2×2 Matrix{Float64}:
 3.0  4.0
 5.0  6.0
```

JuMP will form an `Array` of JuMP variables when it can determine at compile time that the indices are one-based integer ranges. Therefore `x[1:b]` will create an `Array` of JuMP variables, but `x[a:b]` will not. If JuMP cannot determine that the indices are one-based integer ranges (e.g., in the case of `x[a:b]`), JuMP will create a `DenseAxisArray` instead.

DenseAxisArrays

We often want to create arrays where the indices are not one-based integer ranges. For example, we may want to create a variable indexed by the name of a product or a location. The syntax is the same as that above, except with an arbitrary vector as an index as opposed to a one-based range. The biggest difference is that instead of returning an `Array` of JuMP variables, JuMP will return a `DenseAxisArray`. For example:

```
julia> @variable(model, x[1:2, [:A,:B]])
2-dimensional DenseAxisArray{VariableRef,2,...} with index sets:
    Dimension 1, Base.OneTo(2)
    Dimension 2, [:A, :B]
And data, a 2×2 Matrix{VariableRef}:
 x[1,A]  x[1,B]
 x[2,A]  x[2,B]
```

DenseAxisArrays can be indexed and sliced as follows:

```
julia> x[1, :A]
x[1,A]

julia> x[2, :]
1-dimensional DenseAxisArray{VariableRef,1,...} with index sets:
    Dimension 1, [:A, :B]
And data, a 2-element Vector{VariableRef}:
 x[2,A]
 x[2,B]
```

Similarly to the `Array` case, bounds can depend upon indices. For example:

```
julia> @variable(model, x[i=2:3, j=1:2:3] >= 0.5i + j)
2-dimensional DenseAxisArray{VariableRef,2,...} with index sets:
    Dimension 1, 2:3
    Dimension 2, 1:2:3
And data, a 2×2 Matrix{VariableRef}:
 x[2,1]  x[2,3]
 x[3,1]  x[3,3]

julia> lower_bound.(x)
```

```
2-dimensional DenseAxisArray{Float64,2,...} with index sets:
    Dimension 1, 2:3
    Dimension 2, 1:2:3
And data, a 2×2 Matrix{Float64}:
 2.0  4.0
 2.5  4.5
```

SparseAxisArrays

The third container type that JuMP natively supports is `SparseAxisArray`. These arrays are created when the indices do not form a rectangular set. For example, this applies when indices have a dependence upon previous indices (called triangular indexing). JuMP supports this as follows:

```
julia> @variable(model, x[i=1:2, j=i:2])
JuMP.Containers.SparseAxisArray{VariableRef, 2, Tuple{Int64, Int64}} with 3 entries:
  [1, 1]  =  x[1,1]
  [1, 2]  =  x[1,2]
  [2, 2]  =  x[2,2]
```

We can also conditionally create variables via a JuMP-specific syntax. This syntax appends a comparison check that depends upon the named indices and is separated from the indices by a semi-colon (`;`). For example:

```
julia> @variable(model, x[i=1:4; mod(i, 2)==0])
JuMP.Containers.SparseAxisArray{VariableRef, 1, Tuple{Int64}} with 2 entries:
  [2]  =  x[2]
  [4]  =  x[4]
```

Note that with many index dimensions and a large amount of sparsity, variable construction may be unnecessarily slow if the semi-colon syntax is naively applied. When using the semi-colon as a filter, JuMP iterates over all indices and evaluates the conditional for each combination. When this is undesired, the recommended work-around is to work directly with a list of tuples or create a dictionary. Consider the following examples:

```
N = 10
S = [(1, 1, 1),(N, N, N)]
# Slow. It evaluates conditional N^3 times.
@variable(model, x1[i=1:N, j=1:N, k=1:N; (i, j, k) in S])
# Fast.
@variable(model, x2[S])
# Fast. Manually constructs a dictionary and fills it.
x3 = Dict()
for (i, j, k) in S
    x3[i, j, k] = @variable(model)
    # Optional, if you care about pretty printing:
    set_name(x3[i, j, k], "x[$i,$j,$k]")
end
```

Forcing the container type

When creating a container of JuMP variables, JuMP will attempt to choose the tightest container type that can store the JuMP variables. Thus, it will prefer to create an Array before a DenseAxisArray and a DenseAxisArray before a SparseAxisArray. However, because this happens at compile time, it does not always make the best choice. To illustrate this, consider the following example:

```
julia> A = 1:2
1:2

julia> @variable(model, x[A])
1-dimensional DenseAxisArray{VariableRef,1,...} with index sets:
    Dimension 1, 1:2
And data, a 2-element Vector{VariableRef}:
 x[1]
 x[2]
```

Since the value (and type) of `A` is unknown at parsing time, JuMP is unable to infer that `A` is a one-based integer range. Therefore, JuMP creates a `DenseAxisArray`, even though it could store these two variables in a standard one-dimensional `Array`.

We can share our knowledge that it is possible to store these JuMP variables as an array by setting the `container` keyword:

```
julia> @variable(model, y[A], container=Array)
2-element Vector{VariableRef}:
 y[1]
 y[2]
```

JuMP now creates a vector of JuMP variables instead of a DenseAxisArray. Note that choosing an invalid container type will throw an error.

User-defined containers

> **Tip**
>
> This is a point that users often overlook: you are not restricted to the built-in container types in JuMP.

In addition to the built-in container types, you can create your own collections of JuMP variables.

For example, the following code creates a dictionary with symmetric matrices as the values:

```
julia> variables = Dict{Symbol,Array{VariableRef,2}}(
           key => @variable(model, [1:2, 1:2], Symmetric, base_name = "$(key)")
           for key in [:A, :B]
       )
Dict{Symbol, Matrix{VariableRef}} with 2 entries:
  :A => [A[1,1] A[1,2]; A[1,2] A[2,2]]
  :B => [B[1,1] B[1,2]; B[1,2] B[2,2]]
```

Another common scenario is a request to add variables to existing containers, for example:

```
using JuMP
model = Model()
@variable(model, x[1:2] >= 0)
# Later I want to add
@variable(model, x[3:4] >= 0)
```

This is not possible with the built-in JuMP container types. However, you can use regular Julia types instead:

```
model = Model()
x = model[:x] = @variable(model, [1:2], lower_bound = 0, base_name = "x")
append!(x, @variable(model, [1:2], lower_bound = 0, base_name = "y"))
model[:x]

# output

4-element Vector{VariableRef}:
 x[1]
 x[2]
 y[1]
 y[2]
```

10.5 Integrality utilities

Adding integrality constraints to a model such as `@constraint(model, x in MOI.ZeroOne())` and `@constraint(model, x in MOI.Integer())` is a common operation. Therefore, JuMP supports two shortcuts for adding such constraints.

Binary (ZeroOne) constraints

Binary optimization variables are constrained to the set $x \in \{0, 1\}$. (The `MOI.ZeroOne` set in MathOptInterface.) Binary optimization variables can be created in JuMP by passing `Bin` as an optional positional argument:

```
julia> @variable(model, x, Bin)
x
```

We can check if an optimization variable is binary by calling `is_binary` on the JuMP variable, and binary constraints can be removed with `unset_binary`.

```
julia> is_binary(x)
true

julia> unset_binary(x)

julia> is_binary(x)
false
```

Binary optimization variables can also be created by setting the `binary` keyword to `true`.

```
julia> @variable(model, x, binary=true)
x
```

Integer constraints

Integer optimization variables are constrained to the set $x \in \mathbb{Z}$. (The `MOI.Integer` set in MathOptInterface.) Integer optimization variables can be created in JuMP by passing `Int` as an optional positional argument:

```
julia> @variable(model, x, Int)
x
```

Integer optimization variables can also be created by setting the `integer` keyword to `true`.

```
julia> @variable(model, x, integer=true)
x
```

We can check if an optimization variable is integer by calling `is_integer` on the JuMP variable, and integer constraints can be removed with `unset_integer`.

```
julia> is_integer(x)
true

julia> unset_integer(x)

julia> is_integer(x)
false
```

> **Tip**
>
> The `relax_integrality` function relaxes all integrality constraints in the model, returning a function that can be called to undo the operation later on.

10.6 Semidefinite variables

JuMP also supports modeling with semidefinite variables. A square symmetric matrix X is positive semidefinite if all eigenvalues are nonnegative. We can declare a matrix of JuMP variables to be positive semidefinite as follows:

```
julia> @variable(model, x[1:2, 1:2], PSD)
2×2 LinearAlgebra.Symmetric{VariableRef, Matrix{VariableRef}}:
 x[1,1]  x[1,2]
 x[1,2]  x[2,2]
```

or using the syntax for Variables constrained on creation:

```
julia> @variable(model, x[1:2, 1:2] in PSDCone())
2×2 LinearAlgebra.Symmetric{VariableRef, Matrix{VariableRef}}:
 x[1,1]  x[1,2]
 x[1,2]  x[2,2]
```

Note that `x` must be a square 2-dimensional `Array` of JuMP variables; it cannot be a DenseAxisArray or a SparseAxisArray. (See Variable containers, above, for more on this.)

You can also impose a weaker constraint that the square matrix is only symmetric (instead of positive semidefinite) as follows:

```
julia> @variable(model, x[1:2, 1:2], Symmetric)
2×2 LinearAlgebra.Symmetric{VariableRef, Matrix{VariableRef}}:
 x[1,1]  x[1,2]
 x[1,2]  x[2,2]
```

You can impose a constraint that the square matrix is skew symmetric with `SkewSymmetricMatrixSpace`:

```
julia> @variable(model, x[1:2, 1:2] in SkewSymmetricMatrixSpace())
2×2 Matrix{AffExpr}:
 0        x[1,2]
 -x[1,2]  0
```

10.7 Anonymous JuMP variables

In all of the above examples, we have created named JuMP variables. However, it is also possible to create so called anonymous JuMP variables. To create an anonymous JuMP variable, we drop the name of the variable from the macro call. This means dropping the second positional argument if the JuMP variable is a scalar, or dropping the name before the square bracket (`[`) if a container is being created. For example:

```
julia> x = @variable(model)
noname
```

This shows how `@variable(model, x)` is really short for:

```
julia> x = model[:x] = @variable(model, base_name="x")
x
```

An `Array` of anonymous JuMP variables can be created as follows:

```
julia> y = @variable(model, [i=1:2])
2-element Vector{VariableRef}:
 noname
 noname
```

If necessary, you can store `x` in `model` as follows:

```
julia> model[:x] = x
```

The `<=` and `>=` short-hand cannot be used to set bounds on anonymous JuMP variables. Instead, you should use the `lower_bound` and `upper_bound` keywords.

Passing the `Bin` and `Int` variable types are also invalid. Instead, you should use the `binary` and `integer` keywords.

Thus, the anonymous variant of `@variable(model, x[i=1:2] >= i, Int)` is:

```
julia> x = @variable(model, [i=1:2], base_name="x", lower_bound=i, integer=true)
2-element Vector{VariableRef}:
 x[1]
 x[2]
```

> **Warning**
>
> Creating two named JuMP variables with the same name results in an error at runtime. Use anonymous variables as an alternative.

10.8 Variables constrained on creation

> **Info**
>
> When using JuMP in Direct mode, it may be required to constrain variables on creation instead of constraining free variables as the solver may only support variables constrained on creation. In Automatic and Manual modes, both ways of adding constraints on variables are equivalent. Indeed, during the copy of the cache to the optimizer, the choice of the constraints on variables that are copied as variables constrained on creation does not depend on how it was added to the cache.

All uses of the `@variable` macro documented so far translate to a separate call for variable creation and adding of constraints.

For example, `@variable(model, x >= 0, Int)`, is equivalent to:

```
@variable(model, x)
set_lower_bound(x, 0.0)
@constraint(model, x in MOI.Integer())
```

Importantly, the bound and integrality constraints are added after the variable has been created.

However, some solvers require a constraining set at creation time. We say that these variables are constrained on creation.

Use `in` within `@variable` to access the special syntax for constraining variables on creation. For example, the following creates a vector of variables constrained on creation to belong to the `SecondOrderCone`:

```
julia> @variable(model, y[1:3] in SecondOrderCone())
3-element Vector{VariableRef}:
 y[1]
 y[2]
 y[3]
```

For contrast, the more standard approach is as follows:

```
julia> @variable(model, x[1:3])
3-element Vector{VariableRef}:
 x[1]
 x[2]
 x[3]

julia> @constraint(model, x in SecondOrderCone())
[x[1], x[2], x[3]] ∈ MathOptInterface.SecondOrderCone(3)
```

The technical difference between the former and the latter is that the former calls `MOI.add_constrained_variables` while the latter calls `MOI.add_variables` and then `MOI.add_constraint`. This distinction is important only in Direct mode, depending on the solver being used. It's often not possible to delete the `SecondOrderCone` constraint if it was specified at variable creation time.

The `set` keyword

An alternate syntax to `x in Set` is to use the `set` keyword of `@variable`. This is most useful when creating anonymous variables:

```
x = @variable(model, [1:2, 1:2], set = PSDCone())
```

10.9 Delete a variable

Use `delete` to delete a variable from a model. Use `is_valid` to check if a variable belongs to a model and has not been deleted.

```
julia> @variable(model, x)
x

julia> is_valid(model, x)
true

julia> delete(model, x)

julia> is_valid(model, x)
false
```

Deleting a variable does not unregister the symbolic reference from the model. Therefore, creating a new variable of the same name will throw an error:

```
julia> @variable(model, x)
ERROR: An object of name x is already attached to this model. If this
    is intended, consider using the anonymous construction syntax, e.g.,
    `x = @variable(model, [1:N], ...)` where the name of the object does
    not appear inside the macro.

    Alternatively, use `unregister(model, :x)` to first unregister
    the existing name from the model. Note that this will not delete the
    object; it will just remove the reference at `model[:x]`.
[...]
```

After calling `delete`, call `unregister` to remove the symbolic reference:

```
julia> unregister(model, :x)

julia> @variable(model, x)
x
```

Info

`delete` does not automatically `unregister` because we do not distinguish between names that are automatically registered by JuMP macros, and names that are manually registered by the user by setting values in `object_dictionary`. In addition, deleting a variable and then adding a new variable of the same name is an easy way to introduce bugs into your code.

10.10 Listing all variables

Use `JuMP.all_variables` to obtain a list of all variables present in the model. This is useful for performing operations like:

- relaxing all integrality constraints in the model
- setting the starting values for variables to the result of the last solve

10.11 Start values

There are two ways to provide a primal starting solution (also called MIP-start or a warmstart) for each variable:

- using the `start` keyword in the `@variable` macro
- using `set_start_value`

The starting value of a variable can be queried using `start_value`. If no start value has been set, `start_value` will return `nothing`.

```
julia> @variable(model, x)
x

julia> start_value(x)

julia> @variable(model, y, start = 1)
y

julia> start_value(y)
1.0

julia> set_start_value(y, 2)

julia> start_value(y)
2.0
```

Warning

Some solvers do not support start values. If a solver does not support start values, an `MathOptInterface.UnsupportedAttr` error will be thrown.

Note

Prior to JuMP 0.19, the previous solution to a solve was automatically set as the new starting value. JuMP 0.19 no longer does this automatically. To reproduce the functionality, use:

```
set_start_value.(all_variables(model), value.(all_variables(model)))
```

10.12 The `@variables` macro

If you have many `@variable` calls, JuMP provides the macro `@variables` that can improve readability:

```
julia> @variables(model, begin
           x
           y[i=1:2] >= i, (start = i, base_name = "Y_$i")
           z, Bin
       end)

julia> print(model)
Feasibility
Subject to
 Y_1[1] ≥ 1.0
 Y_2[2] ≥ 2.0
 z binary
```

Note

Keyword arguments must be contained within parentheses. (See the example above.)

Chapter 11

Expressions

JuMP has three types of expressions: affine, quadratic, and nonlinear. These expressions can be inserted into constraints or into the objective. This is particularly useful if an expression is used in multiple places in the model.

11.1 Affine expressions

There are four ways of constructing an affine expression in JuMP: with the `@expression` macro, with operator overloading, with the `AffExpr` constructor, and with `add_to_expression!`.

Macros

The recommended way to create an affine expression is via the `@expression` macro.

```
model = Model()
@variable(model, x)
@variable(model, y)
ex = @expression(model, 2x + y - 1)

# output

2 x + y - 1
```

This expression can be used in the objective or added to a constraint. For example:

```
@objective(model, Min, 2 * ex - 1)
objective_function(model)

# output

4 x + 2 y - 3
```

Just like variables and constraints, named expressions can also be created. For example

```
model = Model()
@variable(model, x[i = 1:3])
@expression(model, expr[i = 1:3], i * sum(x[j] for j in i:3))
expr
```

```
# output

3-element Vector{AffExpr}:
 x[1] + x[2] + x[3]
 2 x[2] + 2 x[3]
 3 x[3]
```

Tip

You can read more about containers in the Containers section.

Operator overloading

Expressions can also be created without macros. However, note that in some cases, this can be much slower that constructing an expression using macros.

```
model = Model()
@variable(model, x)
@variable(model, y)
ex = 2x + y - 1

# output

2 x + y - 1
```

Constructors

A third way to create an affine expression is by the `AffExpr` constructor. The first argument is the constant term, and the remaining arguments are variable-coefficient pairs.

```
model = Model()
@variable(model, x)
@variable(model, y)
ex = AffExpr(-1.0, x => 2.0, y => 1.0)

# output

2 x + y - 1
```

`add_to_expression!`

The fourth way to create an affine expression is by using `add_to_expression!`. Compared to the operator overloading method, this approach is faster because it avoids constructing temporary objects. The `@expression` macro uses `add_to_expression!` behind-the-scenes.

```
model = Model()
@variable(model, x)
@variable(model, y)
ex = AffExpr(-1.0)
add_to_expression!(ex, 2.0, x)
add_to_expression!(ex, 1.0, y)

# output
```

```
2 x + y - 1
```

Warning

Read the section Initializing arrays for some cases to be careful about when using `add_to_expression!`.

Removing zero terms

Use `drop_zeros!` to remove terms from an affine expression with a `0` coefficient.

```
julia> model = Model();

julia> @variable(model, x)
x

julia> @expression(model, ex, x + 1 - x)
0 x + 1

julia> drop_zeros!(ex)

julia> ex
1
```

Coefficients

Use `coefficient` to return the coefficient associated with a variable in an affine expression.

```
julia> model = Model();

julia> @variable(model, x)
x

julia> @variable(model, y)
y

julia> @expression(model, ex, 2x + 1)
2 x + 1

julia> coefficient(ex, x)
2.0

julia> coefficient(ex, y)
0.0
```

11.2 Quadratic expressions

Like affine expressions, there are four ways of constructing a quadratic expression in JuMP: macros, operator overloading, constructors, and `add_to_expression!`.

Macros

The `@expression` macro can be used to create quadratic expressions by including quadratic terms.

```
model = Model()
@variable(model, x)
@variable(model, y)
ex = @expression(model, x^2 + 2 * x * y + y^2 + x + y - 1)

# output

x² + 2 y*x + y² + x + y - 1
```

Operator overloading

Operator overloading can also be used to create quadratic expressions. The same performance warning (discussed in the affine expression section) applies.

```
model = Model()
@variable(model, x)
@variable(model, y)
ex = x^2 + 2 * x * y + y^2 + x + y - 1

# output

x² + 2 x*y + y² + x + y - 1
```

Constructors

Quadratic expressions can also be created using the `QuadExpr` constructor. The first argument is an affine expression, and the remaining arguments are pairs, where the first term is a `JuMP.UnorderedPair` and the second term is the coefficient.

```
model = Model()
@variable(model, x)
@variable(model, y)
aff_expr = AffExpr(-1.0, x => 1.0, y => 1.0)
quad_expr = QuadExpr(aff_expr, UnorderedPair(x, x) => 1.0,
                     UnorderedPair(x, y) => 2.0, UnorderedPair(y, y) => 1.0)

# output

x² + 2 x*y + y² + x + y - 1
```

`add_to_expression!`

Finally, `add_to_expression!` can also be used to add quadratic terms.

```
model = Model()
@variable(model, x)
@variable(model, y)
ex = QuadExpr(x + y - 1.0)
add_to_expression!(ex, 1.0, x, x)
add_to_expression!(ex, 2.0, x, y)
add_to_expression!(ex, 1.0, y, y)

# output

x² + 2 x*y + y² + x + y - 1
```

Warning

Read the section Initializing arrays for some cases to be careful about when using `add_to_expression!`.

Removing zero terms

Use `drop_zeros!` to remove terms from a quadratic expression with a `0` coefficient.

```
julia> model = Model();

julia> @variable(model, x)
x

julia> @expression(model, ex, x^2 + x + 1 - x^2)
0 x² + x + 1

julia> drop_zeros!(ex)

julia> ex
x + 1
```

Coefficients

Use `coefficient` to return the coefficient associated with a pair of variables in a quadratic expression.

```
julia> model = Model();

julia> @variable(model, x)
x

julia> @variable(model, y)
y

julia> @expression(model, ex, 2*x*y + 3*x)
2 x*y + 3 x

julia> coefficient(ex, x, y)
2.0

julia> coefficient(ex, x, x)
0.0

julia> coefficient(ex, y, x)
2.0

julia> coefficient(ex, x)
3.0
```

11.3 Nonlinear expressions

Nonlinear expressions can be constructed only using the `@NLexpression` macro and can be used only in `@NLobjective`, `@NLconstraint`, and other `@NLexpression`s. Moreover, quadratic and affine expressions cannot be used in the nonlinear macros. For more details, see the Nonlinear Modeling section.

11.4 Initializing arrays

JuMP implements `zero(AffExpr)` and `one(AffExpr)` in order to support various functions in `LinearAlgebra` (e.g., accessing the off-diagonal of a `Diagonal` matrix).

```
julia> zero(AffExpr)
0

julia> one(AffExpr)
1
```

However, this can result in a subtle bug if you call `add_to_expression!` or the MutableArithmetics API on an element created by `zeros` or `ones`:

```
julia> x = zeros(AffExpr, 2)
2-element Vector{AffExpr}:
 0
 0

julia> add_to_expression!(x[1], 1.1)
1.1

julia> x
2-element Vector{AffExpr}:
 1.1
 1.1
```

Notice how we modified `x[1]`, but we also changed `x[2]`!

This happened because `zeros(AffExpr, 2)` calls `zero(AffExpr)` once to obtain a zero element, and then creates an appropriately sized array filled with the same element.

This also happens with broadcasting calls containing a conversion of `0` or `1`:

```
julia> x = Vector{AffExpr}(undef, 2)
2-element Vector{AffExpr}:
 #undef
 #undef

julia> x .= 0
2-element Vector{AffExpr}:
 0
 0

julia> add_to_expression!(x[1], 1.1)
1.1

julia> x
2-element Vector{AffExpr}:
 1.1
 1.1
```

The recommended way to create an array of empty expressions is as follows:

```
julia> x = Vector{AffExpr}(undef, 2)
2-element Vector{AffExpr}:
 #undef
 #undef

julia> for i in eachindex(x)
           x[i] = AffExpr(0.0)
       end

julia> add_to_expression!(x[1], 1.1)
1.1

julia> x
2-element Vector{AffExpr}:
 1.1
 0
```

Alternatively, use non-mutating operation to avoid updating `x[1]` in-place:

```
julia> x = zeros(AffExpr, 2)
2-element Vector{AffExpr}:
 0
 0

julia> x[1] += 1.1
1.1

julia> x
2-element Vector{AffExpr}:
 1.1
 0
```

Note that for large expressions this will be slower due to the allocation of additional temporary objects.

Chapter 12

Objectives

This page describes macros and functions related to linear and quadratic objective functions only, unless otherwise indicated. For nonlinear objective functions, see Nonlinear Modeling.

12.1 Set a linear objective

Use the `@objective` macro to set a linear objective function.

Use `Min` to create a minimization objective:

```
julia> @objective(model, Min, 2x + 1)
2 x + 1
```

Use `Max` to create a maximization objective:

```
julia> @objective(model, Max, 2x + 1)
2 x + 1
```

12.2 Set a quadratic objective

Use the `@objective` macro to set a quadratic objective function.

Use `^2` to have a variable squared:

```
julia> @objective(model, Min, x^2 + 2x + 1)
x² + 2 x + 1
```

You can also have bilinear terms between variables:

```
julia> @variable(model, x)
x

julia> @variable(model, y)
y

julia> @objective(model, Max, x * y + x + y)
x*y + x + y
```

12.3 Query the objective function

Use `objective_function` to return the current objective function.

```
julia> @objective(model, Min, 2x + 1)
2 x + 1

julia> objective_function(model)
2 x + 1
```

12.4 Evaluate the objective function at a point

Use `value` to evaluate an objective function at a point specifying values for variables.

```
julia> @variable(model, x[1:2]);

julia> @objective(model, Min, 2x[1]^2 + x[1] + 0.5*x[2])
2 x[1]² + x[1] + 0.5 x[2]

julia> f = objective_function(model)
2 x[1]² + x[1] + 0.5 x[2]

julia> point = Dict(x[1] => 2.0, x[2] => 1.0);

julia> value(z -> point[z], f)
10.5
```

12.5 Query the objective sense

Use `objective_sense` to return the current objective sense.

```
julia> @objective(model, Min, 2x + 1)
2 x + 1

julia> objective_sense(model)
MIN_SENSE::OptimizationSense = 0
```

12.6 Modify an objective

To modify an objective, call `@objective` with the new objective function.

```
julia> @objective(model, Min, 2x)
2 x

julia> @objective(model, Max, -2x)
-2 x
```

Alternatively, use `set_objective_function`.

```
julia> @objective(model, Min, 2x)
2 x
```

```
julia> new_objective = @expression(model, -2 * x)
-2 x

julia> set_objective_function(model, new_objective)
```

12.7 Modify an objective coefficient

Use `set_objective_coefficient` to modify an objective coefficient.

```
julia> @objective(model, Min, 2x)
2 x

julia> set_objective_coefficient(model, x, 3)

julia> objective_function(model)
3 x
```

> **Info**
>
> There is no way to modify the coefficient of a quadratic term. Set a new objective instead.

12.8 Modify the objective sense

Use `set_objective_sense` to modify the objective sense.

```
julia> @objective(model, Min, 2x)
2 x

julia> objective_sense(model)
MIN_SENSE::OptimizationSense = 0

julia> set_objective_sense(model, MOI.MAX_SENSE);

julia> objective_sense(model)
MAX_SENSE::OptimizationSense = 1
```

Alternatively, call `@objective` and pass the existing objective function.

```
julia> @objective(model, Min, 2x)
2 x

julia> @objective(model, Max, objective_function(model))
2 x
```

Chapter 13

Constraints

This page explains how to write various types of constraints in JuMP. Before reading further, please make sure you are familiar with JuMP models, and JuMP Variables. For nonlinear constraints, see Nonlinear Modeling instead.

JuMP is based on the MathOptInterface (MOI) API. Because of this, JuMP thinks of a constraint as the restriction that the output of a function belongs to a set. For example, instead of representing a constraint $a^\top x \leq b$ as a less-than-or-equal-to constraint, JuMP models this as the scalar affine function $a^\top x$ belonging to the less-than set $(-\infty, b]$. Thus, instead of a less-than-or-equal-to constraint, we consider this constraint to be a scalar affine -in- less than constraint. More generally, we use the shorthand function-in-set to refer to constraints composed of different types of functions and sets. In the rest of this page, we will introduce the different types of functions and sets that JuMP knows about as needed. You can read more details about this function-in-set concept in the MOI documentation.

> **Note**
>
> The examples use `MOI` as an alias for the `MathOptInterface` module. This alias is defined by `using JuMP`. You may also define it in your code by
>
> ```
> import MathOptInterface
> const MOI = MathOptInterface
> ```

13.1 The `@constraint` macro

Constraints are added to a JuMP model using the `@constraint` macro. Here is an example of how to add the constraint $2x \leq 1$ to a JuMP model:

```
julia> @constraint(model, con, 2x <= 1)
con : 2 x <= 1.0
```

Wasn't that easy! Let's unpack what happened, because just like `@variable` there are a few subtle things going on.

1. The mathematical constraint $2x \leq 1$ was added to the model.
2. A Julia variable called `con` was created that is a reference to the constraint.
3. This Julia variable was stored in `model` and can be accessed by `model[:con]`.

4. JuMP set the name attribute (the one that is shown when printing) of the constraint to `"con"`.

Just like the Julia variables created in `@variable`, `con` can be bound to a different value. For example:

```
julia> con
con : 2 x <= 1.0

julia> con = 1
1

julia> con
1
```

However, the reference can be retrieved by querying the model using the symbolic name:

```
julia> con = model[:con]
con : 2 x <= 1.0

julia> con
con : 2 x <= 1.0
```

Because the named variables and constraints are stored in the same namespace, creating a constraint with the same name as a variable or an existing constraint will result in an error. To overcome this limitation, it is possible to create anonymous constraints, just like it is possible to create Anonymous JuMP variables. This is done by dropping the second argument to `@constraint`:

```
julia> con = @constraint(model, 2x <= 1)
2 x <= 1.0
```

It is also possible use different comparison operators (e.g., >= and ==) to create the following types of constraints:

```
julia> @constraint(model, 2x >= 1)
2 x >= 1.0

julia> @constraint(model, 2x == 1)
2 x = 1.0

julia> @constraint(model, 1 <= 2x <= 3)
2 x ∈ [1.0, 3.0]
```

Note that JuMP normalizes the constraints by moving all of the terms containing variables to the left-hand side, and all of the constant terms to the right-hand side. Thus, we get:

```
julia> @constraint(model, 2x + 1 <= 4x + 4)
-2 x <= 3.0
```

13.2 The `@constraints` macro

Like `@variables`, there is a "plural" version of the `@constraint` macro:

```
julia> @constraints(model, begin
           2x <=  1
            x >= -1
       end)

julia> print(model)
Feasibility
Subject to
 x ≥ -1.0
 2 x ≤ 1.0
```

13.3 Duality

JuMP adopts the notion of conic duality from MOI. For linear programs, a feasible dual on a >= constraint is nonnegative and a feasible dual on a <= constraint is nonpositive. If the constraint is an equality constraint, it depends on which direction is binding.

> **Note**
>
> JuMP's definition of duality is independent of the objective sense. That is, the sign of feasible duals associated with a constraint depends on the direction of the constraint and not whether the problem is maximization or minimization. **This is a different convention from linear programming duality in some common textbooks.** If you have a linear program, and you want the textbook definition, you probably want to use `shadow_price` and `reduced_cost` instead.

The dual value associated with a constraint in the most recent solution can be accessed using the `dual` function. You can use the `has_duals` function to check whether the model has a dual solution available to query. For example:

```
julia> model = Model();

julia> @variable(model, x)
x

julia> @constraint(model, con, x <= 1)
con : x <= 1.0

julia> has_duals(model)
false
```

```
julia> @objective(model, Min, -2x)
-2 x

julia> optimize!(model)

julia> has_duals(model)
true

julia> dual(con)
```

```
-2.0

julia> @objective(model, Max, 2x)
2 x

julia> optimize!(model)

julia> dual(con)
-2.0
```

To help users who may be less familiar with conic duality, JuMP provides the `shadow_price` function which returns a value that can be interpreted as the improvement in the objective in response to an infinitesimal relaxation (on the scale of one unit) in the right-hand side of the constraint. `shadow_price` can be used only on linear constraints with a <=, >=, or == comparison operator.

In the example above, `dual(con)` returned `-2.0` regardless of the optimization sense. However, in the second case when the optimization sense is Max, `shadow_price` returns:

```
julia> shadow_price(con)
2.0
```

To query the dual variables associated with a variable bound, first obtain a constraint reference using one of `UpperBoundRef`, `LowerBoundRef`, or `FixRef`, and then call `dual` on the returned constraint reference. The `reduced_cost` function may simplify this process as it returns the shadow price of an active bound of a variable (or zero, if no active bound exists).

13.4 Constraint names

The name, i.e. the value of the `MOI.ConstraintName` attribute, of a constraint can be obtained by `name(::JuMP.ConstraintRef)` and set by `set_name(::JuMP.ConstraintRef, ::String)`.

```
julia> model = Model(); @variable(model, x);

julia> @constraint(model, con, x <= 1)
con : x <= 1.0

julia> name(con)
"con"

julia> set_name(con, "my_con_name")

julia> con
my_con_name : x <= 1.0
```

Specify a constraint name in the macro via `base_name`:

```
julia> model = Model(); @variable(model, x);

julia> con = @constraint(model, [i=1:2], x <= i, base_name = "my_con")
2-element Vector{ConstraintRef{Model,
↪  MathOptInterface.ConstraintIndex{MathOptInterface.ScalarAffineFunction{Float64},
↪  MathOptInterface.LessThan{Float64}}, ScalarShape}}:
 my_con[1] : x ≤ 1.0
 my_con[2] : x ≤ 2.0
```

Note that names apply to each element of the container, not to the container of constraints:

```
julia> name(con[1])
"my_con[1]"

julia> set_name(con[1], "c")

julia> con
2-element Vector{ConstraintRef{Model,
↪  MathOptInterface.ConstraintIndex{MathOptInterface.ScalarAffineFunction{Float64},
↪  MathOptInterface.LessThan{Float64}}, ScalarShape}}:
 c : x ≤ 1.0
 my_con[2] : x ≤ 2.0
```

Retrieve a constraint by name

Retrieve a constraint from a model using `constraint_by_name`:

```
julia> constraint_by_name(model, "c")
c : x ≤ 1.0
```

If the name is not present, `nothing` will be returned:

```
julia> constraint_by_name(model, "bad_name")
```

You can only look up invididual constraints using `constraint_by_name`. Something like this will not work:

```
julia> model = Model(); @variable(model, x);

julia> con = @constraint(model, [i=1:2], x <= i, base_name = "my_con")
2-element Vector{ConstraintRef{Model,
↪  MathOptInterface.ConstraintIndex{MathOptInterface.ScalarAffineFunction{Float64},
↪  MathOptInterface.LessThan{Float64}}, ScalarShape}}:
 my_con[1] : x ≤ 1.0
 my_con[2] : x ≤ 2.0

julia> constraint_by_name(model, "my_con")
```

To look up a collection of constraints, do not use `constraint_by_name`. Instead, register them using the `model[:key] = value` syntax:

```
julia> model = Model(); @variable(model, x);

julia> model[:con] = @constraint(model, [i=1:2], x <= i, base_name = "my_con")
2-element Vector{ConstraintRef{Model,
↪  MathOptInterface.ConstraintIndex{MathOptInterface.ScalarAffineFunction{Float64},
↪  MathOptInterface.LessThan{Float64}}, ScalarShape}}:
 my_con[1] : x ≤ 1.0
 my_con[2] : x ≤ 2.0

julia> model[:con]
```

```
2-element Vector{ConstraintRef{Model,
↪  MathOptInterface.ConstraintIndex{MathOptInterface.ScalarAffineFunction{Float64},
↪  MathOptInterface.LessThan{Float64}}, ScalarShape}}:
 my_con[1] : x ≤ 1.0
 my_con[2] : x ≤ 2.0
```

13.5 Start Values

Provide a starting value (also called warmstart) for a constraint's dual using `set_dual_start_value`.

The start value of a constraint's dual can be queried using `dual_start_value`. If no start value has been set, `dual_start_value` will return `nothing`.

```
julia> @variable(model, x)
x

julia> @constraint(model, con, x >= 10)
con : x ≥ 10.0

julia> dual_start_value(con)

julia> set_dual_start_value(con, 2)

julia> dual_start_value(con)
2.0
```

A vector constraint will require a vector warmstart:

```
julia> @variable(model, x[1:3])
3-element Vector{VariableRef}:
 x[1]
 x[2]
 x[3]

julia> @constraint(model, con, x in SecondOrderCone())
con : [x[1], x[2], x[3]] in MathOptInterface.SecondOrderCone(3)

julia> dual_start_value(con)

julia> set_dual_start_value(con, [1.0, 2.0, 3.0])

julia> dual_start_value(con)
3-element Vector{Float64}:
 1.0
 2.0
 3.0
```

To take the dual solution from the last solve and use it as the starting point for a new solve, use:

```
for (F, S) in list_of_constraint_types(model)
    for con in all_constraints(model, F, S)
        set_dual_start_value(con, dual(con))
    end
end
```

Note

Some constraints might not have well defined duals, hence one might need to filter `(F, S)` pairs.

13.6 Constraint containers

So far, we've added constraints one-by-one. However, just like Variable containers, JuMP provides a mechanism for building groups of constraints compactly. References to these groups of constraints are returned in containers. Three types of constraint containers are supported: `Array`s, `DenseAxisArray`s, and `SparseAxisArray`s. We explain each of these in the following.

Tip

You can read more about containers in the Containers section.

Arrays

One way of adding a group of constraints compactly is the following:

```
julia> @constraint(model, con[i = 1:3], i * x <= i + 1)
3-element Vector{ConstraintRef{Model,
↪  MathOptInterface.ConstraintIndex{MathOptInterface.ScalarAffineFunction{Float64},
↪  MathOptInterface.LessThan{Float64}}, ScalarShape}}:
 con[1] : x ≤ 2.0
 con[2] : 2 x ≤ 3.0
 con[3] : 3 x ≤ 4.0
```

JuMP returns references to the three constraints in an `Array` that is bound to the Julia variable `con`. This array can be accessed and sliced as you would with any Julia array:

```
julia> con[1]
con[1] : x <= 2.0

julia> con[2:3]
2-element Vector{ConstraintRef{Model,
↪  MathOptInterface.ConstraintIndex{MathOptInterface.ScalarAffineFunction{Float64},
↪  MathOptInterface.LessThan{Float64}}, ScalarShape}}:
 con[2] : 2 x ≤ 3.0
 con[3] : 3 x ≤ 4.0
```

Anonymous containers can also be constructed by dropping the name (e.g. `con`) before the square brackets:

```
julia> @constraint(model, [i = 1:2], i * x <= i + 1)
2-element Vector{ConstraintRef{Model,
↪  MathOptInterface.ConstraintIndex{MathOptInterface.ScalarAffineFunction{Float64},
↪  MathOptInterface.LessThan{Float64}}, ScalarShape}}:
 x ≤ 2.0
 2 x ≤ 3.0
```

Just like `@variable`, JuMP will form an `Array` of constraints when it can determine at parse time that the indices are one-based integer ranges. Therefore `con[1:b]` will create an `Array`, but `con[a:b]` will not. A special case is `con[Base.OneTo(n)]` which will produce an `Array`. If JuMP cannot determine that the indices are one-based integer ranges (e.g., in the case of `con[a:b]`), JuMP will create a `DenseAxisArray` instead.

DenseAxisArrays

The syntax for constructing a DenseAxisArray of constraints is very similar to the syntax for constructing a DenseAxisArray of variables.

```
julia> @constraint(model, con[i = 1:2, j = 2:3], i * x <= j + 1)
2-dimensional DenseAxisArray{ConstraintRef{Model,
↪  MathOptInterface.ConstraintIndex{MathOptInterface.ScalarAffineFunction{Float64},
↪  MathOptInterface.LessThan{Float64}}, ScalarShape},2,...} with index sets:
    Dimension 1, Base.OneTo(2)
    Dimension 2, 2:3
And data, a 2×2 Matrix{ConstraintRef{Model,
↪  MathOptInterface.ConstraintIndex{MathOptInterface.ScalarAffineFunction{Float64},
↪  MathOptInterface.LessThan{Float64}}, ScalarShape}}:
 con[1,2] : x ≤ 3.0    con[1,3] : x ≤ 4.0
 con[2,2] : 2 x ≤ 3.0  con[2,3] : 2 x ≤ 4.0
```

SparseAxisArrays

The syntax for constructing a SparseAxisArray of constraints is very similar to the syntax for constructing a SparseAxisArray of variables.

```
julia> @constraint(model, con[i = 1:2, j = 1:2; i != j], i * x <= j + 1)
JuMP.Containers.SparseAxisArray{ConstraintRef{Model,
↪  MathOptInterface.ConstraintIndex{MathOptInterface.ScalarAffineFunction{Float64},
↪  MathOptInterface.LessThan{Float64}}, ScalarShape}, 2, Tuple{Int64, Int64}} with 2 entries:
  [1, 2]  =  con[1,2] : x ≤ 3.0
  [2, 1]  =  con[2,1] : 2 x ≤ 2.0
```

Forcing the container type

When creating a container of constraints, JuMP will attempt to choose the tightest container type that can store the constraints. However, because this happens at parse time, it does not always make the best choice. Just like in @variable, we can force the type of container using the container keyword. For syntax and the reason behind this, take a look at the variable docs.

13.7 Vectorized constraints

We can also add constraints to JuMP using vectorized linear algebra. For example:

```
julia> @variable(model, x[i=1:2])
2-element Vector{VariableRef}:
 x[1]
 x[2]

julia> A = [1 2; 3 4]
2×2 Matrix{Int64}:
 1  2
 3  4

julia> b = [5, 6]
2-element Vector{Int64}:
 5
 6
```

```
julia> @constraint(model, con, A * x .== b)
2-element Vector{ConstraintRef{Model,
↪  MathOptInterface.ConstraintIndex{MathOptInterface.ScalarAffineFunction{Float64},
↪  MathOptInterface.EqualTo{Float64}}, ScalarShape}}:
 con : x[1] + 2 x[2] = 5.0
 con : 3 x[1] + 4 x[2] = 6.0
```

Note

Make sure to use Julia's dot syntax in front of the comparison operators (e.g. `.==`, `.>=`, and `.<=`). If you use a comparison without the dot, an error will be thrown.

Instead of adding an array of `ScalarAffineFunction-in-EqualTo` constraints, we can instead construct a `VectorAffineFunction-in-Nonnegatives` constraint as follows:

```
julia> @constraint(model, A * x - b in MOI.Nonnegatives(2))
[x[1] + 2 x[2] - 5, 3 x[1] + 4 x[2] - 6] in MathOptInterface.Nonnegatives(2)
```

In addition to the `Nonnegatives` set, MOI defines a number of other vector-valued sets such as `Nonpositives`. See the Vector cones section for more information.

Note

For the first time we have used an explicit function-in-set description of the constraint. Read more about this in Standard form problem.

13.8 Constraints on a single variable

In Variables, we saw how to modify the variable bounds, as well as add binary and integer restrictions to the domain of each variable. This can also be achieved using the `@constraint` macro. For example, `MOI.ZeroOne()` restricts the domain to $\{0, 1\}$:

```
julia> @constraint(model, x in MOI.ZeroOne())
x binary
```

and `MOI.Integer()` restricts to the domain to the integers $\mathbb{Z}$:

```
julia> @constraint(model, x in MOI.Integer())
x integer
```

JuMP also supports modeling semi-continuous variables, whose domain is $\{0\}[l, u]$, using the `MOI.Semicontinuous` set:

```
julia> @constraint(model, x in MOI.Semicontinuous(1.5, 3.5))
x in MathOptInterface.Semicontinuous{Float64}(1.5, 3.5)
```

as well as semi-integer variables, whose domain is $\{0\}\{l, l+1, \ldots, u\}$, using the `MOI.Semiinteger` set:

```
julia> @constraint(model, x in MOI.Semiinteger(1.0, 3.0))
x in MathOptInterface.Semiinteger{Float64}(1.0, 3.0)
```

13.9 Quadratic constraints

In addition to affine functions, JuMP also supports constraints with quadratic terms. (For more general nonlinear functions, see Nonlinear Modeling.) For example:

```
julia> @variable(model, x[i=1:2])
2-element Vector{VariableRef}:
 x[1]
 x[2]

julia> @variable(model, t >= 0)
t

julia> @constraint(model, x[1]^2 + x[2]^2 <= t^2)
x[1]² + x[2]² - t² <= 0.0
```

Note that this quadratic constraint (including the lower bound on `t`) is equivalent to a second order cone constraint where `||x[1]^2 + x[2]^2||\_2 ≤ t` and `t ≥ 0`. Instead of writing out the quadratic expansion, we can pass JuMP the constraint in function-in-set form. To do so, we need to define the function and the set.

The function is a vector of variables:

```
julia> [t, x[1], x[2]]
3-element Vector{VariableRef}:
 t
 x[1]
 x[2]
```

Note that the variable `t` comes first, followed by the `x` arguments. The set is an instance of `SecondOrderCone`: `SecondOrderCone()`. Thus, we can add the second order cone constraint as follows:

```
julia> @constraint(model, [t, x[1], x[2]] in SecondOrderCone())
[t, x[1], x[2]] in MathOptInterface.SecondOrderCone(3)
```

JuMP also supports the `RotatedSecondOrderCone` which requires the addition of a perspective variable `u`. The rotated second order cone constraints the variables `t`, `u`, and `x` such that: `||x[1]^2 + x[2]^2||\_2 ≤ t × u` and `t, u ≥ 0`. It can be added as follows:

```
julia> @variable(model, u)
u

julia> @constraint(model, [t, u, x[1], x[2]] in RotatedSecondOrderCone())
[t, u, x[1], x[2]] in MathOptInterface.RotatedSecondOrderCone(4)
```

13.10 Constraints on a collection of variables

In addition to constraining the domain of a single variable, JuMP supports placing constraints of a subset of the variables. We already saw an example of this in the Quadratic constraints section when we constrained a vector of variables to belong to the second order cone.

In a special ordered set of type I (often denoted SOS-I), at most one variable can take a non-zero value. We can construct SOS-I constraints using the `MOI.SOS1` set:

```
julia> @variable(model, x[1:3])
3-element Vector{VariableRef}:
 x[1]
 x[2]
 x[3]

julia> @constraint(model, x in MOI.SOS1([1.0, 2.0, 3.0]))
[x[1], x[2], x[3]] in MathOptInterface.SOS1{Float64}([1.0, 2.0, 3.0])
```

Note that we have to pass `MOI.SOS1` a weight vector. This vector implies an ordering on the variables. If the decision variables are related and have a physical ordering (e.g., they correspond to the size of a factory to be built, and the SOS-I constraint enforces that only one factory can be built), then the weight vector, although not used directly in the constraint, can help the solver make a better decision in the solution process.

This ordering is more important in a special ordered set of type II (SOS-II), in which at most two values can be non-zero, and if there are two non-zeros, they must be consecutive according to the ordering. For example, in the following constraint, the possible non-zero pairs are (`x[1]` and `x[3]`) and (`x[2]` and `x[3]`):

```
julia> @constraint(model, x in MOI.SOS2([3.0, 1.0, 2.0]))
[x[1], x[2], x[3]] in MathOptInterface.SOS2{Float64}([3.0, 1.0, 2.0])
```

13.11 Indicator constraints

JuMP provides a special syntax for creating indicator constraints, that is, enforce a constraint to hold depending on the value of a binary variable. In order to constrain the constraint `x + y <= 1` to hold when a binary variable `a` is one, use the following syntax:

```
julia> @variable(model, x)
x

julia> @variable(model, y)
y

julia> @variable(model, a, Bin)
a

julia> @constraint(model, a => {x + y <= 1})
a => {x + y ≤ 1.0}
```

If instead the constraint should hold when `a` is zero, simply add a `!` or ¬ before the binary variable.

```
julia> @constraint(model, !a => {x + y <= 1})
!a => {x + y ≤ 1.0}
```

13.12 Semidefinite constraints

Use PSDCone to constrain a matrix to be symmetric positive semidefinite (PSD). For example,

```
julia> @variable(model, X[1:2, 1:2])
2×2 Matrix{VariableRef}:
 X[1,1]  X[1,2]
 X[2,1]  X[2,2]
```

```
julia> @constraint(model, X >= 0, PSDCone())
[X[1,1]  X[1,2];
 X[2,1]  X[2,2]] ∈ PSDCone()
```

The inequality `X >= Y` between two square matrices `X` and `Y` is understood as constraining `X - Y` to be symmetric positive semidefinite.

```
julia> Y = [1 2; 2 1]
2×2 Matrix{Int64}:
 1  2
 2  1

julia> @constraint(model, X >= Y, PSDCone())
[X[1,1] - 1  X[1,2] - 2;
 X[2,1] - 2  X[2,2] - 1] ∈ PSDCone()
```

Tip

`@constraint(model, X >= Y, Set())` is short-hand for `@constraint(model, X - Y in Set())`. Therefore, the following calls are equivalent:

- `@constraint(model, X >= Y, PSDCone())`
- `@constraint(model, Y <= X, PSDCone())`
- `@constraint(model, X - Y in PSDCone())`

This also works for any vector-valued cone, so if `x` and `y` are vectors of length 2, you can write `@constraint(model, x >= y, MOI.Nonnegatives(2))` instead of `@constraint(model, x - y in MOI.Nonnegatives(2))`.

Warning

Non-zero constants are not supported in this syntax:

```
julia> @constraint(model, X >= 1, PSDCone())
ERROR: Operation `sub_mul` between `Matrix{VariableRef}` and `Int64` is not allowed. You
↪  should use broadcast.
Stacktrace:
[...]
```

Use instead:

```
julia> @constraint(model, X .- 1 >= 0, PSDCone())
[X[1,1] - 1  X[1,2] - 1;
 X[2,1] - 1  X[2,2] - 1] ∈ PSDCone()
```

Symmetry

Solvers supporting PSD constraints usually expect to be given a matrix that is symbolically symmetric, that is, for which the expression in corresponding off-diagonal entries are the same. In our example, the expressions of entries `(1, 2)` and `(2, 1)` are respectively `X[1,2] - 2` and `X[2,1] - 2` which are different.

To bridge the gap between the constraint modeled and what the solver expects, solvers may add an equality constraint `X[1,2] - 2 == X[2,1] - 2` to force symmetry. Use `LinearAlgebra.Symmetric` to explicitly tell the solver that the matrix is symmetric:

```
julia> import LinearAlgebra

julia> Z = [X[1, 1] X[1, 2]; X[1, 2] X[2, 2]]
2×2 Matrix{VariableRef}:
 X[1,1]  X[1,2]
 X[1,2]  X[2,2]

julia> @constraint(model, LinearAlgebra.Symmetric(Z) >= 0, PSDCone())
[X[1,1]  X[1,2];
 X[1,2]  X[2,2]] ∈ PSDCone()
```

Note that the lower triangular entries are silently ignored even if they are different so use it with caution:

```
julia> @constraint(model, LinearAlgebra.Symmetric(X) >= 0, PSDCone())
[X[1,1]  X[1,2];
 X[1,2]  X[2,2]] ∈ PSDCone()
```

(Note the `(2, 1)` element of the constraint is `X[1,2]`, not `X[2,1]`.)

13.13 Modify a constraint

Modifying a constant term (Option 1)

Use `set_normalized_rhs` to modify the right-hand side (constant) term of a constraint. Use `normalized_rhs` to query the right-hand side term.

```
julia> @constraint(model, con, 2x <= 1)
con : 2 x <= 1.0

julia> set_normalized_rhs(con, 3)

julia> con
con : 2 x <= 3.0

julia> normalized_rhs(con)
3.0
```

> **Note**
>
> JuMP normalizes constraints into a standard form by moving all constant terms onto the right-hand side of the constraint.
>
> ```
> @constraint(model, 2x - 1 <= 2)
> ```
>
> will be normalized to
>
> ```
> @constraint(model, 2x <= 3)
> ```
>
> `set_normalized_rhs` sets the right-hand side term of the normalized constraint.

Modifying a constant term (Option 2)

If constraints are complicated, e.g., they are composed of a number of components, each of which has a constant term, then it may be difficult to calculate what the right-hand side term should be in the standard form.

For this situation, JuMP includes the ability to fix variables to a value using the `fix` function. Fixing a variable sets its lower and upper bound to the same value. Thus, changes in a constant term can be simulated by adding a dummy variable and fixing it to different values. Here is an example:

```
julia> @variable(model, const_term)
const_term

julia> @constraint(model, con, 2x <= const_term + 1)
con : 2 x - const_term <= 1.0

julia> fix(const_term, 1.0)
```

The constraint `con` is now equivalent to `2x <= 2`.

> **Note**
>
> Even though `const_term` is fixed, it is still a decision variable. Thus, `const_term * x` is bilinear. Fixed variables are not replaced with constants when communicating the problem to a solver.

Another option is to use `add_to_function_constant`. The constant given is added to the function of a `func-in-set` constraint. In the following example, adding 2 to the function has the effect of removing 2 to the right-hand side:

```
julia> @constraint(model, con, 2x <= 1)
con : 2 x <= 1.0

julia> add_to_function_constant(con, 2)

julia> con
con : 2 x <= -1.0

julia> normalized_rhs(con)
-1.0
```

In the case of interval constraints, the constant is removed in each bounds.

```
julia> @constraint(model, con, 0 <= 2x + 1 <= 2)
con : 2 x ∈ [-1.0, 1.0]

julia> add_to_function_constant(con, 3)

julia> con
con : 2 x ∈ [-4.0, -2.0]
```

Modifying a variable coefficient

To modify the coefficients for a linear term in a constraint (but notably not yet the coefficients on a quadratic term), use `set_normalized_coefficient`. To query the current coefficient, use `normalized_coefficient`.

```
julia> @constraint(model, con, 2x[1] + x[2] <= 1)
con : 2 x[1] + x[2] ≤ 1.0

julia> set_normalized_coefficient(con, x[2], 0)

julia> con
con : 2 x[1] ≤ 1.0

julia> normalized_coefficient(con, x[2])
0.0
```

> **Note**
>
> JuMP normalizes constraints into a standard form by moving all terms involving variables onto the left-hand side of the constraint.
>
> ```
> @constraint(model, 2x <= 1 - x)
> ```
>
> will be normalized to
>
> ```
> @constraint(model, 3x <= 1)
> ```
>
> `set_normalized_coefficient` sets the coefficient of the normalized constraint.

13.14 Delete a constraint

Use `delete` to delete a constraint from a model. Use `is_valid` to check if a constraint belongs to a model and has not been deleted.

```
julia> @constraint(model, con, 2x <= 1)
con : 2 x <= 1.0

julia> is_valid(model, con)
true

julia> delete(model, con)

julia> is_valid(model, con)
false
```

Deleting a constraint does not unregister the symbolic reference from the model. Therefore, creating a new constraint of the same name will throw an error:

```
julia> @constraint(model, con, 2x <= 1)
ERROR: An object of name con is already attached to this model. If this
    is intended, consider using the anonymous construction syntax, e.g.,
    `x = @variable(model, [1:N], ...)` where the name of the object does
    not appear inside the macro.
```

```
    Alternatively, use `unregister(model, :con)` to first unregister
    the existing name from the model. Note that this will not delete the
    object; it will just remove the reference at `model[:con]`.
[...]
```

After calling `delete`, call `unregister` to remove the symbolic reference:

```
julia> unregister(model, :con)

julia> @constraint(model, con, 2x <= 1)
con : 2 x <= 1.0
```

Info

`delete` does not automatically `unregister` because we do not distinguish between names that are automatically registered by JuMP macros, and names that are manually registered by the user by setting values in `object_dictionary`. In addition, deleting a constraint and then adding a new constraint of the same name is an easy way to introduce bugs into your code.

13.15 Accessing constraints from a model

You can query the types of constraints currently present in the model by calling `list_of_constraint_types`. Then, given a function and set type, use `num_constraints` to access the number of constraints of this type and `all_constraints` to access a list of their references. Then use `constraint_object` to get an instance of an `AbstractConstraint` object, either `ScalarConstraint` or `VectorConstraint`, that stores the constraint data.

```
julia> model = Model();

julia> @variable(model, x[i=1:2] >= i, Int);

julia> @constraint(model, x[1] + x[2] <= 1);

julia> list_of_constraint_types(model)
3-element Vector{Tuple{Type, Type}}:
 (AffExpr, MathOptInterface.LessThan{Float64})
 (VariableRef, MathOptInterface.GreaterThan{Float64})
 (VariableRef, MathOptInterface.Integer)

julia> num_constraints(model, VariableRef, MOI.Integer)
2

julia> all_constraints(model, VariableRef, MOI.Integer)
2-element Vector{ConstraintRef{Model,
↪  MathOptInterface.ConstraintIndex{MathOptInterface.VariableIndex, MathOptInterface.Integer},
↪  ScalarShape}}:
 x[1] integer
 x[2] integer

julia> num_constraints(model, VariableRef, MOI.GreaterThan{Float64})
2

julia> all_constraints(model, VariableRef, MOI.GreaterThan{Float64})
```

```
2-element Vector{ConstraintRef{Model,
↪  MathOptInterface.ConstraintIndex{MathOptInterface.VariableIndex,
↪  MathOptInterface.GreaterThan{Float64}}, ScalarShape}}:
 x[1] ≥ 1.0
 x[2] ≥ 2.0

julia> num_constraints(model, GenericAffExpr{Float64,VariableRef}, MOI.LessThan{Float64})
1

julia> less_than_constraints = all_constraints(model, GenericAffExpr{Float64,VariableRef},
↪  MOI.LessThan{Float64})
1-element Vector{ConstraintRef{Model,
↪  MathOptInterface.ConstraintIndex{MathOptInterface.ScalarAffineFunction{Float64},
↪  MathOptInterface.LessThan{Float64}}, ScalarShape}}:
 x[1] + x[2] ≤ 1.0

julia> con = constraint_object(less_than_constraints[1])
ScalarConstraint{AffExpr, MathOptInterface.LessThan{Float64}}(x[1] + x[2],
↪  MathOptInterface.LessThan{Float64}(1.0))

julia> con.func
x[1] + x[2]

julia> con.set
MathOptInterface.LessThan{Float64}(1.0)
```

13.16 Complementarity constraints

A mixed complementarity constraint `F(x) ⊥ x` consists of finding `x` in the interval `[lb, ub]`, such that the following holds:

- `F(x) == 0` if `lb < x < ub`
- `F(x) >= 0` if `lb == x`
- `F(x) <= 0` if `x == ub`

JuMP supports mixed complementarity constraints via `complements(F(x), x)` or `F(x) ⊥ x` in the `@constraint` macro. The interval set `[lb, ub]` is obtained from the variable bounds on `x`.

For example, to define the problem `2x - 1 ⊥ x` with `x ∈ [0, ∞)`, do:

```
julia> @variable(model, x >= 0)
x

julia> @constraint(model, 2x - 1 ⊥ x)
[2 x - 1, x] ∈ MathOptInterface.Complements(2)
```

This problem has a unique solution at `x = 0.5`.

The perp operator `⊥` can be entered in most editors (and the Julia REPL) by typing `\perp<tab>`.

An alternative approach that does not require the `⊥` symbol uses the `complements` function as follows:

```
julia> @constraint(model, complements(2x - 1, x))
[2 x - 1, x] ∈ MathOptInterface.Complements(2)
```

In both cases, the mapping `F(x)` is supplied as the first argument, and the matching variable `x` is supplied as the second.

Vector-valued complementarity constraints are also supported:

```
julia> @variable(model, -2 <= y[1:2] <= 2)
2-element Vector{VariableRef}:
 y[1]
 y[2]

julia> M = [1 2; 3 4]
2×2 Matrix{Int64}:
 1  2
 3  4

julia> q = [5, 6]
2-element Vector{Int64}:
 5
 6

julia> @constraint(model, M * y + q ⊥ y)
[y[1] + 2 y[2] + 5, 3 y[1] + 4 y[2] + 6, y[1], y[2]] ∈ MathOptInterface.Complements(4)
```

13.17 Special Ordered Sets (SOS1 and SOS2)

Type 1

In a Special Ordered Set of Type 1 (often denoted SOS-I or SOS1), at most one element can take a non-zero value.

Construct SOS-I constraints using the SOS1 set:

```
julia> @variable(model, x[1:3])
3-element Vector{VariableRef}:
 x[1]
 x[2]
 x[3]

julia> @constraint(model, x in SOS1())
[x[1], x[2], x[3]] in MathOptInterface.SOS1{Float64}([1.0, 2.0, 3.0])
```

Although not required for feasibility, solvers can benefit from an ordering of the variables (e.g., the variables represent different factories to build, at most one factory can be built, and the factories can be ordered according to cost). To induce an ordering, weights can be provided; as such, they should be unique values. The kth element in the ordered set corresponds to the kth weight in weights when the weights are sorted.

For example, in the constraint:

```
julia> @constraint(model, x in SOS1([3.1, 1.2, 2.3]))
[x[1], x[2], x[3]] in MathOptInterface.SOS1{Float64}([3.1, 1.2, 2.3])
```

the variables `x` have precedence `x[2]`, `x[3]`, `x[1]`.

Type 2

In a Special Ordered Set of Type 2 (SOS-II), at most two elements can be non-zero, and if there are two non-zeros, they must be consecutive according to the ordering induced by a weight vector.

Construct SOS-II constraints using the SOS2 set.

```
julia> @constraint(model, x in SOS2([3.0, 1.0, 2.0]))
[x[1], x[2], x[3]] in MathOptInterface.SOS2{Float64}([3.0, 1.0, 2.0])
```

In the following constraint, the possible non-zero pairs are (`x[1]` and `x[3]`) and (`x[2]` and `x[3]`):

If the weight vector is omitted, JuMP induces an ordering from `1:length(x)`:

```
julia> @constraint(model, x in SOS2())
[x[1], x[2], x[3]] in MathOptInterface.SOS2{Float64}([1.0, 2.0, 3.0])
```

Chapter 14

Containers

JuMP provides specialized containers similar to `AxisArrays` that enable multi-dimensional arrays with non-integer indices.

These containers are created automatically by JuMP's macros. Each macro has the same basic syntax:

```
@macroname(model, name[key1=index1, index2; optional_condition], other stuff)
```

The containers are generated by the `name[key1=index1, index2; optional_condition]` syntax. Everything else is specific to the particular macro.

Containers can be named, e.g., `name[key=index]`, or unnamed, e.g., `[key=index]`. We call unnamed containers anonymous.

We call the bits inside the square brackets and before the `;` the index sets. The index sets can be named, e.g., `[i = 1:4]`, or they can be unnamed, e.g., `[1:4]`.

We call the bit inside the square brackets and after the `;` the condition. Conditions are optional.

In addition to the standard JuMP macros like `@variable` and `@constraint`, which construct containers of variables and constraints respectively, you can use `Containers.@container` to construct containers with arbitrary elements.

We will use this macro to explain the three types of containers that are natively supported by JuMP: `Array`, `Containers.DenseAxisArray`, and `Containers.SparseAxisArray`.

14.1 Array

An `Array` is created when the index sets are rectangular and the index sets are of the form `1:n`.

```
julia> Containers.@container(x[i = 1:2, j = 1:3], (i, j))
2×3 Matrix{Tuple{Int64, Int64}}:
 (1, 1)  (1, 2)  (1, 3)
 (2, 1)  (2, 2)  (2, 3)
```

The result is just a normal Julia array, so you can do all the usual things.

Slicing

Arrays can be sliced

```
julia> x[:, 1]
2-element Vector{Tuple{Int64, Int64}}:
 (1, 1)
 (2, 1)

julia> x[2, :]
3-element Vector{Tuple{Int64, Int64}}:
 (2, 1)
 (2, 2)
 (2, 3)
```

Looping

Use eachindex to loop over the elements:

```
julia> for key in eachindex(x)
           println(x[key])
       end
(1, 1)
(2, 1)
(1, 2)
(2, 2)
(1, 3)
(2, 3)
```

Get the index sets

Use axes to obtain the index sets:

```
julia> axes(x)
(Base.OneTo(2), Base.OneTo(3))
```

Broadcasting

Broadcasting over an Array returns an Array

```
julia> swap(x::Tuple) = (last(x), first(x))
swap (generic function with 1 method)

julia> swap.(x)
2×3 Matrix{Tuple{Int64, Int64}}:
 (1, 1)  (2, 1)  (3, 1)
 (1, 2)  (2, 2)  (3, 2)
```

14.2 DenseAxisArray

A `Containers.DenseAxisArray` is created when the index sets are rectangular, but not of the form `1:n`. The index sets can be of any type.

```
julia> x = Containers.@container([i = 1:2, j = [:A, :B]], (i, j))
2-dimensional DenseAxisArray{Tuple{Int64, Symbol},2,...} with index sets:
    Dimension 1, Base.OneTo(2)
    Dimension 2, [:A, :B]
And data, a 2×2 Matrix{Tuple{Int64, Symbol}}:
 (1, :A)  (1, :B)
 (2, :A)  (2, :B)
```

Slicing

DenseAxisArrays can be sliced

```
julia> x[:, :A]
1-dimensional DenseAxisArray{Tuple{Int64, Symbol},1,...} with index sets:
    Dimension 1, Base.OneTo(2)
And data, a 2-element Vector{Tuple{Int64, Symbol}}:
 (1, :A)
 (2, :A)

julia> x[1, :]
1-dimensional DenseAxisArray{Tuple{Int64, Symbol},1,...} with index sets:
    Dimension 1, [:A, :B]
And data, a 2-element Vector{Tuple{Int64, Symbol}}:
 (1, :A)
 (1, :B)
```

Looping

Use eachindex to loop over the elements:

```
julia> for key in eachindex(x)
           println(x[key])
       end
(1, :A)
(2, :A)
(1, :B)
(2, :B)
```

Get the index sets

Use axes to obtain the index sets:

```
julia> axes(x)
(Base.OneTo(2), [:A, :B])
```

Broadcasting

Broadcasting over a DenseAxisArray returns a DenseAxisArray

```
julia> swap(x::Tuple) = (last(x), first(x))
swap (generic function with 1 method)

julia> swap.(x)
2-dimensional DenseAxisArray{Tuple{Symbol, Int64},2,...} with index sets:
```

```
    Dimension 1, Base.OneTo(2)
    Dimension 2, [:A, :B]
And data, a 2×2 Matrix{Tuple{Symbol, Int64}}:
 (:A, 1)  (:B, 1)
 (:A, 2)  (:B, 2)
```

Access internal data

Use `Array(x)` to copy the internal data array into a new `Array`:

```
julia> Array(x)
2×2 Matrix{Tuple{Int64, Symbol}}:
 (1, :A)  (1, :B)
 (2, :A)  (2, :B)
```

To access the internal data without a copy, use `x.data`.

```
julia> x.data
2×2 Matrix{Tuple{Int64, Symbol}}:
 (1, :A)  (1, :B)
 (2, :A)  (2, :B)
```

14.3 SparseAxisArray

A `Containers.SparseAxisArray` is created when the index sets are non-rectangular. This occurs in two circumstances:

An index depends on a prior index:

```
julia> Containers.@container([i = 1:2, j = i:2], (i, j))
JuMP.Containers.SparseAxisArray{Tuple{Int64, Int64}, 2, Tuple{Int64, Int64}} with 3 entries:
  [1, 1]  =  (1, 1)
  [1, 2]  =  (1, 2)
  [2, 2]  =  (2, 2)
```

The `[indices; condition]` syntax is used:

```
julia> x = Containers.@container([i = 1:3, j = [:A, :B]; i > 1 && j == :B], (i, j))
JuMP.Containers.SparseAxisArray{Tuple{Int64, Symbol}, 2, Tuple{Int64, Symbol}} with 2 entries:
  [2, B]  =  (2, :B)
  [3, B]  =  (3, :B)
```

Here we have the index sets `i = 1:3, j = [:A, :B]`, followed by `;`, and then a condition, which evaluates to `true` or `false`: `i > 1 && j == :B`.

Slicing

Slicing is not supported.

```
julia> x[:, :B]
ERROR: ArgumentError: Indexing with `:` is not supported by Containers.SparseAxisArray
[...]
```

Looping

Use `eachindex` to loop over the elements:

```
julia> for key in eachindex(x)
           println(x[key])
       end
(2, :B)
(3, :B)
```

Broadcasting

Broadcasting over a SparseAxisArray returns a SparseAxisArray

```
julia> swap(x::Tuple) = (last(x), first(x))
swap (generic function with 1 method)

julia> swap.(x)
JuMP.Containers.SparseAxisArray{Tuple{Symbol, Int64}, 2, Tuple{Int64, Symbol}} with 2 entries:
  [2, B]  =  (:B, 2)
  [3, B]  =  (:B, 3)
```

14.4 Forcing the container type

Pass `container = T` to use `T` as the container. For example:

```
julia> Containers.@container([i = 1:2, j = 1:2], i + j, container = Array)
2×2 Matrix{Int64}:
 2  3
 3  4

julia> Containers.@container([i = 1:2, j = 1:2], i + j, container = Dict)
Dict{Tuple{Int64, Int64}, Int64} with 4 entries:
  (1, 2) => 3
  (1, 1) => 2
  (2, 2) => 4
  (2, 1) => 3
```

You can also pass `DenseAxisArray` or `SparseAxisArray`.

14.5 How different container types are chosen

If the compiler can prove at compile time that the index sets are rectangular, and indexed by a compact set of integers that start at 1, `Containers.@container` will return an array. This is the case if your index sets are visible to the macro as `1:n`:

```
julia> Containers.@container([i=1:3, j=1:5], i + j)
3×5 Matrix{Int64}:
 2  3  4  5  6
 3  4  5  6  7
 4  5  6  7  8
```

or an instance of `Base.OneTo`:

```
julia> set = Base.OneTo(3)
Base.OneTo(3)

julia> Containers.@container([i=set, j=1:5], i + j)
3×5 Matrix{Int64}:
 2  3  4  5  6
 3  4  5  6  7
 4  5  6  7  8
```

If the compiler can prove that the index set is rectangular, but not necessarily of the form `1:n` at compile time, then a `Containers.DenseAxisArray` will be constructed instead:

```
julia> set = 1:3
1:3

julia> Containers.@container([i=set, j=1:5], i + j)
2-dimensional DenseAxisArray{Int64,2,...} with index sets:
    Dimension 1, 1:3
    Dimension 2, Base.OneTo(5)
And data, a 3×5 Matrix{Int64}:
 2  3  4  5  6
 3  4  5  6  7
 4  5  6  7  8
```

> **Info**
>
> What happened here? Although we know that `set` contains `1:3`, at compile time the `typeof(set)` is a `UnitRange{Int}`. Therefore, Julia can't prove that the range starts at 1 (it only finds this out at runtime), and it defaults to a `DenseAxisArray`. The case where we explicitly wrote `i = 1:3` worked because the macro can "see" the 1 at compile time.

However, if you know that the indices really do form an `Array`, you can force the container type with `container = Array`:

```
julia> set = 1:3
1:3

julia> Containers.@container([i=set, j=1:5], i + j, container = Array)
3×5 Matrix{Int64}:
 2  3  4  5  6
 3  4  5  6  7
 4  5  6  7  8
```

Here's another example with something similar:

```
julia> a = 1
1

julia> Containers.@container([i=a:3, j=1:5], i + j)
2-dimensional DenseAxisArray{Int64,2,...} with index sets:
    Dimension 1, 1:3
    Dimension 2, Base.OneTo(5)
And data, a 3×5 Matrix{Int64}:
```

```
 2  3  4  5  6
 3  4  5  6  7
 4  5  6  7  8

julia> Containers.@container([i=1:a, j=1:5], i + j)
1×5 Matrix{Int64}:
 2  3  4  5  6
```

Finally, if the compiler cannot prove that the index set is rectangular, a `Containers.SparseAxisArray` will be created.

This occurs when some indices depend on a previous one:

```
julia> Containers.@container([i=1:3, j=1:i], i + j)
JuMP.Containers.SparseAxisArray{Int64, 2, Tuple{Int64, Int64}} with 6 entries:
  [1, 1]  =  2
  [2, 1]  =  3
  [2, 2]  =  4
  [3, 1]  =  4
  [3, 2]  =  5
  [3, 3]  =  6
```

or if there is a condition on the index sets:

```
julia> Containers.@container([i = 1:5; isodd(i)], i^2)
JuMP.Containers.SparseAxisArray{Int64, 1, Tuple{Int64}} with 3 entries:
  [1]  =  1
  [3]  =  9
  [5]  =  25
```

The condition can depend on multiple indices; it just needs to be a function that returns `true` or `false`:

```
julia> condition(i, j) = isodd(i) && iseven(j)
condition (generic function with 1 method)

julia> Containers.@container([i = 1:2, j = 1:4; condition(i, j)], i + j)
JuMP.Containers.SparseAxisArray{Int64, 2, Tuple{Int64, Int64}} with 2 entries:
  [1, 2]  =  3
  [1, 4]  =  5
```

Chapter 15

Solutions

This section of the manual describes how to access a solved solution to a problem. It uses the following model as an example:

```
model = Model(GLPK.Optimizer)
@variable(model, x >= 0)
@variable(model, y[[:a, :b]] <= 1)
@objective(model, Max, -12x - 20y[:a])
@expression(model, my_expr, 6x + 8y[:a])
@constraint(model, my_expr >= 100)
@constraint(model, c1, 7x + 12y[:a] >= 120)
optimize!(model)
print(model)

# output

Max -12 x - 20 y[a]
Subject to
 6 x + 8 y[a] ≥ 100.0
 c1 : 7 x + 12 y[a] ≥ 120.0
 x ≥ 0.0
 y[a] ≤ 1.0
 y[b] ≤ 1.0
```

15.1 Solutions summary

`solution_summary` can be used for checking the summary of the optimization solutions.

```
julia> solution_summary(model)
* Solver : GLPK

* Status
  Termination status : OPTIMAL
  Primal status      : FEASIBLE_POINT
  Dual status        : FEASIBLE_POINT
  Message from the solver:
  "Solution is optimal"

* Candidate solution
  Objective value      : -205.14285714285714
```

```
  Objective bound      : Inf
  Dual objective value : -205.1428571428571

* Work counters
  Solve time (sec)   : 0.00008

julia> solution_summary(model, verbose=true)
* Solver : GLPK

* Status
  Termination status : OPTIMAL
  Primal status      : FEASIBLE_POINT
  Dual status        : FEASIBLE_POINT
  Result count       : 1
  Has duals          : true
  Message from the solver:
  "Solution is optimal"

* Candidate solution
  Objective value      : -205.14285714285714
  Objective bound      : Inf
  Dual objective value : -205.1428571428571
  Primal solution :
    x : 15.428571428571429
    y[a] : 1.0
    y[b] : 1.0
  Dual solution :
    c1 : 1.7142857142857142

* Work counters
  Solve time (sec)   : 0.00008
```

15.2 Why did the solver stop?

Use`termination_status` to understand why the solver stopped.

```
julia> termination_status(model)
OPTIMAL::TerminationStatusCode = 1
```

The `MOI.TerminationStatusCode` enum describes the full list of statuses that could be returned.

Common return values include `OPTIMAL`, `LOCALLY_SOLVED`, `INFEASIBLE`, `DUAL_INFEASIBLE`, and `TIME_LIMIT`.

> **Info**
>
> A return status of `OPTIMAL` means the solver found (and proved) a globally optimal solution. A return status of `LOCALLY_SOLVED` means the solver found a locally optimal solution (which may also be globally optimal, but it could not prove so).

> **Warning**
>
> A return status of `DUAL_INFEASIBLE` does not guarantee that the primal is unbounded. When the dual is infeasible, the primal is unbounded if there exists a feasible primal solution.

Use `raw_status` to get a solver-specific string explaining why the optimization stopped:

```
julia> raw_status(model)
"Solution is optimal"
```

15.3 Primal solutions

Primal solution status

Use `primal_status` to return an `MOI.ResultStatusCode` enum describing the status of the primal solution.

```
julia> primal_status(model)
FEASIBLE_POINT::ResultStatusCode = 1
```

Other common returns are `NO_SOLUTION`, and `INFEASIBILITY_CERTIFICATE`. The first means that the solver doesn't have a solution to return, and the second means that the primal solution is a certificate of dual infeasibility (a primal unbounded ray).

You can also use `has_values`, which returns `true` if there is a solution that can be queried, and `false` otherwise.

```
julia> has_values(model)
true
```

Objective values

The objective value of a solved problem can be obtained via `objective_value`. The best known bound on the optimal objective value can be obtained via `objective_bound`. If the solver supports it, the value of the dual objective can be obtained via `dual_objective_value`.

```
julia> objective_value(model)
-205.14285714285714

julia> objective_bound(model)  # GLPK only implements objective bound for MIPs
Inf

julia> dual_objective_value(model)
-205.1428571428571
```

Primal solution values

If the solver has a primal solution to return, use `value` to access it:

```
julia> value(x)
15.428571428571429
```

Broadcast `value` over containers:

```
julia> value.(y)
1-dimensional DenseAxisArray{Float64,1,...} with index sets:
    Dimension 1, Symbol[:a, :b]
And data, a 2-element Array{Float64,1}:
 1.0
 1.0
```

`value` also works on expressions:

```
julia> value(my_expr)
100.57142857142857
```

and constraints:

```
julia> value(c1)
120.0
```

> **Info**
>
> Calling `value` on a constraint returns the constraint function evaluated at the solution.

15.4 Dual solutions

Dual solution status

Use `dual_status` to return an `MOI.ResultStatusCode` enum describing the status of the dual solution.

```
julia> dual_status(model)
FEASIBLE_POINT::ResultStatusCode = 1
```

Other common returns are `NO_SOLUTION`, and `INFEASIBILITY_CERTIFICATE`. The first means that the solver doesn't have a solution to return, and the second means that the dual solution is a certificate of primal infeasibility (a dual unbounded ray).

You can also use `has_duals`, which returns `true` if there is a solution that can be queried, and `false` otherwise.

```
julia> has_duals(model)
true
```

Dual solution values

If the solver has a dual solution to return, use `dual` to access it:

```
julia> dual(c1)
1.7142857142857142
```

Query the duals of variable bounds using `LowerBoundRef`, `UpperBoundRef`, and `FixRef`:

```
julia> dual(LowerBoundRef(x))
0.0

julia> dual.(UpperBoundRef.(y))
1-dimensional DenseAxisArray{Float64,1,...} with index sets:
    Dimension 1, Symbol[:a, :b]
And data, a 2-element Array{Float64,1}:
 -0.5714285714285694
  0.0
```

Warning

JuMP's definition of duality is independent of the objective sense. That is, the sign of feasible duals associated with a constraint depends on the direction of the constraint and not whether the problem is maximization or minimization. **This is a different convention from linear programming duality in some common textbooks.** If you have a linear program, and you want the textbook definition, you probably want to use `shadow_price` and `reduced_cost` instead.

```
julia> shadow_price(c1)
1.7142857142857142

julia> reduced_cost(x)
0.0

julia> reduced_cost.(y)
1-dimensional DenseAxisArray{Float64,1,...} with index sets:
    Dimension 1, Symbol[:a, :b]
And data, a 2-element Array{Float64,1}:
  0.5714285714285694
 -0.0
```

15.5 Recommended workflow

The recommended workflow for solving a model and querying the solution is something like the following:

```
if termination_status(model) == OPTIMAL
    println("Solution is optimal")
elseif termination_status(model) == TIME_LIMIT && has_values(model)
    println("Solution is suboptimal due to a time limit, but a primal solution is available")
else
    error("The model was not solved correctly.")
end
println("  objective value = ", objective_value(model))
if primal_status(model) == FEASIBLE_POINT
    println("  primal solution: x = ", value(x))
end
if dual_status(model) == FEASIBLE_POINT
    println("  dual solution: c1 = ", dual(c1))
end

# output

Solution is optimal
  objective value = -205.14285714285714
  primal solution: x = 15.428571428571429
  dual solution: c1 = 1.7142857142857142
```

15.6 OptimizeNotCalled errors

Modifing a model after calling `optimize!` will reset the model into the `MOI.OPTIMIZE_NOT_CALLED` state. If you attempt to query solution information, an `OptimizeNotCalled` error will be thrown.

If you are iteratively querying solution information and modifying a model, query all the results first, then modify the problem.

For example, instead of:

```
model = Model(GLPK.Optimizer)
@variable(model, x >= 0)
optimize!(model)
set_lower_bound(x, 1)  # This will modify the model
x_val = value(x)       # This will fail because the model has been modified
set_start_value(x, x_val)
```

do

```
model = Model(GLPK.Optimizer)
@variable(model, x >= 0)
optimize!(model)
x_val = value(x)
set_lower_bound(x, 1)
set_start_value(x, x_val)
```

15.7 Accessing attributes

MathOptInterface defines a large number of model attributes that can be queried. Some attributes can be directly accessed by getter functions. These include:

- `solve_time`
- `relative_gap`
- `simplex_iterations`
- `barrier_iterations`
- `node_count`

15.8 Sensitivity analysis for LP

Given an LP problem and an optimal solution corresponding to a basis, we can question how much an objective coefficient or standard form right-hand side coefficient (c.f., `normalized_rhs`) can change without violating primal or dual feasibility of the basic solution.

Note that not all solvers compute the basis, and for sensitivity analysis, the solver interface must implement `MOI.ConstraintBasisStatus`.

To give a simple example, we could analyze the sensitivity of the optimal solution to the following (non-degenerate) LP problem:

```
model = Model(GLPK.Optimizer)
@variable(model, x[1:2])
set_lower_bound(x[2], -0.5)
set_upper_bound(x[2], 0.5)
@constraint(model, c1, x[1] + x[2] <= 1)
@constraint(model, c2, x[1] - x[2] <= 1)
@objective(model, Max, x[1])
print(model)

# output
```

```
Max x[1]
Subject to
 c1 : x[1] + x[2] ≤ 1.0
 c2 : x[1] - x[2] ≤ 1.0
 x[2] ≥ -0.5
 x[2] ≤ 0.5
```

To analyze the sensitivity of the problem we could check the allowed perturbation ranges of, e.g., the cost coefficients and the right-hand side coefficient of the constraint `c1` as follows:

```
julia> optimize!(model)

julia> value.(x)
2-element Vector{Float64}:
 1.0
 0.0

julia> report = lp_sensitivity_report(model);

julia> x1_lo, x1_hi = report[x[1]]
(-1.0, Inf)

julia> println("The objective coefficient of x[1] could decrease by $(x1_lo) or increase by
↪  $(x1_hi).")
The objective coefficient of x[1] could decrease by -1.0 or increase by Inf.

julia> x2_lo, x2_hi = report[x[2]]
(-1.0, 1.0)

julia> println("The objective coefficient of x[2] could decrease by $(x2_lo) or increase by
↪  $(x2_hi).")
The objective coefficient of x[2] could decrease by -1.0 or increase by 1.0.

julia> c_lo, c_hi = report[c1]
(-1.0, 1.0)

julia> println("The RHS of c1 could decrease by $(c_lo) or increase by $(c_hi).")
The RHS of c1 could decrease by -1.0 or increase by 1.0.
```

The range associated with a variable is the range of the allowed perturbation of the corresponding objective coefficient. Note that the current primal solution remains optimal within this range; however the corresponding dual solution might change since a cost coefficient is perturbed. Similarly, the range associated with a constraint is the range of the allowed perturbation of the corresponding right-hand side coefficient. In this range the current dual solution remains optimal, but the optimal primal solution might change.

If the problem is degenerate, there are multiple optimal bases and hence these ranges might not be as intuitive and seem too narrow. E.g., a larger cost coefficient perturbation might not invalidate the optimality of the current primal solution. Moreover, if a problem is degenerate, due to finite precision, it can happen that, e.g., a perturbation seems to invalidate a basis even though it doesn't (again providing too narrow ranges). To prevent this, increase the `atol` keyword argument to `lp_sensitivity_report`. Note that this might make the ranges too wide for numerically challenging instances. Thus, do not blindly trust these ranges, especially not for highly degenerate or numerically unstable instances.

15.9 Conflicts

When the model you input is infeasible, some solvers can help you find the cause of this infeasibility by offering a conflict, i.e., a subset of the constraints that create this infeasibility. Depending on the solver, this can also be called an IIS (irreducible inconsistent subsystem).

The function `compute_conflict!` is used to trigger the computation of a conflict. Once this process is finished, the attribute `MOI.ConflictStatus` returns a `MOI.ConflictStatusCode`.

If there is a conflict, you can query from each constraint whether it participates in the conflict or not using the attribute `MOI.ConstraintConflictStatus`, which returns a `MOI.ConflictParticipationStatusCode`.

To create a new model containing only the constraints that participate in the conflict, use `copy_conflict`. It may be helpful to write this model to a file for easier debugging using `write_to_file`.

For instance, this is how you can use this functionality:

```
using JuMP
model = Model() # You must use a solver that supports conflict refining/IIS
# computation, like CPLEX or Gurobi
# e.g. using Gurobi; model = Model(Gurobi.Optimizer)
@variable(model, x >= 0)
@constraint(model, c1, x >= 2)
@constraint(model, c2, x <= 1)
optimize!(model)

# termination_status(model) will likely be INFEASIBLE,
# depending on the solver

compute_conflict!(model)
if MOI.get(model, MOI.ConflictStatus()) != MOI.CONFLICT_FOUND
    error("No conflict could be found for an infeasible model.")
end

# Both constraints should participate in the conflict.
MOI.get(model, MOI.ConstraintConflictStatus(), c1)
MOI.get(model, MOI.ConstraintConflictStatus(), c2)

# Get a copy of the model with only the constraints in the conflict.
new_model, reference_map = copy_conflict(model)
```

Conflicting constraints can be collected in a list and printed as follows:

```
conflict_constraint_list = ConstraintRef[]
for (F, S) in list_of_constraint_types(model)
    for con in all_constraints(model, F, S)
        if MOI.get(model, MOI.ConstraintConflictStatus(), con) == MOI.IN_CONFLICT
            push!(conflict_constraint_list, con)
            println(con)
        end
    end
end
```

15.10 Multiple solutions

Some solvers support returning multiple solutions. You can check how many solutions are available to query using `result_count`.

Functions for querying the solutions, e.g., `primal_status` and `value`, all take an additional keyword argument `result` which can be used to specify which result to return.

> **Warning**
>
> Even if `termination_status` is `OPTIMAL`, some of the returned solutions may be suboptimal! However, if the solver found at least one optimal solution, then `result = 1` will always return an optimal solution. Use `objective_value` to assess the quality of the remaining solutions.

```
using JuMP
model = Model()
@variable(model, x[1:10] >= 0)
# ... other constraints ...
optimize!(model)

if termination_status(model) != OPTIMAL
    error("The model was not solved correctly.")
end

an_optimal_solution = value.(x; result = 1)
optimal_objective = objective_value(model; result = 1)
for i in 2:result_count(model)
    @assert has_values(model; result = i)
    println("Solution $(i) = ", value.(x; result = i))
    obj = objective_value(model; result = i)
    println("Objective $(i) = ", obj)
    if isapprox(obj, optimal_objective; atol = 1e-8)
        print("Solution $(i) is also optimal!")
    end
end
```

15.11 Checking feasibility of solutions

To check the feasibility of a primal solution, use `primal_feasibility_report`, which takes a `model`, a dictionary mapping each variable to a primal solution value (defaults to the last solved solution), and a tolerance `atol` (defaults to `0.0`).

The function returns a dictionary which maps the infeasible constraint references to the distance between the primal value of the constraint and the nearest point in the corresponding set. A point is classed as infeasible if the distance is greater than the supplied tolerance `atol`.

```
julia> model = Model(GLPK.Optimizer);

julia> @variable(model, x >= 1, Int);

julia> @variable(model, y);

julia> @constraint(model, c1, x + y <= 1.95);

julia> point = Dict(x => 1.9, y => 0.06);

julia> primal_feasibility_report(model, point)
Dict{Any, Float64} with 2 entries:
  x integer         => 0.1
  c1 : x + y ≤ 1.95 => 0.01
```

```
julia> primal_feasibility_report(model, point; atol = 0.02)
Dict{Any, Float64} with 1 entry:
  x integer => 0.1
```

If the point is feasible, an empty dictionary is returned:

```
julia> primal_feasibility_report(model, Dict(x => 1.0, y => 0.0))
Dict{Any, Float64}()
```

To use the primal solution from a solve, omit the `point` argument:

```
julia> optimize!(model)

julia> primal_feasibility_report(model)
Dict{Any, Float64}()
```

Pass `skip_mising = true` to skip constraints which contain variables that are not in `point`:

```
julia> primal_feasibility_report(model, Dict(x => 2.1); skip_missing = true)
Dict{Any, Float64} with 1 entry:
  x integer => 0.1
```

You can also use the functional form, where the first argument is a function that maps variables to their primal values:

```
julia> optimize!(model)

julia> primal_feasibility_report(v -> value(v), model)
Dict{Any, Float64}()
```

Chapter 16

Nonlinear Modeling

JuMP has support for general smooth nonlinear (convex and nonconvex) optimization problems. JuMP is able to provide exact, sparse second-order derivatives to solvers. This information can improve solver accuracy and performance.

There are three main changes to solve nonlinear programs in JuMP.

- Use `@NLobjective` instead of `@objective`
- Use `@NLconstraint` instead of `@constraint`
- Use `@NLexpression` instead of `@expression`

> **Info**
>
> There are some restrictions on what syntax you can use in the `@NLxxx` macros. Make sure to read the Syntax notes.

16.1 Set a nonlinear objective

Use `@NLobjective` to set a nonlinear objective.

```
julia> @NLobjective(model, Min, exp(x[1]) - sqrt(x[2]))
```

16.2 Add a nonlinear constraint

Use `@NLconstraint` to add a nonlinear constraint.

```
julia> @NLconstraint(model, exp(x[1]) <= 1)
exp(x[1]) - 1.0 ≤ 0

julia> @NLconstraint(model, [i = 1:2], x[i]^i >= i)
2-element Vector{NonlinearConstraintRef{ScalarShape}}:
 x[1] ^ 1.0 - 1.0 ≥ 0
 x[2] ^ 2.0 - 2.0 ≥ 0

julia> @NLconstraint(model, con[i = 1:2], prod(x[j] for j = 1:i) == i)
2-element Vector{NonlinearConstraintRef{ScalarShape}}:
 (*)(x[1]) - 1.0 = 0
 x[1] * x[2] - 2.0 = 0
```

Info

You can only create nonlinear constraints with <=, >=, and ==. More general Nonlinear-in-Set constraints are not supported.

16.3 Create a nonlinear expression

Use `@NLexpression` to create nonlinear expression objects. The syntax is identical to `@expression`, except that the expression can contain nonlinear terms.

```
julia> expr = @NLexpression(model, exp(x[1]) + sqrt(x[2]))
"subexpression[1]: exp(x[1]) + sqrt(x[2])"

julia> my_anon_expr = @NLexpression(model, [i = 1:2], sin(x[i]))
2-element Vector{NonlinearExpression}:
 "subexpression[2]: sin(x[1])"
 "subexpression[3]: sin(x[2])"

julia> @NLexpression(model, my_expr[i = 1:2], sin(x[i]))
2-element Vector{NonlinearExpression}:
 "subexpression[4]: sin(x[1])"
 "subexpression[5]: sin(x[2])"
```

Nonlinear expression can be used in `@NLobjective`, `@NLconstraint`, and even nested in other `@NLexpression`s.

```
julia> @NLobjective(model, Min, expr^2 + 1)

julia> @NLconstraint(model, [i = 1:2], my_expr[i] <= i)
2-element Vector{NonlinearConstraintRef{ScalarShape}}:
 subexpression[4] - 1.0 ≤ 0
 subexpression[5] - 2.0 ≤ 0

julia> @NLexpression(model, nested[i = 1:2], sin(my_expr[i]))
2-element Vector{NonlinearExpression}:
 "subexpression[6]: sin(subexpression[4])"
 "subexpression[7]: sin(subexpression[5])"
```

16.4 Create a nonlinear parameter

For nonlinear models only, JuMP offers a syntax for explicit "parameter" objects, which are constants in the model that can be efficiently updated between solves.

Nonlinear parameters are declared by using the `@NLparameter` macro and may be indexed by arbitrary sets analogously to JuMP variables and expressions.

The initial value of the parameter must be provided on the right-hand side of the == sign.

```
julia> @NLparameter(model, p[i = 1:2] == i)
2-element Vector{NonlinearParameter}:
 "parameter[1] == 1.0"
 "parameter[2] == 2.0"
```

Create anonymous parameters using the `value` keyword:

```
julia> anon_parameter = @NLparameter(model, value = 1)
"parameter[3] == 1.0"
```

> **Info**
>
> A parameter is not an optimization variable. It must be fixed to a value with ==. If you want a parameter that is <= or >=, create a variable instead using `@variable`.

Use `value` and `set_value` to query or update the value of a parameter.

```
julia> value.(p)
2-element Vector{Float64}:
 1.0
 2.0

julia> set_value(p[2], 3.0)
3.0

julia> value.(p)
2-element Vector{Float64}:
 1.0
 3.0
```

Nonlinear parameters can be used within nonlinear macros only:

```
julia> @objective(model, Max, p[1] * x)
ERROR: MethodError: no method matching *(::NonlinearParameter, ::VariableRef)
[...]

julia> @NLobjective(model, Max, p[1] * x)

julia> @expression(model, my_expr, p[1] * x^2)
ERROR: MethodError: no method matching *(::NonlinearParameter, ::QuadExpr)
Closest candidates are:
[...]

julia> @NLexpression(model, my_nl_expr, p[1] * x^2)
"subexpression[1]: parameter[1] * x ^ 2.0"
```

When to use a parameter

Nonlinear parameters are useful when solving nonlinear models in a sequence:

```
using JuMP, Ipopt
model = Model(Ipopt.Optimizer)
set_silent(model)
@variable(model, z)
@NLparameter(model, x == 1.0)
@NLobjective(model, Min, (z - x)^2)
optimize!(model)
@show value(z) # Equals 1.0.

# Now, update the value of x to solve a different problem.
set_value(x, 5.0)
```

```
optimize!(model)
@show value(z) # Equals 5.0
```

```
value(z) = 1.0
value(z) = 5.0
```

Info

Using nonlinear parameters can be faster than creating a new model from scratch with updated data because JuMP is able to avoid repeating a number of steps in processing the model before handing it off to the solver.

16.5 Syntax notes

The syntax accepted in nonlinear macros is more restricted than the syntax for linear and quadratic macros. We note some important points below.

No operator overloading

There is no operator overloading provided to build up nonlinear expressions. For example, if x is a JuMP variable, the code 3x will return an `AffExpr` object that can be used inside of future expressions and linear constraints. However, the code `sin(x)` is an error. All nonlinear expressions must be inside of macros.

```
julia> expr = sin(x) + 1
ERROR: sin is not defined for type AbstractVariableRef. Are you trying to build a nonlinear problem?
↪  Make sure you use @NLconstraint/@NLobjective.
[...]

julia> expr = @NLexpression(model, sin(x) + 1)
"subexpression[1]: sin(x) + 1.0"
```

Scalar operations only

With the exception of the splatting syntax discussed below, all expressions must be simple scalar operations. You cannot use `dot`, matrix-vector products, vector slices, etc.

```
julia> @NLobjective(model, Min, c' * x + 3y)
ERROR: Unexpected array [1 2] in nonlinear expression. Nonlinear expressions may contain only scalar
↪  expressions.
[...]
```

Translate vector operations into explicit `sum()` operations:

```
julia> @NLobjective(model, Min, sum(c[i] * x[i] for i = 1:2) + 3y)
```

Or use an `@expression`:

```
julia> @expression(model, expr, c' * x)
x[1] + 2 x[2]

julia> @NLobjective(model, Min, expr + 3y)
```

Splatting

The splatting operator `...` is recognized in a very restricted setting for expanding function arguments. The expression splatted can be only a symbol. More complex expressions are not recognized.

```
julia> model = Model();

julia> @variable(model, x[1:3]);

julia> @NLconstraint(model, *(x...) <= 1.0)
x[1] * x[2] * x[3] - 1.0 ≤ 0

julia> @NLconstraint(model, *((x / 2)...) <= 0.0)
ERROR: LoadError: Unexpected expression in (*)(x / 2...). JuMP supports splatting only symbols. For
↪  example, x... is ok, but (x + 1)..., [x; y]... and g(f(y)...) are not.
```

16.6 User-defined Functions

JuMP's library of recognized univariate functions is derived from the Calculus.jl package.

In addition to this list of functions, it is possible to register custom user-defined nonlinear functions.

> **Tip**
>
> User-defined functions can be used anywhere in `@NLobjective`, `@NLconstraint`, and `@NLexpression`.

> **Tip**
>
> JuMP will attempt to automatically register functions it detects in your nonlinear expressions, which means that in most cases, manually registering a function is not needed. Two exceptions are if you want to provide custom derivatives, or if the function is not available in the scope of the nonlinear expression.

> **Warning**
>
> User-defined functions must return a scalar output. For a work-around, see User-defined functions with vector outputs.

Automatic differentiation

JuMP does not support black-box optimization, so all user-defined functions must provide derivatives in some form.

Fortunately, JuMP supports **automatic differentiation of user-defined functions**, a feature to our knowledge not available in any comparable modeling systems.

> **Info**
>
> Automatic differentiation is not finite differencing. JuMP's automatically computed derivatives are not subject to approximation error.

JuMP uses ForwardDiff.jl to perform automatic differentiation; see the ForwardDiff.jl documentation for a description of how to write a function suitable for automatic differentiation.

> **Warning**
>
> Get an error like `No method matching Float64(::ForwardDiff.Dual)`? Read this section, and see the guidelines at ForwardDiff.jl.

The most common error is that your user-defined function is not generic with respect to the number type, i.e., don't assume that the input to the function is `Float64`.

```
f(x::Float64) = 2 * x  # This will not work.
f(x::Real)    = 2 * x  # This is good.
f(x)          = 2 * x  # This is also good.
```

Another reason you may encounter this error is if you create arrays inside your function which are `Float64`.

```
function bad_f(x...)
    y = zeros(length(x))  # This constructs an array of `Float64`!
    for i = 1:length(x)
        y[i] = x[i]^i
    end
    return sum(y)
end

function good_f(x::T...) where {T<:Real}
    y = zeros(T, length(x))  # Construct an array of type `T` instead!
    for i = 1:length(x)
        y[i] = x[i]^i
    end
    return sum(y)
end
```

Register a function

To register a user-defined function with derivatives computed by automatic differentiation, use the `register` method as in the following example:

```
square(x) = x^2
f(x, y) = (x - 1)^2 + (y - 2)^2

model = Model()

register(model, :square, 1, square; autodiff = true)
register(model, :my_f, 2, f; autodiff = true)

@variable(model, x[1:2] >= 0.5)
@NLobjective(model, Min, my_f(x[1], square(x[2])))
```

The above code creates a JuMP model with the objective function `(x[1] - 1)^2 + (x[2]^2 - 2)^2`. The arguments to `register` are:

1. The model for which the functions are registered.
2. A Julia symbol object which serves as the name of the user-defined function in JuMP expressions.
3. The number of input arguments that the function takes.

4. The Julia method which computes the function
5. A flag to instruct JuMP to compute exact gradients automatically.

Tip

The symbol `:my_f` doesn't have to match the name of the function `f`. However, it's generally more readable if it does. Make sure you use `my_f` and not `f` in the macros.

Warning

If you use multi-variate user-defined functions, JuMP will disable second-derivative information. This can lead to significant slow-downs in some cases. Only use a user-defined function if you cannot write out the expression algebraically in the macro.

Warning

User-defined functions cannot be re-registered and will not update if you modify the underlying Julia function. If you want to change a user-defined function between solves, rebuild the model or use a different name. To use a different name programmatically, see Raw expression input.

Register a function and gradient

Forward-mode automatic differentiation as implemented by ForwardDiff.jl has a computational cost that scales linearly with the number of input dimensions. As such, it is not the most efficient way to compute gradients of user-defined functions if the number of input arguments is large. In this case, users may want to provide their own routines for evaluating gradients.

Univariate functions

For univariate functions, the gradient function `∇f` should return a number that represents the first-order derivative:

```
f(x) = x^2
∇f(x) = 2x
model = Model()
register(model, :my_square, 1, f, ∇f; autodiff = true)
@variable(model, x >= 0)
@NLobjective(model, Min, my_square(x))
```

If `autodiff = true`, JuMP will use automatic differentiation to compute the hessian.

Multivariate functions

For multivariate functions, the gradient function `∇f` must take a gradient vector as the first argument that is filled in-place:

```
f(x, y) = (x - 1)^2 + (y - 2)^2
function ∇f(g::AbstractVector{T}, x::T, y::T) where {T}
    g[1] = 2 * (x - 1)
    g[2] = 2 * (y - 2)
    return
end
```

```
model = Model()
register(model, :my_square, 2, f, ∇f)
@variable(model, x[1:2] >= 0)
@NLobjective(model, Min, my_square(x[1], x[2]))
```

> **Warning**
>
> Make sure the first argument to `∇f` supports an `AbstractVector`, and do not assume the input is `Float64`.

Register a function, gradient, and hessian

> **Warning**
>
> The ability to explicitly register a hessian is only available for univariate functions.

Instead of automatically differentiating the hessian, you can instead pass a function which returns a number representing the second-order derivative.

```
f(x) = x^2
∇f(x) = 2x
∇²f(x) = 2
model = Model()
register(model, :my_square, 1, f, ∇f, ∇²f)
@variable(model, x >= 0)
@NLobjective(model, Min, my_square(x))
```

User-defined functions with vector inputs

User-defined functions which take vectors as input arguments (e.g. `f(x::Vector)`) are not supported. Instead, use Julia's splatting syntax to create a function with scalar arguments. For example, instead of

```
f(x::Vector) = sum(x[i]^i for i in 1:length(x))
```

define:

```
f(x...) = sum(x[i]^i for i in 1:length(x))
```

This function `f` can be used in a JuMP model as follows:

```
model = Model()
@variable(model, x[1:5] >= 0)
f(x...) = sum(x[i]^i for i in 1:length(x))
register(model, :f, 5, f; autodiff = true)
@NLobjective(model, Min, f(x...))
```

> **Tip**
>
> Make sure to read the syntax restrictions of Splatting.

16.7 Factors affecting solution time

The execution time when solving a nonlinear programming problem can be divided into two parts, the time spent in the optimization algorithm (the solver) and the time spent evaluating the nonlinear functions and corresponding derivatives. Ipopt explicitly displays these two timings in its output, for example:

```
Total CPU secs in IPOPT (w/o function evaluations)   =      7.412
Total CPU secs in NLP function evaluations           =      2.083
```

For Ipopt in particular, one can improve the performance by installing advanced sparse linear algebra packages, see Installation Guide. For other solvers, see their respective documentation for performance tips.

The function evaluation time, on the other hand, is the responsibility of the modeling language. JuMP computes derivatives by using reverse-mode automatic differentiation with graph coloring methods for exploiting sparsity of the Hessian matrix [1]. As a conservative bound, JuMP's performance here currently may be expected to be within a factor of 5 of AMPL's.

16.8 Querying derivatives from a JuMP model

For some advanced use cases, one may want to directly query the derivatives of a JuMP model instead of handing the problem off to a solver. Internally, JuMP implements the `MOI.AbstractNLPEvaluator` interface. To obtain an NLP evaluator object from a JuMP model, use `NLPEvaluator`. `index` returns the `MOI.VariableIndex` corresponding to a JuMP variable. `MOI.VariableIndex` itself is a type-safe wrapper for `Int64` (stored in the `.value` field.)

For example:

```
raw_index(v::MOI.VariableIndex) = v.value
model = Model()
@variable(model, x)
@variable(model, y)
@NLobjective(model, Min, sin(x) + sin(y))
values = zeros(2)
x_index = raw_index(JuMP.index(x))
y_index = raw_index(JuMP.index(y))
values[x_index] = 2.0
values[y_index] = 3.0
d = NLPEvaluator(model)
MOI.initialize(d, [:Grad])
MOI.eval_objective(d, values) # == sin(2.0) + sin(3.0)

# output
1.0504174348855488
```

```
∇f = zeros(2)
MOI.eval_objective_gradient(d, ∇f, values)
(∇f[x_index], ∇f[y_index]) # == (cos(2.0), cos(3.0))

# output
(-0.4161468365471424, -0.9899924966004454)
```

Only nonlinear constraints (those added with `@NLconstraint`), and nonlinear objectives (added with `@NLobjective`) exist in the scope of the `NLPEvaluator`.

The `NLPEvaluator` does not evaluate derivatives of linear or quadratic constraints or objectives.

The index method applied to a nonlinear constraint reference object returns its index as a NonlinearConstraintIndex. The .value field of NonlinearConstraintIndex stores the raw integer index. For example:

```
julia> model = Model();

julia> @variable(model, x);

julia> @NLconstraint(model, cons1, sin(x) <= 1);

julia> @NLconstraint(model, cons2, x + 5 == 10);

julia> typeof(cons1)
NonlinearConstraintRef{ScalarShape} (alias for ConstraintRef{Model, NonlinearConstraintIndex,
↪  ScalarShape})

julia> index(cons1)
NonlinearConstraintIndex(1)

julia> index(cons2)
NonlinearConstraintIndex(2)
```

Note that for one-sided nonlinear constraints, JuMP subtracts any values on the right-hand side when computing expressions. In other words, one-sided nonlinear constraints are always transformed to have a right-hand side of zero.

This method of querying derivatives directly from a JuMP model is convenient for interacting with the model in a structured way, e.g., for accessing derivatives of specific variables. For example, in statistical maximum likelihood estimation problems, one is often interested in the Hessian matrix at the optimal solution, which can be queried using the NLPEvaluator.

16.9 Raw expression input

> **Warning**
>
> This section requires advanced knowledge of Julia's Expr. You should read the Expressions and evaluation section of the Julia documentation first.

In addition to the @NLexpression, @NLobjective and @NLconstraint macros, it is also possible to provide Julia Expr objects directly by using add_NL_expression, set_NL_objective and add_NL_constraint.

This input form may be useful if the expressions are generated programmatically.

Add a nonlinear expression

Use add_NL_expression to add a nonlinear expression to the model.

```
julia> @variable(model, x)
x

julia> expr = :($(x) + sin($(x)^2))
:(x + sin(x ^ 2))

julia> expr_ref = add_NL_expression(model, expr)
"subexpression[1]: x + sin(x ^ 2.0)"
```

This is equivalent to

```
julia> expr_ref = @NLexpression(model, x + sin(x^2))
"subexpression[1]: x + sin(x ^ 2.0)"
```

> **Note**
>
> You must interpolate the variables directly into the expression `expr`.

Set the objective function

Use `set_NL_objective` to set a nonlinear objective.

```
julia> expr = :($(x) + $(x)^2)
:(x + x ^ 2)

julia> set_NL_objective(model, MOI.MIN_SENSE, expr)
```

This is equivalent to

```
julia> @NLobjective(model, Min, x + x^2)
```

> **Note**
>
> You must use `MOI.MIN_SENSE` or `MOI.MAX_SENSE` instead of `Min` and `Max`.

Add a constraint

Use `add_NL_constraint` to add a nonlinear constraint.

```
julia> expr = :($(x) + $(x)^2)
:(x + x ^ 2)

julia> add_NL_constraint(model, :($(expr) <= 1))
(x + x ^ 2.0) - 1.0 ≤ 0
```

This is equivalent to

```
julia> @NLconstraint(model, Min, x + x^2 <= 1)
(x + x ^ 2.0) - 1.0 ≤ 0
```

More complicated examples

Raw expression input is most useful when the expressions are generated programmatically, often in conjunction with user-defined functions.

As an example, we construct a model with the nonlinear constraints `f(x) <= 1`, where `f(x) = x^2` and `f(x) = sin(x)^2`:

```
julia> function main(functions::Vector{Function})
           model = Model()
           @variable(model, x)
           for (i, f) in enumerate(functions)
               f_sym = Symbol("f_$(i)")
               register(model, f_sym, 1, f; autodiff = true)
               add_NL_constraint(model, :($(f_sym)($(x)) <= 1))
           end
           print(model)
           return
       end
main (generic function with 1 method)

julia> main([x -> x^2, x -> sin(x)^2])
Feasibility
Subject to
 f_1(x) - 1.0 ≤ 0
 f_2(x) - 1.0 ≤ 0
```

As another example, we construct a model with the constraint `x^2 + sin(x)^2 <= 1`:

```
julia> function main(functions::Vector{Function})
           model = Model()
           @variable(model, x)
           expr = Expr(:call, :+)
           for (i, f) in enumerate(functions)
               f_sym = Symbol("f_$(i)")
               register(model, f_sym, 1, f; autodiff = true)
               push!(expr.args, :($(f_sym)($(x))))
           end
           add_NL_constraint(model, :($(expr) <= 1))
           print(model)
           return
       end
main (generic function with 1 method)

julia> main([x -> x^2, x -> sin(x)^2])
Feasibility
Subject to
 (f_1(x) + f_2(x)) - 1.0 ≤ 0
```

[1] Dunning, Huchette, and Lubin, "JuMP: A Modeling Language for Mathematical Optimization", SIAM Review, PDF.

Chapter 17

Solver-independent Callbacks

Many mixed-integer (linear, conic, and nonlinear) programming solvers offer the ability to modify the solve process. Examples include changing branching decisions in branch-and-bound, adding custom cutting planes, providing custom heuristics to find feasible solutions, or implementing on-demand separators to add new constraints only when they are violated by the current solution (also known as lazy constraints).

While historically this functionality has been limited to solver-specific interfaces, JuMP provides solver-independent support for three types of callbacks:

1. lazy constraints
2. user-cuts
3. heuristic solutions

17.1 Available solvers

Solver-independent callback support is limited to a few solvers. This includes CPLEX, GLPK, Gurobi, and Xpress.

Warning

While JuMP provides a solver-independent way of accessing callbacks, you should not assume that you will see identical behavior when running the same code on different solvers. For example, some solvers may ignore user-cuts for various reasons, while other solvers may add every user-cut. Read the underlying solver's callback documentation to understand details specific to each solver.

Tip

This page discusses solver-independent callbacks. However, each solver listed above also provides a solver-dependent callback to provide access to the full range of solver-specific features. Consult the solver's README for an example of how to use the solver-dependent callback. This will require you to understand the C interface of the solver.

17.2 Things you can and cannot do during solver-independent callbacks

There is a very limited range of things you can do during a callback. Only use the functions and macros explicitly stated in this page of the documentation, or in the Callbacks tutorial.

Using any other part of the JuMP API (e.g., adding a constraint with `@constraint` or modifying a variable bound with `set_lower_bound`) is undefined behavior, and your solver may throw an error, return an incorrect solution, or result in a segfault that aborts Julia.

In each of the three solver-independent callbacks, there are two things you may query:

- `callback_node_status` returns an `MOI.CallbackNodeStatusCode` enum indicating if the current primal solution is integer feasible.
- `callback_value` returns the current primal solution of a variable.

If you need to query any other information, use a solver-dependent callback instead. Each solver supporting a solver-dependent callback has information on how to use it in the README of their Github repository.

If you want to modify the problem in a callback, you must use a lazy constraint.

> **Warning**
>
> You can only set each callback once. Calling set twice will over-write the earlier callback. In addition, if you use a solver-independent callback, you cannot set a solver-dependent callback.

17.3 Lazy constraints

Lazy constraints are useful when the full set of constraints is too large to explicitly include in the initial formulation. When a MIP solver reaches a new solution, for example with a heuristic or by solving a problem at a node in the branch-and-bound tree, it will give the user the chance to provide constraint(s) that would make the current solution infeasible. For some more information about lazy constraints, see this blog post by Paul Rubin.

A lazy constraint callback can be set using the following syntax:

```
model = Model(GLPK.Optimizer)
@variable(model, x <= 10, Int)
@objective(model, Max, x)
function my_callback_function(cb_data)
    status = callback_node_status(cb_data, model)
    if status == MOI.CALLBACK_NODE_STATUS_FRACTIONAL
        # `callback_value(cb_data, x)` is not integer (to some tolerance).
        # If, for example, your lazy constraint generator requires an
        # integer-feasible primal solution, you can add a `return` here.
        return
    elseif status == MOI.CALLBACK_NODE_STATUS_INTEGER
        # `callback_value(cb_data, x)` is integer (to some tolerance).
    else
        @assert status == MOI.CALLBACK_NODE_STATUS_UNKNOWN
        # `callback_value(cb_data, x)` might be fractional or integer.
    end
    x_val = callback_value(cb_data, x)
    if x_val > 2 + 1e-6
        con = @build_constraint(x <= 2)
        MOI.submit(model, MOI.LazyConstraint(cb_data), con)
    end
end
MOI.set(model, MOI.LazyConstraintCallback(), my_callback_function)
```

Info

The lazy constraint callback may be called at fractional or integer nodes in the branch-and-bound tree. There is no guarantee that the callback is called at every primal solution.

Warning

Only add a lazy constraint if your primal solution violates the constraint. Adding the lazy constraint irrespective of feasibility may result in the solver returning an incorrect solution, or lead to a large number of constraints being added, slowing down the solution process.

```
model = Model(GLPK.Optimizer)
@variable(model, x <= 10, Int)
@objective(model, Max, x)
function bad_callback_function(cb_data)
    # Don't do this!
    con = @build_constraint(x <= 2)
    MOI.submit(model, MOI.LazyConstraint(cb_data), con)
end
function good_callback_function(cb_data)
    if callback_value(x) > 2
        con = @build_constraint(x <= 2)
        MOI.submit(model, MOI.LazyConstraint(cb_data), con)
    end
end
MOI.set(model, MOI.LazyConstraintCallback(), good_callback_function)
```

Warning

During the solve, a solver may visit a point that was cut off by a previous lazy constraint, e.g., because the earlier lazy constraint was removed during presolve. However, the solver will not terminate until it reaches a solution that satisfies all added lazy constraints.

17.4 User cuts

User cuts, or simply cuts, provide a way for the user to tighten the LP relaxation using problem-specific knowledge that the solver cannot or is unable to infer from the model. Just like with lazy constraints, when a MIP solver reaches a new node in the branch-and-bound tree, it will give the user the chance to provide cuts to make the current relaxed (fractional) solution infeasible in the hopes of obtaining an integer solution. For more details about the difference between user cuts and lazy constraints see the aforementioned blog post.

A user-cut callback can be set using the following syntax:

```
model = Model(GLPK.Optimizer)
@variable(model, x <= 10.5, Int)
@objective(model, Max, x)
function my_callback_function(cb_data)
    x_val = callback_value(cb_data, x)
    con = @build_constraint(x <= floor(x_val))
    MOI.submit(model, MOI.UserCut(cb_data), con)
end
MOI.set(model, MOI.UserCutCallback(), my_callback_function)
```

Warning

Your user cuts should not change the set of integer feasible solutions. Equivalently, your cuts can only remove fractional solutions. If you add a cut that removes an integer solution, the solver may return an incorrect solution.

Info

The user-cut callback may be called at fractional nodes in the branch-and-bound tree. There is no guarantee that the callback is called at every fractional primal solution.

17.5 Heuristic solutions

Integer programming solvers frequently include heuristics that run at the nodes of the branch-and-bound tree. They aim to find integer solutions quicker than plain branch-and-bound would to tighten the bound, allowing us to fathom nodes quicker and to tighten the integrality gap.

Some heuristics take integer solutions and explore their "local neighborhood" (e.g., flipping binary variables, fix some variables and solve a smaller MILP) and others take fractional solutions and attempt to round them in an intelligent way.

You may want to add a heuristic of your own if you have some special insight into the problem structure that the solver is not aware of, e.g. you can consistently take fractional solutions and intelligently guess integer solutions from them.

A heuristic solution callback can be set using the following syntax:

```
model = Model(GLPK.Optimizer)
@variable(model, x <= 10.5, Int)
@objective(model, Max, x)
function my_callback_function(cb_data)
    x_val = callback_value(cb_data, x)
    status = MOI.submit(
        model, MOI.HeuristicSolution(cb_data), [x], [floor(Int, x_val)]
    )
    println("I submitted a heuristic solution, and the status was: ", status)
end
MOI.set(model, MOI.HeuristicCallback(), my_callback_function)
```

The third argument to `submit` should be a vector of JuMP variables, and the fourth argument should be a vector of values corresponding to each variable.

`MOI.submit` returns an enum that depends on whether the solver accepted the solution. The possible return codes are:

- `MOI.HEURISTIC_SOLUTION_ACCEPTED`
- `MOI.HEURISTIC_SOLUTION_REJECTED`
- `MOI.HEURISTIC_SOLUTION_UNKNOWN`

Warning

Some solvers may accept partial solutions. Others require a feasible integer solution for every variable. If in doubt, provide a complete solution.

Info

The heuristic solution callback may be called at fractional nodes in the branch-and-bound tree. There is no guarantee that the callback is called at every fractional primal solution.

Part IV

API Reference

Chapter 18

Models

More information can be found in the Models section of the manual.

18.1 Constructors

JuMP.Model – Type.

```
Model
```

A mathematical model of an optimization problem.

source

JuMP.direct_model – Function.

```
direct_model(backend::MOI.ModelLike)
```

Return a new JuMP model using backend to store the model and solve it.

As opposed to the Model constructor, no cache of the model is stored outside of backend and no bridges are automatically applied to backend.

Notes

The absence of a cache reduces the memory footprint but, it is important to bear in mind the following implications of creating models using this direct mode:

- When backend does not support an operation, such as modifying constraints or adding variables/-constraints after solving, an error is thrown. For models created using the Model constructor, such situations can be dealt with by storing the modifications in a cache and loading them into the optimizer when `optimize!` is called.
- No constraint bridging is supported by default.
- The optimizer used cannot be changed the model is constructed.
- The model created cannot be copied.

source

```
direct_model(factory::MOI.OptimizerWithAttributes)
```

Create a direct_model using `factory`, a `MOI.OptimizerWithAttributes` object created by optimizer_with_attributes.

Example

```
model = direct_model(
    optimizer_with_attributes(
        Gurobi.Optimizer,
        "Presolve" => 0,
        "OutputFlag" => 1,
    )
)
```

is equivalent to:

```
model = direct_model(Gurobi.Optimizer())
set_optimizer_attribute(model, "Presolve", 0)
set_optimizer_attribute(model, "OutputFlag", 1)
```

source

18.2 Enums

JuMP.ModelMode – Type.

```
ModelMode
```

An enum to describe the state of the CachingOptimizer inside a JuMP model.

source

JuMP.AUTOMATIC – Constant.

moi_backend field holds a CachingOptimizer in AUTOMATIC mode.

source

JuMP.MANUAL – Constant.

moi_backend field holds a CachingOptimizer in MANUAL mode.

source

JuMP.DIRECT – Constant.

moi_backend field holds an AbstractOptimizer. No extra copy of the model is stored. The moi_backend must support add_constraint etc.

source

18.3 Basic functions

JuMP.backend – Function.

```
backend(model::Model)
```

Return the lower-level MathOptInterface model that sits underneath JuMP. This model depends on which operating mode JuMP is in (see mode).

- If JuMP is in DIRECT mode (i.e., the model was created using direct_model), the backend will be the optimizer passed to direct_model.

- If JuMP is in `MANUAL` or `AUTOMATIC` mode, the backend is a `MOI.Utilities.CachingOptimizer`.

This function should only be used by advanced users looking to access low-level MathOptInterface or solver-specific functionality.

Notes

If JuMP is not in `DIRECT` mode, the type returned by `backend` may change between any JuMP releases. Therefore, only use the public API exposed by MathOptInterface, and do not access internal fields. If you require access to the innermost optimizer, see `unsafe_backend`. Alternatively, use `direct_model` to create a JuMP model in `DIRECT` mode.

See also: `unsafe_backend`.

source

`JuMP.unsafe_backend` – Function.

```
unsafe_backend(model::Model)
```

Return the innermost optimizer associated with the JuMP model `model`.

This function should only be used by advanced users looking to access low-level solver-specific functionality. It has a high-risk of incorrect usage. We strongly suggest you use the alternative suggested below.

See also: `backend`.

Unsafe behavior

This function is unsafe for two main reasons.

First, the formulation and order of variables and constraints in the unsafe backend may be different to the variables and constraints in `model`. This can happen because of bridges, or because the solver requires the variables or constraints in a specific order. In addition, the variable or constraint index returned by `index` at the JuMP level may be different to the index of the corresponding variable or constraint in the `unsafe_backend`. There is no solution to this. Use the alternative suggested below instead.

Second, the `unsafe_backend` may be empty, or lack some modifications made to the JuMP model. Thus, before calling `unsafe_backend` you should first call `MOI.Utilities.attach_optimizer` to ensure that the backend is synchronized with the JuMP model.

```
MOI.Utilities.attach_optimizer(model)
inner = unsafe_backend(model)
```

Moreover, if you modify the JuMP model, the reference you have to the backend (i.e., `inner` in the example above) may be out-dated, and you should call `MOI.Utilities.attach_optimizer` again.

This function is also unsafe in the reverse direction: if you modify the unsafe backend, e.g., by adding a new constraint to `inner`, the changes may be silently discarded by JuMP when the JuMP `model` is modified or solved.

Alternative

Instead of `unsafe_backend`, create a model using `direct_model` and call `backend` instead.

For example, instead of:

```
model = Model(GLPK.Optimizer)
@variable(model, x >= 0)
MOI.Utilities.attach_optimizer(model)
glpk = unsafe_backend(model)
```

Use:

```
model = direct_model(GLPK.Optimizer())
@variable(model, x >= 0)
glpk = backend(model)  # No need to call `attach_optimizer`.
```

source

JuMP.name – Method.

```
name(model::AbstractModel)
```

Return the MOI.Name attribute of model's backend, or a default if empty.

source

JuMP.solver_name – Function.

```
solver_name(model::Model)
```

If available, returns the SolverName property of the underlying optimizer.

Returns "No optimizer attached" in AUTOMATIC or MANUAL modes when no optimizer is attached.

Returns "SolverName() attribute not implemented by the optimizer." if the attribute is not implemented.

source

Base.empty! – Method.

```
empty!(model::Model)::Model
```

Empty the model, that is, remove all variables, constraints and model attributes but not optimizer attributes. Always return the argument.

Note: removes extensions data.

source

JuMP.mode – Function.

```
mode(model::Model)
```

Return the ModelMode (DIRECT, AUTOMATIC, or MANUAL) of model.

source

JuMP.object_dictionary – Function.

```
object_dictionary(model::Model)
```

Return the dictionary that maps the symbol name of a variable, constraint, or expression to the corresponding object.

Objects are registered to a specific symbol in the macros. For example, @variable(model, x[1:2, 1:2]) registers the array of variables x to the symbol :x.

This method should be defined for any subtype of AbstractModel.

source

JuMP.unregister – Function.

```
unregister(model::Model, key::Symbol)
```

Unregister the name key from `model` so that a new variable, constraint, or expression can be created with the same key.

Note that this will not delete the object `model[key]`; it will just remove the reference at `model[key]`. To delete the object, use

```
delete(model, model[key])
unregister(model, key)
```

See also: `object_dictionary`.

Examples

```
julia> @variable(model, x)
x

julia> @variable(model, x)
ERROR: An object of name x is already attached to this model. If
this is intended, consider using the anonymous construction syntax,
e.g., `x = @variable(model, [1:N], ...)` where the name of the object
does not appear inside the macro.

Alternatively, use `unregister(model, :x)` to first unregister the
existing name from the model. Note that this will not delete the object;
it will just remove the reference at `model[:x]`.
[...]

julia> num_variables(model)
1

julia> unregister(model, :x)

julia> @variable(model, x)
x

julia> num_variables(model)
2
```

source

JuMP.latex_formulation – Function.

```
latex_formulation(model::AbstractModel)
```

Wrap `model` in a type so that it can be pretty-printed as `text/latex` in a notebook like IJulia, or in Documenter.

To render the model, end the cell with `latex_formulation(model)`, or call `display(latex_formulation(model))` in to force the display of the model from inside a function.

source

18.4 Working with attributes

JuMP.set_optimizer – Function.

```
set_optimizer(
    model::Model,
    optimizer_factory;
    add_bridges::Bool = true,
)
```

Creates an empty MathOptInterface.AbstractOptimizer instance by calling optimizer_factory() and sets it as the optimizer of model. Specifically, optimizer_factory must be callable with zero arguments and return an empty MathOptInterface.AbstractOptimizer.

If add_bridges is true, constraints and objectives that are not supported by the optimizer are automatically bridged to equivalent supported formulation. Passing add_bridges = false can improve performance if the solver natively supports all of the elements in model.

See set_optimizer_attributes and set_optimizer_attribute for setting solver-specific parameters of the optimizer.

Examples

```
model = Model()
set_optimizer(model, GLPK.Optimizer)
set_optimizer(model, GLPK.Optimizer; add_bridges = false)
```

source

JuMP.optimizer_with_attributes – Function.

```
optimizer_with_attributes(optimizer_constructor, attrs::Pair...)
```

Groups an optimizer constructor with the list of attributes attrs. Note that it is equivalent to MOI.OptimizerWithAttributes.

When provided to the Model constructor or to set_optimizer, it creates an optimizer by calling optimizer_constructor(), and then sets the attributes using set_optimizer_attribute.

Example

```
model = Model(
    optimizer_with_attributes(
        Gurobi.Optimizer, "Presolve" => 0, "OutputFlag" => 1
    )
)
```

is equivalent to:

```
model = Model(Gurobi.Optimizer)
set_optimizer_attribute(model, "Presolve", 0)
set_optimizer_attribute(model, "OutputFlag", 1)
```

Note

The string names of the attributes are specific to each solver. One should consult the solver's documentation to find the attributes of interest.

See also: set_optimizer_attribute, set_optimizer_attributes, get_optimizer_attribute.

source

JuMP.get_optimizer_attribute – Function.

```
get_optimizer_attribute(model, name::String)
```

Return the value associated with the solver-specific attribute named `name`.

Note that this is equivalent to `get_optimizer_attribute(model, MOI.RawOptimizerAttribute(name))`.

Example

```
get_optimizer_attribute(model, "SolverSpecificAttributeName")
```

See also: set_optimizer_attribute, set_optimizer_attributes.

source

```
get_optimizer_attribute(
    model::Model, attr::MOI.AbstractOptimizerAttribute
)
```

Return the value of the solver-specific attribute `attr` in `model`.

Example

```
get_optimizer_attribute(model, MOI.Silent())
```

See also: set_optimizer_attribute, set_optimizer_attributes.

source

JuMP.set_optimizer_attribute – Function.

```
set_optimizer_attribute(model::Model, name::String, value)
```

Sets solver-specific attribute identified by `name` to `value`.

Note that this is equivalent to `set_optimizer_attribute(model, MOI.RawOptimizerAttribute(name), value)`.

Example

```
set_optimizer_attribute(model, "SolverSpecificAttributeName", true)
```

See also: set_optimizer_attributes, get_optimizer_attribute.

source

```
set_optimizer_attribute(
    model::Model,
    attr::MOI.AbstractOptimizerAttribute,
    value,
)
```

Set the solver-specific attribute `attr` in `model` to `value`.

Example

```
set_optimizer_attribute(model, MOI.Silent(), true)
```

See also: set_optimizer_attributes, get_optimizer_attribute.

source

JuMP.set_optimizer_attributes – Function.

```
set_optimizer_attributes(model::Model, pairs::Pair...)
```

Given a list of attribute => value pairs, calls set_optimizer_attribute(model, attribute, value) for each pair.

Example

```
model = Model(Ipopt.Optimizer)
set_optimizer_attributes(model, "tol" => 1e-4, "max_iter" => 100)
```

is equivalent to:

```
model = Model(Ipopt.Optimizer)
set_optimizer_attribute(model, "tol", 1e-4)
set_optimizer_attribute(model, "max_iter", 100)
```

See also: set_optimizer_attribute, get_optimizer_attribute.

source

JuMP.set_silent – Function.

```
set_silent(model::Model)
```

Takes precedence over any other attribute controlling verbosity and requires the solver to produce no output.

See also: unset_silent.

source

JuMP.unset_silent – Function.

```
unset_silent(model::Model)
```

Neutralize the effect of the set_silent function and let the solver attributes control the verbosity.

See also: set_silent.

source

JuMP.set_time_limit_sec – Function.

```
set_time_limit_sec(model::Model, limit)
```

Set the time limit (in seconds) of the solver.

Can be unset using unset_time_limit_sec or with limit set to nothing.

See also: unset_time_limit_sec, time_limit_sec.

source

JuMP.unset_time_limit_sec – Function.

```
unset_time_limit_sec(model::Model)
```

Unset the time limit of the solver.

See also: set_time_limit_sec, time_limit_sec.

source

JuMP.time_limit_sec – Function.

```
time_limit_sec(model::Model)
```

Return the time limit (in seconds) of the model.

Returns nothing if unset.

See also: set_time_limit_sec, unset_time_limit_sec.

source

18.5 Copying

JuMP.ReferenceMap – Type.

```
ReferenceMap
```

Mapping between variable and constraint reference of a model and its copy. The reference of the copied model can be obtained by indexing the map with the reference of the corresponding reference of the original model.

source

JuMP.copy_model – Function.

```
copy_model(model::Model; filter_constraints::Union{Nothing, Function}=nothing)
```

Return a copy of the model model and a ReferenceMap that can be used to obtain the variable and constraint reference of the new model corresponding to a given model's reference. A Base.copy(::AbstractModel) method has also been implemented, it is similar to copy_model but does not return the reference map.

If the filter_constraints argument is given, only the constraints for which this function returns true will be copied. This function is given a constraint reference as argument.

Note

Model copy is not supported in DIRECT mode, i.e. when a model is constructed using the direct_model constructor instead of the Model constructor. Moreover, independently on whether an optimizer was provided at model construction, the new model will have no optimizer, i.e., an optimizer will have to be provided to the new model in the optimize! call.

Examples

In the following example, a model model is constructed with a variable x and a constraint cref. It is then copied into a model new_model with the new references assigned to x_new and cref_new.

```
model = Model()
@variable(model, x)
@constraint(model, cref, x == 2)

new_model, reference_map = copy_model(model)
x_new = reference_map[x]
cref_new = reference_map[cref]
```

source

JuMP.copy_extension_data – Function.

```
copy_extension_data(data, new_model::AbstractModel, model::AbstractModel)
```

Return a copy of the extension data `data` of the model `model` to the extension data of the new model `new_model`.

A method should be added for any JuMP extension storing data in the `ext` field.

> **Warning**
> Do not engage in type piracy by implementing this method for types of `data` that you did not define! JuMP extensions should store types that they define in `model.ext`, rather than regular Julia types.

source

Base.copy – Method.

```
copy(model::AbstractModel)
```

Return a copy of the model `model`. It is similar to `copy_model` except that it does not return the mapping between the references of `model` and its copy.

Note

Model copy is not supported in `DIRECT` mode, i.e. when a model is constructed using the `direct_model` constructor instead of the `Model` constructor. Moreover, independently on whether an optimizer was provided at model construction, the new model will have no optimizer, i.e., an optimizer will have to be provided to the new model in the `optimize!` call.

Examples

In the following example, a model `model` is constructed with a variable `x` and a constraint `cref`. It is then copied into a model `new_model` with the new references assigned to `x_new` and `cref_new`.

```
model = Model()
@variable(model, x)
@constraint(model, cref, x == 2)

new_model = copy(model)
x_new = model[:x]
cref_new = model[:cref]
```

source

18.6 I/O

JuMP.write_to_file – Function.

```
write_to_file(
    model::Model,
    filename::String;
    format::MOI.FileFormats.FileFormat = MOI.FileFormats.FORMAT_AUTOMATIC
)
```

Write the JuMP model `model` to `filename` in the format `format`.

If the filename ends in `.gz`, it will be compressed using Gzip. If the filename ends in `.bz2`, it will be compressed using BZip2.

source

`Base.write` – Method.

```
Base.write(
    io::IO,
    model::Model;
    format::MOI.FileFormats.FileFormat = MOI.FileFormats.FORMAT_MOF
)
```

Write the JuMP model `model` to `io` in the format `format`.

source

`JuMP.read_from_file` – Function.

```
read_from_file(
    filename::String;
    format::MOI.FileFormats.FileFormat = MOI.FileFormats.FORMAT_AUTOMATIC
)
```

Return a JuMP model read from `filename` in the format `format`.

If the filename ends in `.gz`, it will be uncompressed using Gzip. If the filename ends in `.bz2`, it will be uncompressed using BZip2.

source

`Base.read` – Method.

```
Base.read(io::IO, ::Type{Model}; format::MOI.FileFormats.FileFormat)
```

Return a JuMP model read from `io` in the format `format`.

source

18.7 Caching Optimizer

`MathOptInterface.Utilities.reset_optimizer` – Method.

```
MOIU.reset_optimizer(model::Model)
```

Call `MOIU.reset_optimizer` on the backend of `model`.

Cannot be called in direct mode.

source

`MathOptInterface.Utilities.drop_optimizer` – Method.

```
MOIU.drop_optimizer(model::Model)
```

Call `MOIU.drop_optimizer` on the backend of `model`.

Cannot be called in direct mode.

source

MathOptInterface.Utilities.attach_optimizer – Method.

```
MOIU.attach_optimizer(model::Model)
```

Call MOIU.attach_optimizer on the backend of model.

Cannot be called in direct mode.

source

18.8 Bridge tools

JuMP.bridge_constraints – Function.

```
bridge_constraints(model::Model)
```

When in direct mode, return false. When in manual or automatic mode, return a Bool indicating whether the optimizer is set and unsupported constraints are automatically bridged to equivalent supported constraints when an appropriate transformation is available.

source

JuMP.print_bridge_graph – Function.

```
print_bridge_graph([io::IO,] model::Model)
```

Print the hyper-graph containing all variable, constraint, and objective types that could be obtained by bridging the variables, constraints, and objectives that are present in the model.

Each node in the hyper-graph corresponds to a variable, constraint, or objective type.

- Variable nodes are indicated by [ ]
- Constraint nodes are indicated by ( )
- Objective nodes are indicated by | |

The number inside each pair of brackets is an index of the node in the hyper-graph.

Note that this hyper-graph is the full list of possible transformations. When the bridged model is created, we select the shortest hyper-path(s) from this graph, so many nodes may be un-used.

For more information, see Legat, B., Dowson, O., Garcia, J., and Lubin, M. (2020). "MathOptInterface: a data structure for mathematical optimization problems." URL: https://arxiv.org/abs/2002.03447

source

18.9 Extension tools

JuMP.AbstractModel – Type.

```
AbstractModel
```

An abstract type that should be subtyped for users creating JuMP extensions.

source

JuMP.operator_warn – Function.

```
operator_warn(model::AbstractModel)
operator_warn(model::Model)
```

This function is called on the model whenever two affine expressions are added together without using `destructive_add!`, and at least one of the two expressions has more than 50 terms.

For the case of `Model`, if this function is called more than 20,000 times then a warning is generated once.

source

`JuMP.error_if_direct_mode` – Function.

```
error_if_direct_mode(model::Model, func::Symbol)
```

Errors if `model` is in direct mode during a call from the function named `func`.

Used internally within JuMP, or by JuMP extensions who do not want to support models in direct mode.

source

Chapter 19

Variables

More information can be found in the Variables section of the manual.

19.1 Macros

JuMP.@variable – Macro.

```
@variable(model, kw_args...)
```

Add an anonymous variable to the model `model` described by the keyword arguments `kw_args` and returns the variable.

```
@variable(model, expr, args..., kw_args...)
```

Add a variable to the model `model` described by the expression `expr`, the positional arguments `args` and the keyword arguments `kw_args`. The expression `expr` can either be (note that in the following the symbol <= can be used instead of ≤, the symbol >=can be used instead of ≥, the symbol `in` can be used instead of ∈)

- of the form `varexpr` creating variables described by `varexpr`;
- of the form `varexpr ≤ ub` (resp. `varexpr ≥ lb`) creating variables described by `varexpr` with upper bounds given by `ub` (resp. lower bounds given by `lb`);
- of the form `varexpr == value` creating variables described by `varexpr` with fixed values given by `value`; or
- of the form `lb ≤ varexpr ≤ ub` or `ub ≥ varexpr ≥ lb` creating variables described by `varexpr` with lower bounds given by `lb` and upper bounds given by `ub`.
- of the form `varexpr ∈ set` creating variables described by `varexpr` constrained to belong to `set`, see Variables constrained on creation.

The expression `varexpr` can either be

- of the form `varname` creating a scalar real variable of name `varname`;
- of the form `varname[...]` or `[...]` creating a container of variables.

The recognized positional arguments in `args` are the following:

- `Bin`: Sets the variable to be binary, i.e. either 0 or 1.

- `Int`: Sets the variable to be integer, i.e. one of ..., -2, -1, 0, 1, 2, ...
- `Symmetric`: Only available when creating a square matrix of variables, i.e. when `varexpr` is of the form `varname[1:n,1:n]` or `varname[i=1:n,j=1:n]`. It creates a symmetric matrix of variable, that is, it only creates a new variable for `varname[i,j]` with `i ≤ j` and sets `varname[j,i]` to the same variable as `varname[i,j]`. It is equivalent to using `varexpr in SymMatrixSpace()` as `expr`.
- `PSD`: The square matrix of variable is both `Symmetric` and constrained to be positive semidefinite. It is equivalent to using `varexpr in PSDCone()` as `expr`.

The recognized keyword arguments in `kw_args` are the following:

- `base_name`: Sets the name prefix used to generate variable names. It corresponds to the variable name for scalar variable, otherwise, the variable names are set to `base_name[...]` for each index `...` of the axes `axes`.
- `lower_bound`: Sets the value of the variable lower bound.
- `upper_bound`: Sets the value of the variable upper bound.
- `start`: Sets the variable starting value used as initial guess in optimization.
- `binary`: Sets whether the variable is binary or not.
- `integer`: Sets whether the variable is integer or not.
- `variable_type`: See the "Note for extending the variable macro" section below.
- `set`: Equivalent to using `varexpr in value` as `expr` where `value` is the value of the keyword argument.
- `container`: Specify the container type.

Examples

The following are equivalent ways of creating a variable `x` of name `x` with lower bound 0:

```
# Specify everything in `expr`
@variable(model, x >= 0)
# Specify the lower bound using a keyword argument
@variable(model, x, lower_bound=0)
# Specify everything in `kw_args`
x = @variable(model, base_name="x", lower_bound=0)
```

The following are equivalent ways of creating a `DenseAxisArray` of index set `[:a, :b]` and with respective upper bounds 2 and 3 and names `x[a]` and `x[b]`. The upper bound can either be specified in `expr`:

```
ub = Dict(:a => 2, :b => 3)
@variable(model, x[i=keys(ub)] <= ub[i])

# output
1-dimensional DenseAxisArray{VariableRef,1,...} with index sets:
    Dimension 1, Symbol[:a, :b]
And data, a 2-element Array{VariableRef,1}:
 x[a]
 x[b]
```

or it can be specified with the `upper_bound` keyword argument:

```
@variable(model, y[i=keys(ub)], upper_bound=ub[i])

# output
1-dimensional DenseAxisArray{VariableRef,1,...} with index sets:
    Dimension 1, Symbol[:a, :b]
And data, a 2-element Array{VariableRef,1}:
 y[a]
 y[b]
```

source

JuMP.@variables – Macro.

```
@variables(model, args...)
```

Adds multiple variables to model at once, in the same fashion as the @variable macro.

The model must be the first argument, and multiple variables can be added on multiple lines wrapped in a `begin ... end` block.

Examples

```
@variables(model, begin
    x
    y[i = 1:2] >= 0, (start = i)
    z, Bin, (start = 0, base_name = "Z")
end)
```

> **Note**
> Keyword arguments must be contained within parentheses (refer to the example above).

source

19.2 Basic utilities

JuMP.VariableRef – Type.

```
VariableRef <: AbstractVariableRef
```

Holds a reference to the model and the corresponding MOI.VariableIndex.

source

JuMP.num_variables – Function.

```
num_variables(model::Model)::Int64
```

Returns number of variables in `model`.

source

JuMP.all_variables – Function.

```
all_variables(model::Model)::Vector{VariableRef}
```

Returns a list of all variables currently in the model. The variables are ordered by creation time.

Example

```
model = Model()
@variable(model, x)
@variable(model, y)
all_variables(model)

# output

2-element Array{VariableRef,1}:
 x
 y
```

source

JuMP.owner_model – Function.

```
owner_model(s::AbstractJuMPScalar)
```

Return the model owning the scalar s.

source

JuMP.index – Method.

```
index(v::VariableRef)::MOI.VariableIndex
```

Return the index of the variable that corresponds to v in the MOI backend.

source

JuMP.optimizer_index – Method.

```
optimizer_index(v::VariableRef)::MOI.VariableIndex
```

Return the index of the variable that corresponds to v in the optimizer model. It throws NoOptimizer if no optimizer is set and throws an ErrorException if the optimizer is set but is not attached.

source

JuMP.check_belongs_to_model – Function.

```
check_belongs_to_model(func::AbstractJuMPScalar, model::AbstractModel)
```

Throw VariableNotOwned if the owner_model of one of the variables of the function func is not model.

```
check_belongs_to_model(constraint::AbstractConstraint, model::AbstractModel)
```

Throw VariableNotOwned if the owner_model of one of the variables of the constraint constraint is not model.

source

JuMP.VariableNotOwned – Type.

```
struct VariableNotOwned{V <: AbstractVariableRef} <: Exception
    variable::V
end
```

The variable variable was used in a model different to owner_model(variable).

source

JuMP.VariableConstrainedOnCreation – Type.

```
VariablesConstrainedOnCreation <: AbstractVariable
```

Variable `scalar_variables` constrained to belong to `set`. Adding this variable can be understood as doing:

```
function JuMP.add_variable(model::Model, variable::JuMP.VariableConstrainedOnCreation, names)
    var_ref = JuMP.add_variable(model, variable.scalar_variable, name)
    JuMP.add_constraint(model, JuMP.VectorConstraint(var_ref, variable.set))
    return var_ref
end
```

but adds the variables with `MOI.add_constrained_variable(model, variable.set)` instead. See the MOI documentation for the difference between adding the variables with `MOI.add_constrained_variable` and adding them with `MOI.add_variable` and adding the constraint separately.

source

JuMP.VariablesConstrainedOnCreation – Type.

```
VariablesConstrainedOnCreation <: AbstractVariable
```

Vector of variables `scalar_variables` constrained to belong to `set`. Adding this variable can be thought as doing:

```
function JuMP.add_variable(model::Model, variable::JuMP.VariablesConstrainedOnCreation, names)
    var_refs = JuMP.add_variable.(model, variable.scalar_variables,
                                  JuMP.vectorize(names, variable.shape))
    JuMP.add_constraint(model, JuMP.VectorConstraint(var_refs, variable.set))
    return JuMP.reshape_vector(var_refs, variable.shape)
end
```

but adds the variables with `MOI.add_constrained_variables(model, variable.set)` instead. See the MOI documentation for the difference between adding the variables with `MOI.add_constrained_variables` and adding them with `MOI.add_variables` and adding the constraint separately.

source

19.3 Names

JuMP.name – Method.

```
name(v::VariableRef)::String
```

Get a variable's name attribute.

source

JuMP.set_name – Method.

```
set_name(v::VariableRef, s::AbstractString)
```

Set a variable's name attribute.

source

JuMP.variable_by_name – Function.

```
variable_by_name(model::AbstractModel,
                 name::String)::Union{AbstractVariableRef, Nothing}
```

Returns the reference of the variable with name attribute `name` or `Nothing` if no variable has this name attribute. Throws an error if several variables have `name` as their name attribute.

```
julia> model = Model();

julia> @variable(model, x)
x

julia> variable_by_name(model, "x")
x

julia> @variable(model, base_name="x")
x

julia> variable_by_name(model, "x")
ERROR: Multiple variables have the name x.
Stacktrace:
 [1] error(::String) at ./error.jl:33
 [2] get(::MOIU.Model{Float64}, ::Type{MathOptInterface.VariableIndex}, ::String) at
↪  /home/blegat/.julia/dev/MathOptInterface/src/Utilities/model.jl:222
 [3] get at /home/blegat/.julia/dev/MathOptInterface/src/Utilities/universalfallback.jl:201
↪  [inlined]
 [4]
↪  get(::MathOptInterface.Utilities.CachingOptimizer{MathOptInterface.AbstractOptimizer,MathOptInterface.Utilitie
↪  ::Type{MathOptInterface.VariableIndex}, ::String) at
↪  /home/blegat/.julia/dev/MathOptInterface/src/Utilities/cachingoptimizer.jl:490
 [5] variable_by_name(::Model, ::String) at /home/blegat/.julia/dev/JuMP/src/variables.jl:268
 [6] top-level scope at none:0

julia> var = @variable(model, base_name="y")
y

julia> variable_by_name(model, "y")
y

julia> set_name(var, "z")

julia> variable_by_name(model, "y")

julia> variable_by_name(model, "z")
z

julia> @variable(model, u[1:2])
2-element Array{VariableRef,1}:
 u[1]
 u[2]

julia> variable_by_name(model, "u[2]")
u[2]
```

source

19.4 Start values

JuMP.set_start_value – Function.

```
set_start_value(variable::VariableRef, value::Union{Real,Nothing})
```

Set the start value (MOI attribute VariablePrimalStart) of the variable to value.

Pass nothing to unset the start value.

Note: VariablePrimalStarts are sometimes called "MIP-starts" or "warmstarts".

See also start_value.

source

JuMP.start_value – Function.

```
start_value(v::VariableRef)
```

Return the start value (MOI attribute VariablePrimalStart) of the variable v.

Note: VariablePrimalStarts are sometimes called "MIP-starts" or "warmstarts".

See also set_start_value.

source

19.5 Lower bounds

JuMP.has_lower_bound – Function.

```
has_lower_bound(v::VariableRef)
```

Return true if v has a lower bound. If true, the lower bound can be queried with lower_bound.

See also LowerBoundRef, lower_bound, set_lower_bound, delete_lower_bound.

source

JuMP.lower_bound – Function.

```
lower_bound(v::VariableRef)
```

Return the lower bound of a variable. Error if one does not exist.

See also LowerBoundRef, has_lower_bound, set_lower_bound, delete_lower_bound.

source

JuMP.set_lower_bound – Function.

```
set_lower_bound(v::VariableRef, lower::Number)
```

Set the lower bound of a variable. If one does not exist, create a new lower bound constraint.

See also LowerBoundRef, has_lower_bound, lower_bound, delete_lower_bound.

source

JuMP.delete_lower_bound – Function.

```
delete_lower_bound(v::VariableRef)
```

Delete the lower bound constraint of a variable.

See also LowerBoundRef, has_lower_bound, lower_bound, set_lower_bound.

source

JuMP.LowerBoundRef – Function.

```
LowerBoundRef(v::VariableRef)
```

Return a constraint reference to the lower bound constraint of v. Errors if one does not exist.

See also has_lower_bound, lower_bound, set_lower_bound, delete_lower_bound.

source

19.6 Upper bounds

JuMP.has_upper_bound – Function.

```
has_upper_bound(v::VariableRef)
```

Return true if v has a upper bound. If true, the upper bound can be queried with upper_bound.

See also UpperBoundRef, upper_bound, set_upper_bound, delete_upper_bound.

source

JuMP.upper_bound – Function.

```
upper_bound(v::VariableRef)
```

Return the upper bound of a variable. Error if one does not exist.

See also UpperBoundRef, has_upper_bound, set_upper_bound, delete_upper_bound.

source

JuMP.set_upper_bound – Function.

```
set_upper_bound(v::VariableRef,upper::Number)
```

Set the upper bound of a variable. If one does not exist, create an upper bound constraint.

See also UpperBoundRef, has_upper_bound, upper_bound, delete_upper_bound.

source

JuMP.delete_upper_bound – Function.

```
delete_upper_bound(v::VariableRef)
```

Delete the upper bound constraint of a variable.

See also UpperBoundRef, has_upper_bound, upper_bound, set_upper_bound.

source

JuMP.UpperBoundRef – Function.

```
UpperBoundRef(v::VariableRef)
```

Return a constraint reference to the upper bound constraint of `v`. Errors if one does not exist.

See also `has_upper_bound`, `upper_bound`, `set_upper_bound`, `delete_upper_bound`.

source

19.7 Fixed bounds

`JuMP.is_fixed` – Function.

```
is_fixed(v::VariableRef)
```

Return `true` if `v` is a fixed variable. If `true`, the fixed value can be queried with `fix_value`.

See also `FixRef`, `fix_value`, `fix`, `unfix`.

source

`JuMP.fix_value` – Function.

```
fix_value(v::VariableRef)
```

Return the value to which a variable is fixed. Error if one does not exist.

See also `FixRef`, `is_fixed`, `fix`, `unfix`.

source

`JuMP.fix` – Function.

```
fix(v::VariableRef, value::Number; force::Bool = false)
```

Fix a variable to a value. Update the fixing constraint if one exists, otherwise create a new one.

If the variable already has variable bounds and `force=false`, calling `fix` will throw an error. If `force=true`, existing variable bounds will be deleted, and the fixing constraint will be added. Note a variable will have no bounds after a call to `unfix`.

See also `FixRef`, `is_fixed`, `fix_value`, `unfix`.

source

`JuMP.unfix` – Function.

```
unfix(v::VariableRef)
```

Delete the fixing constraint of a variable.

See also `FixRef`, `is_fixed`, `fix_value`, `fix`.

source

`JuMP.FixRef` – Function.

```
FixRef(v::VariableRef)
```

Return a constraint reference to the constraint fixing the value of `v`. Errors if one does not exist.

See also `is_fixed`, `fix_value`, `fix`, `unfix`.

source

19.8 Integer variables

JuMP.is_integer – Function.

```
is_integer(v::VariableRef)
```

Return `true` if v is constrained to be integer.

See also IntegerRef, set_integer, unset_integer.

source

JuMP.set_integer – Function.

```
set_integer(variable_ref::VariableRef)
```

Add an integrality constraint on the variable `variable_ref`.

See also IntegerRef, is_integer, unset_integer.

source

JuMP.unset_integer – Function.

```
unset_integer(variable_ref::VariableRef)
```

Remove the integrality constraint on the variable `variable_ref`.

See also IntegerRef, is_integer, set_integer.

source

JuMP.IntegerRef – Function.

```
IntegerRef(v::VariableRef)
```

Return a constraint reference to the constraint constraining v to be integer. Errors if one does not exist.

See also is_integer, set_integer, unset_integer.

source

19.9 Binary variables

JuMP.is_binary – Function.

```
is_binary(v::VariableRef)
```

Return `true` if v is constrained to be binary.

See also BinaryRef, set_binary, unset_binary.

source

JuMP.set_binary – Function.

```
set_binary(v::VariableRef)
```

Add a constraint on the variable v that it must take values in the set $\{0, 1\}$.

See also BinaryRef, is_binary, unset_binary.

source

JuMP.unset_binary – Function.

```
unset_binary(variable_ref::VariableRef)
```

Remove the binary constraint on the variable `variable_ref`.

See also BinaryRef, is_binary, set_binary.

source

JuMP.BinaryRef – Function.

```
BinaryRef(v::VariableRef)
```

Return a constraint reference to the constraint constraining v to be binary. Errors if one does not exist.

See also is_binary, set_binary, unset_binary.

source

19.10 Integrality utilities

JuMP.relax_integrality – Function.

```
relax_integrality(model::Model)
```

Modifies `model` to "relax" all binary and integrality constraints on variables. Specifically,

- Binary constraints are deleted, and variable bounds are tightened if necessary to ensure the variable is constrained to the interval $[0, 1]$.
- Integrality constraints are deleted without modifying variable bounds.
- An error is thrown if semi-continuous or semi-integer constraints are present (support may be added for these in the future).
- All other constraints are ignored (left in place). This includes discrete constraints like SOS and indicator constraints.

Returns a function that can be called without any arguments to restore the original model. The behavior of this function is undefined if additional changes are made to the affected variables in the meantime.

Example

```
julia> model = Model();

julia> @variable(model, x, Bin);

julia> @variable(model, 1 <= y <= 10, Int);

julia> @objective(model, Min, x + y);

julia> undo_relax = relax_integrality(model);

julia> print(model)
Min x + y
Subject to
 x ≥ 0.0
 y ≥ 1.0
```

```
 x ≤ 1.0
 y ≤ 10.0

julia> undo_relax()

julia> print(model)
Min x + y
Subject to
 y ≥ 1.0
 y ≤ 10.0
 y integer
 x binary
```

source

19.11 Extensions

JuMP.AbstractVariable – Type.

```
AbstractVariable
```

Variable returned by build_variable. It represents a variable that has not been added yet to any model. It can be added to a given model with add_variable.

source

JuMP.AbstractVariableRef – Type.

```
AbstractVariableRef
```

Variable returned by add_variable. Affine (resp. quadratic) operations with variables of type V<:AbstractVariableRef and coefficients of type T create a GenericAffExpr{T,V} (resp. GenericQuadExpr{T,V}).

source

JuMP.parse_one_operator_variable – Function.

```
parse_one_operator_variable(_error::Function, infoexpr::_VariableInfoExpr, sense::Val{S}, value)
↪  where S
```

Update infoexr for a variable expression in the @variable macro of the form variable name S value.

source

Chapter 20

Expressions

More information can be found in the Expressions section of the manual.

20.1 Macros

JuMP.@expression – Macro.

```
@expression(args...)
```

Efficiently builds a linear or quadratic expression but does not add to model immediately. Instead, returns the expression which can then be inserted in other constraints. For example:

```
@expression(m, shared, sum(i*x[i] for i=1:5))
@constraint(m, shared + y >= 5)
@constraint(m, shared + z <= 10)
```

The `ref` accepts index sets in the same way as `@variable`, and those indices can be used in the construction of the expressions:

```
@expression(m, expr[i=1:3], i*sum(x[j] for j=1:3))
```

Anonymous syntax is also supported:

```
expr = @expression(m, [i=1:3], i*sum(x[j] for j=1:3))
```

source

JuMP.@expressions – Macro.

```
@expressions(model, args...)
```

Adds multiple expressions to model at once, in the same fashion as the `@expression` macro.

The model must be the first argument, and multiple variables can be added on multiple lines wrapped in a `begin ... end` block.

Examples

```
@expressions(model, begin
    my_expr, x^2 + y^2
    my_expr_1[i = 1:2], a[i] - z[i]
end)
```

source

20.2 Affine expressions

JuMP.GenericAffExpr – Type.

```
mutable struct GenericAffExpr{CoefType,VarType} <: AbstractJuMPScalar
    constant::CoefType
    terms::OrderedDict{VarType,CoefType}
end
```

An expression type representing an affine expression of the form: $\sum a_i x_i + c$.

Fields

- `.constant`: the constant c in the expression.
- `.terms`: an `OrderedDict`, with keys of `VarType` and values of `CoefType` describing the sparse vector a.

source

JuMP.AffExpr – Type.

```
AffExpr
```

Alias for `GenericAffExpr{Float64,VariableRef}`, the specific GenericAffExpr used by JuMP.

source

JuMP.linear_terms – Function.

```
linear_terms(aff::GenericAffExpr{C, V})
```

Provides an iterator over coefficient-variable tuples `(a_i::C, x_i::V)` in the linear part of the affine expression.

source

```
linear_terms(quad::GenericQuadExpr{C, V})
```

Provides an iterator over tuples `(coefficient::C, variable::V)` in the linear part of the quadratic expression.

source

20.3 Quadratic expressions

JuMP.GenericQuadExpr – Type.

```
mutable struct GenericQuadExpr{CoefType,VarType} <: AbstractJuMPScalar
    aff::GenericAffExpr{CoefType,VarType}
    terms::OrderedDict{UnorderedPair{VarType}, CoefType}
end
```

An expression type representing an quadratic expression of the form: $\sum q_{i,j} x_i x_j + \sum a_i x_i + c$.

Fields

- `.aff`: an GenericAffExpr representing the affine portion of the expression.

- `.terms`: an `OrderedDict`, with keys of `UnorderedPair{VarType}` and values of `CoefType`, describing the sparse list of terms `q`.

source

`JuMP.QuadExpr` – Type.

```
QuadExpr
```

An alias for `GenericQuadExpr{Float64,VariableRef}`, the specific `GenericQuadExpr` used by JuMP.

source

`JuMP.UnorderedPair` – Type.

```
UnorderedPair(a::T, b::T)
```

A wrapper type used by `GenericQuadExpr` with fields `.a` and `.b`.

source

`JuMP.quad_terms` – Function.

```
quad_terms(quad::GenericQuadExpr{C, V})
```

Provides an iterator over tuples `(coefficient::C, var_1::V, var_2::V)` in the quadratic part of the quadratic expression.

source

20.4 Utilities and modifications

`JuMP.constant` – Function.

```
constant(aff::GenericAffExpr{C, V})::C
```

Return the constant of the affine expression.

source

```
constant(aff::GenericQuadExpr{C, V})::C
```

Return the constant of the quadratic expression.

source

`JuMP.coefficient` – Function.

```
coefficient(v1::VariableRef, v2::VariableRef)
```

Return `1.0` if `v1 == v2`, and `0.0` otherwise.

This is a fallback for other `coefficient` methods to simplify code in which the expression may be a single variable.

source

```
coefficient(a::GenericAffExpr{C,V}, v::V) where {C,V}
```

Return the coefficient associated with variable v in the affine expression a.

source

```
coefficient(a::GenericAffExpr{C,V}, v1::V, v2::V) where {C,V}
```

Return the coefficient associated with the term `v1 * v2` in the quadratic expression a.

Note that `coefficient(a, v1, v2)` is the same as `coefficient(a, v2, v1)`.

source

```
coefficient(a::GenericQuadExpr{C,V}, v::V) where {C,V}
```

Return the coefficient associated with variable v in the affine component of a.

source

JuMP.isequal_canonical – Function.

```
isequal_canonical(
    aff::GenericAffExpr{C,V},
    other::GenericAffExpr{C,V}
) where {C,V}
```

Return `true` if `aff` is equal to `other` after dropping zeros and disregarding the order. Mainly useful for testing.

source

JuMP.add_to_expression! – Function.

```
add_to_expression!(expression, terms...)
```

Updates `expression` in place to `expression + (*)(terms...)`. This is typically much more efficient than `expression += (*)(terms...)`. For example, `add_to_expression!(expression, a, b)` produces the same result as `expression += a*b`, and `add_to_expression!(expression, a)` produces the same result as `expression += a`.

Only a few methods are defined, mostly for internal use, and only for the cases when (1) they can be implemented efficiently and (2) `expression` is capable of storing the result. For example, `add_to_expression!(::AffExpr, ::VariableRef, ::VariableRef)` is not defined because a `GenericAffExpr` cannot store the product of two variables.

source

JuMP.drop_zeros! – Function.

```
drop_zeros!(expr::GenericAffExpr)
```

Remove terms in the affine expression with `0` coefficients.

source

```
drop_zeros!(expr::GenericQuadExpr)
```

Remove terms in the quadratic expression with `0` coefficients.

source

JuMP.map_coefficients – Function.

```
map_coefficients(f::Function, a::GenericAffExpr)
```

Apply `f` to the coefficients and constant term of an `GenericAffExpr` `a` and return a new expression.

See also: `map_coefficients_inplace!`

Example

```
julia> a = GenericAffExpr(1.0, x => 1.0)
x + 1

julia> map_coefficients(c -> 2 * c, a)
2 x + 2

julia> a
x + 1
```

source

```
map_coefficients(f::Function, a::GenericQuadExpr)
```

Apply `f` to the coefficients and constant term of an `GenericQuadExpr` `a` and return a new expression.

See also: `map_coefficients_inplace!`

Example

```
julia> a = @expression(model, x^2 + x + 1)
x² + x + 1

julia> map_coefficients(c -> 2 * c, a)
2 x² + 2 x + 2

julia> a
x² + x + 1
```

source

`JuMP.map_coefficients_inplace!` – Function.

```
map_coefficients_inplace!(f::Function, a::GenericAffExpr)
```

Apply `f` to the coefficients and constant term of an `GenericAffExpr` `a` and update them in-place.

See also: `map_coefficients`

Example

```
julia> a = GenericAffExpr(1.0, x => 1.0)
x + 1

julia> map_coefficients_inplace!(c -> 2 * c, a)
2 x + 2

julia> a
2 x + 2
```

source

```
map_coefficients_inplace!(f::Function, a::GenericQuadExpr)
```

Apply f to the coefficients and constant term of an GenericQuadExpr a and update them in-place.

See also: map_coefficients

Example

```
julia> a = @expression(model, x^2 + x + 1)
x² + x + 1

julia> map_coefficients_inplace!(c -> 2 * c, a)
2 x² + 2 x + 2

julia> a
2 x² + 2 x + 2
```

source

20.5 JuMP-to-MOI converters

JuMP.variable_ref_type – Function.

```
variable_ref_type(::GenericAffExpr{C, V}) where {C, V}
```

A helper function used internally by JuMP and some JuMP extensions. Returns the variable type V from a GenericAffExpr

source

JuMP.jump_function – Function.

```
jump_function(x)
```

Given an MathOptInterface object x, return the JuMP equivalent.

See also: moi_function.

source

JuMP.jump_function_type – Function.

```
jump_function_type(::Type{T}) where {T}
```

Given an MathOptInterface object type T, return the JuMP equivalent.

See also: moi_function_type.

source

JuMP.moi_function – Function.

```
moi_function(x)
```

Given a JuMP object x, return the MathOptInterface equivalent.

See also: jump_function.

source

JuMP.moi_function_type – Function.

```
moi_function_type(::Type{T}) where {T}
```

Given a JuMP object type T, return the MathOptInterface equivalent.

See also: jump_function_type.

source

Chapter 21

Objectives

More information can be found in the Objectives section of the manual.

21.1 Objective functions

`JuMP.@objective` – Macro.

```
@objective(model::Model, sense, func)
```

Set the objective sense to `sense` and objective function to `func`. The objective sense can be either `Min`, `Max`, `MathOptInterface.MIN_SENSE`, `MathOptInterface.MAX_SENSE` or `MathOptInterface.FEASIBILITY_SENSE`; see `MathOptInterface.ObjectiveSense`. In order to set the sense programmatically, i.e., when `sense` is a Julia variable whose value is the sense, one of the three `MathOptInterface.ObjectiveSense` values should be used. The function `func` can be a single JuMP variable, an affine expression of JuMP variables or a quadratic expression of JuMP variables.

Examples

To minimize the value of the variable `x`, do as follows:

```
julia> model = Model()
A JuMP Model
Feasibility problem with:
Variables: 0
Model mode: AUTOMATIC
CachingOptimizer state: NO_OPTIMIZER
Solver name: No optimizer attached.

julia> @variable(model, x)
x

julia> @objective(model, Min, x)
x
```

To maximize the value of the affine expression `2x - 1`, do as follows:

```
julia> @objective(model, Max, 2x - 1)
2 x - 1
```

To set a quadratic objective and set the objective sense programmatically, do as follows:

```
julia> sense = MOI.MIN_SENSE
MIN_SENSE::OptimizationSense = 0

julia> @objective(model, sense, x^2 - 2x + 1)
x² - 2 x + 1
```

source

JuMP.objective_function – Function.

```
objective_function(model::Model,
                   T::Type{<:AbstractJuMPScalar}=objective_function_type(model))
```

Return an object of type T representing the objective function. Error if the objective is not convertible to type T.

Examples

```
julia> model = Model()
A JuMP Model
Feasibility problem with:
Variables: 0
Model mode: AUTOMATIC
CachingOptimizer state: NO_OPTIMIZER
Solver name: No optimizer attached.

julia> @variable(model, x)
x

julia> @objective(model, Min, 2x + 1)
2 x + 1

julia> objective_function(model, AffExpr)
2 x + 1

julia> objective_function(model, QuadExpr)
2 x + 1

julia> typeof(objective_function(model, QuadExpr))
GenericQuadExpr{Float64,VariableRef}
```

We see with the last two commands that even if the objective function is affine, as it is convertible to a quadratic function, it can be queried as a quadratic function and the result is quadratic.

However, it is not convertible to a variable.

```
julia> objective_function(model, VariableRef)
ERROR: InexactError: convert(MathOptInterface.VariableIndex,
↪  MathOptInterface.ScalarAffineFunction{Float64}(MathOptInterface.ScalarAffineTerm{Float64}[MathOptInterface.Sca
↪  MathOptInterface.VariableIndex(1))], 1.0))
[...]
```

source

JuMP.set_objective_function – Function.

```
set_objective_function(
    model::Model,
    func::Union{AbstractJuMPScalar, MathOptInterface.AbstractScalarFunction})
```

Sets the objective function of the model to the given function. See `set_objective_sense` to set the objective sense. These are low-level functions; the recommended way to set the objective is with the `@objective` macro.

source

`JuMP.set_objective_coefficient` – Function.

```
set_objective_coefficient(model::Model, variable::VariableRef, coefficient::Real)
```

Set the linear objective coefficient associated with `Variable` to `coefficient`.

Note: this function will throw an error if a nonlinear objective is set.

source

`JuMP.set_objective` – Function.

```
set_objective(model::AbstractModel, sense::MOI.OptimizationSense, func)
```

The functional equivalent of the `@objective` macro.

Sets the objective sense and objective function simultaneously, and is equivalent to:

```
set_objective_sense(model, sense)
set_objective_function(model, func)
```

Examples

```
model = Model()
@variable(model, x)
set_objective(model, MOI.MIN_SENSE, x)
```

source

`JuMP.objective_function_type` – Function.

```
objective_function_type(model::Model)::AbstractJuMPScalar
```

Return the type of the objective function.

source

`JuMP.objective_function_string` – Function.

```
objective_function_string(print_mode, model::AbstractModel)::String
```

Return a `String` describing the objective function of the model.

source

`JuMP.show_objective_function_summary` – Function.

```
show_objective_function_summary(io::IO, model::AbstractModel)
```

Write to `io` a summary of the objective function type.

source

21.2 Objective sense

JuMP.objective_sense – Function.

```
objective_sense(model::Model)::MathOptInterface.OptimizationSense
```

Return the objective sense.

source

JuMP.set_objective_sense – Function.

```
set_objective_sense(model::Model, sense::MathOptInterface.OptimizationSense)
```

Sets the objective sense of the model to the given sense. See set_objective_function to set the objective function. These are low-level functions; the recommended way to set the objective is with the @objective macro.

source

Chapter 22

Constraints

More information can be found in the Constraints section of the manual.

22.1 Macros

JuMP.@constraint – Macro.

```
@constraint(m::Model, expr, kw_args...)
```

Add a constraint described by the expression `expr`.

```
@constraint(m::Model, ref[i=..., j=..., ...], expr, kw_args...)
```

Add a group of constraints described by the expression `expr` parametrized by `i`, `j`, ...

The expression `expr` can either be

- of the form `func in set` constraining the function `func` to belong to the set `set` which is either a `MOI.AbstractSet` or one of the JuMP shortcuts `SecondOrderCone`, `RotatedSecondOrderCone` and `PSDCone`, e.g. `@constraint(model, [1, x-1, y-2] in SecondOrderCone())` constrains the norm of `[x-1, y-2]` be less than 1;
- of the form `a sign b`, where `sign` is one of ==, ≥, >=, ≤ and <= building the single constraint enforcing the comparison to hold for the expression `a` and `b`, e.g. `@constraint(m, x^2 + y^2 == 1)` constrains `x` and `y` to lie on the unit circle;
- of the form `a ≤ b ≤ c` or `a ≥ b ≥ c` (where ≤ and <= (resp. ≥ and >=) can be used interchangeably) constraining the paired the expression `b` to lie between `a` and `c`;
- of the forms `@constraint(m, a .sign b)` or `@constraint(m, a .sign b .sign c)` which broadcast the constraint creation to each element of the vectors.

The recognized keyword arguments in `kw_args` are the following:

- `base_name`: Sets the name prefix used to generate constraint names. It corresponds to the constraint name for scalar constraints, otherwise, the constraint names are set to `base_name[...]` for each index `...` of the axes `axes`.
- `container`: Specify the container type.

Note for extending the constraint macro

Each constraint will be created using `add_constraint(m, build_constraint(_error, func, set))` where

- `_error` is an error function showing the constraint call in addition to the error message given as argument,
- `func` is the expression that is constrained
- and `set` is the set in which it is constrained to belong.

For `expr` of the first type (i.e. `@constraint(m, func in set)`), `func` and `set` are passed unchanged to `build_constraint` but for the other types, they are determined from the expressions and signs. For instance, `@constraint(m, x^2 + y^2 == 1)` is transformed into `add_constraint(m, build_constraint(_error, x^2 + y^2, MOI.EqualTo(1.0)))`.

To extend JuMP to accept new constraints of this form, it is necessary to add the corresponding methods to `build_constraint`. Note that this will likely mean that either `func` or `set` will be some custom type, rather than e.g. a `Symbol`, since we will likely want to dispatch on the type of the function or set appearing in the constraint.

For extensions that need to create constraints with more information than just `func` and `set`, an additional positional argument can be specified to `@constraint` that will then be passed on `build_constraint`. Hence, we can enable this syntax by defining extensions of `build_constraint(_error, func, set, my_arg; kw_args...)`. This produces the user syntax: `@constraint(model, ref[...], expr, my_arg, kw_args...)`.

source

`JuMP.@constraints` – Macro.

```
@constraints(model, args...)
```

Adds groups of constraints at once, in the same fashion as the `@constraint` macro.

The model must be the first argument, and multiple constraints can be added on multiple lines wrapped in a `begin ... end` block.

Examples

```
@constraints(model, begin
    x >= 1
    y - w <= 2
    sum_to_one[i=1:3], z[i] + y == 1
end)
```

source

`JuMP.ConstraintRef` – Type.

```
ConstraintRef
```

Holds a reference to the model and the corresponding MOI.ConstraintIndex.

source

`JuMP.AbstractConstraint` – Type.

```
abstract type AbstractConstraint
```

An abstract base type for all constraint types. `AbstractConstraints` store the function and set directly, unlike `ConstraintRef`s that are merely references to constraints stored in a model. `AbstractConstraints` do not need to be attached to a model.

source

JuMP.ScalarConstraint – Type.

```
struct ScalarConstraint
```

The data for a scalar constraint. The `func` field contains a JuMP object representing the function and the `set` field contains the MOI set. See also the documentation on JuMP's representation of constraints for more background.

source

JuMP.VectorConstraint – Type.

```
struct VectorConstraint
```

The data for a vector constraint. The `func` field contains a JuMP object representing the function and the `set` field contains the MOI set. The `shape` field contains an `AbstractShape` matching the form in which the constraint was constructed (e.g., by using matrices or flat vectors). See also the documentation on JuMP's representation of constraints.

source

22.2 Names

JuMP.name – Method.

```
name(con_ref::ConstraintRef)
```

Get a constraint's name attribute.

source

JuMP.set_name – Method.

```
set_name(con_ref::ConstraintRef, s::AbstractString)
```

Set a constraint's name attribute.

source

JuMP.constraint_by_name – Function.

```
constraint_by_name(model::AbstractModel,
                   name::String)::Union{ConstraintRef, Nothing}
```

Return the reference of the constraint with name attribute `name` or `Nothing` if no constraint has this name attribute. Throws an error if several constraints have `name` as their name attribute.

```
constraint_by_name(model::AbstractModel,
                   name::String,
                   F::Type{<:Union{AbstractJuMPScalar,
                                   Vector{<:AbstractJuMPScalar},
                                   MOI.AbstactFunction}},
                   S::Type{<:MOI.AbstractSet})::Union{ConstraintRef, Nothing}
```

Similar to the method above, except that it throws an error if the constraint is not an F-in-S contraint where F is either the JuMP or MOI type of the function, and S is the MOI type of the set. This method is recommended if you know the type of the function and set since its returned type can be inferred while for the method above (i.e. without F and S), the exact return type of the constraint index cannot be inferred.

```
julia> using JuMP

julia> model = Model()
A JuMP Model
Feasibility problem with:
Variables: 0
Model mode: AUTOMATIC
CachingOptimizer state: NO_OPTIMIZER
Solver name: No optimizer attached.

julia> @variable(model, x)
x

julia> @constraint(model, con, x^2 == 1)
con : x² = 1.0

julia> constraint_by_name(model, "kon")

julia> constraint_by_name(model, "con")
con : x² = 1.0

julia> constraint_by_name(model, "con", AffExpr, MOI.EqualTo{Float64})

julia> constraint_by_name(model, "con", QuadExpr, MOI.EqualTo{Float64})
con : x² = 1.0
```

source

22.3 Modification

JuMP.normalized_coefficient – Function.

```
normalized_coefficient(con_ref::ConstraintRef, variable::VariableRef)
```

Return the coefficient associated with `variable` in `constraint` after JuMP has normalized the constraint into its standard form. See also set_normalized_coefficient.

source

JuMP.set_normalized_coefficient – Function.

```
set_normalized_coefficient(con_ref::ConstraintRef, variable::VariableRef, value)
```

Set the coefficient of `variable` in the constraint `constraint` to `value`.

Note that prior to this step, JuMP will aggregate multiple terms containing the same variable. For example, given a constraint `2x + 3x <= 2`, `set_normalized_coefficient(con, x, 4)` will create the constraint `4x <= 2`.

```
model = Model()
@variable(model, x)
@constraint(model, con, 2x + 3x <= 2)
set_normalized_coefficient(con, x, 4)
con

# output

con : 4 x <= 2.0
```

source

JuMP.normalized_rhs – Function.

```
normalized_rhs(con_ref::ConstraintRef)
```

Return the right-hand side term of `con_ref` after JuMP has converted the constraint into its normalized form. See also `set_normalized_rhs`.

source

JuMP.set_normalized_rhs – Function.

```
set_normalized_rhs(con_ref::ConstraintRef, value)
```

Set the right-hand side term of `constraint` to `value`.

Note that prior to this step, JuMP will aggregate all constant terms onto the right-hand side of the constraint. For example, given a constraint `2x + 1 <= 2`, `set_normalized_rhs(con, 4)` will create the constraint `2x <= 4`, not `2x + 1 <= 4`.

```
julia> @constraint(model, con, 2x + 1 <= 2)
con : 2 x <= 1.0

julia> set_normalized_rhs(con, 4)

julia> con
con : 2 x <= 4.0
```

source

JuMP.add_to_function_constant – Function.

```
add_to_function_constant(constraint::ConstraintRef, value)
```

Add `value` to the function constant term.

Note that for scalar constraints, JuMP will aggregate all constant terms onto the right-hand side of the constraint so instead of modifying the function, the set will be translated by `-value`. For example, given a constraint `2x <= 3`, `add_to_function_constant(c, 4)` will modify it to `2x <= -1`.

Examples

For scalar constraints, the set is translated by `-value`:

```
julia> @constraint(model, con, 0 <= 2x - 1 <= 2)
con : 2 x ∈ [1.0, 3.0]

julia> add_to_function_constant(con, 4)

julia> con
con : 2 x ∈ [-3.0, -1.0]
```

For vector constraints, the constant is added to the function:

```
julia> @constraint(model, con, [x + y, x, y] in SecondOrderCone())
con : [x + y, x, y] ∈ MathOptInterface.SecondOrderCone(3)
```

```
julia> add_to_function_constant(con, [1, 2, 2])

julia> con
con : [x + y + 1, x + 2, y + 2] ∈ MathOptInterface.SecondOrderCone(3)
```

source

22.4 Deletion

JuMP.delete – Function.

```
delete(model::Model, con_ref::ConstraintRef)
```

Delete the constraint associated with `constraint_ref` from the model `model`.

Note that `delete` does not unregister the name from the model, so adding a new constraint of the same name will throw an error. Use `unregister` to unregister the name after deletion as follows:

```
@constraint(model, c, 2x <= 1)
delete(model, c)
unregister(model, :c)
```

See also: `unregister`

source

```
delete(model::Model, con_refs::Vector{<:ConstraintRef})
```

Delete the constraints associated with `con_refs` from the model `model`. Solvers may implement specialized methods for deleting multiple constraints of the same concrete type, i.e., when `isconcretetype(eltype(con_refs))`. These may be more efficient than repeatedly calling the single constraint delete method.

See also: `unregister`

source

```
delete(model::Model, variable_ref::VariableRef)
```

Delete the variable associated with `variable_ref` from the model `model`.

Note that `delete` does not unregister the name from the model, so adding a new variable of the same name will throw an error. Use `unregister` to unregister the name after deletion as follows:

```
@variable(model, x)
delete(model, x)
unregister(model, :x)
```

See also: `unregister`

source

```
delete(model::Model, variable_refs::Vector{VariableRef})
```

Delete the variables associated with `variable_refs` from the model `model`. Solvers may implement methods for deleting multiple variables that are more efficient than repeatedly calling the single variable delete method.

See also: `unregister`

source

JuMP.is_valid – Function.

```
is_valid(model::Model, con_ref::ConstraintRef{<:AbstractModel})
```

Return `true` if `constraint_ref` refers to a valid constraint in `model`.

source

```
is_valid(model::Model, variable_ref::VariableRef)
```

Return `true` if `variable` refers to a valid variable in `model`.

source

```
is_valid(model::Model, c::ConstraintRef{Model,NonlinearConstraintIndex})
```

Return `true` if `c` refers to a valid nonlinear constraint in `model`.

source

JuMP.ConstraintNotOwned – Type.

```
struct ConstraintNotOwned{C <: ConstraintRef} <: Exception
    constraint_ref::C
end
```

The constraint `constraint_ref` was used in a model different to `owner_model(constraint_ref)`.

source

22.5 Query constraints

JuMP.list_of_constraint_types – Function.

```
list_of_constraint_types(model::Model)::Vector{Tuple{Type,Type}}
```

Return a list of tuples of the form `(F, S)` where `F` is a JuMP function type and `S` is an MOI set type such that `all_constraints(model, F, S)` returns a nonempty list.

Example

```
julia> model = Model();

julia> @variable(model, x >= 0, Bin);

julia> @constraint(model, 2x <= 1);

julia> list_of_constraint_types(model)
3-element Array{Tuple{Type,Type},1}:
 (GenericAffExpr{Float64,VariableRef}, MathOptInterface.LessThan{Float64})
 (VariableRef, MathOptInterface.GreaterThan{Float64})
 (VariableRef, MathOptInterface.ZeroOne)
```

source

JuMP.all_constraints – Function.

```
all_constraints(model::Model, function_type, set_type)::Vector{<:ConstraintRef}
```

Return a list of all constraints currently in the model where the function has type `function_type` and the set has type `set_type`. The constraints are ordered by creation time.

See also `list_of_constraint_types` and `num_constraints`.

Example

```
julia> model = Model();

julia> @variable(model, x >= 0, Bin);

julia> @constraint(model, 2x <= 1);

julia> all_constraints(model, VariableRef, MOI.GreaterThan{Float64})
1-element
↪  Array{ConstraintRef{Model,MathOptInterface.ConstraintIndex{MathOptInterface.VariableIndex,MathOptInterface.Gre
 x ≥ 0.0

julia> all_constraints(model, VariableRef, MOI.ZeroOne)
1-element
↪  Array{ConstraintRef{Model,MathOptInterface.ConstraintIndex{MathOptInterface.VariableIndex,MathOptInterface.Zer
 x binary

julia> all_constraints(model, AffExpr, MOI.LessThan{Float64})
1-element
↪  Array{ConstraintRef{Model,MathOptInterface.ConstraintIndex{MathOptInterface.ScalarAffineFunction{Float64},Math
 2 x ≤ 1.0
```

source

`JuMP.num_constraints` – Function.

```
num_constraints(model::Model, function_type, set_type)::Int64
```

Return the number of constraints currently in the model where the function has type `function_type` and the set has type `set_type`.

See also `list_of_constraint_types` and `all_constraints`.

Example

```
julia> model = Model();

julia> @variable(model, x >= 0, Bin);

julia> @variable(model, y);

julia> @constraint(model, y in MOI.GreaterThan(1.0));

julia> @constraint(model, y <= 1.0);

julia> @constraint(model, 2x <= 1);

julia> num_constraints(model, VariableRef, MOI.GreaterThan{Float64})
2

julia> num_constraints(model, VariableRef, MOI.ZeroOne)
1
```

```
julia> num_constraints(model, AffExpr, MOI.LessThan{Float64})
2
```

source

JuMP.index – Method.

```
index(cr::ConstraintRef)::MOI.ConstraintIndex
```

Return the index of the constraint that corresponds to `cr` in the MOI backend.

source

JuMP.optimizer_index – Method.

```
optimizer_index(cr::ConstraintRef{Model})::MOI.ConstraintIndex
```

Return the index of the constraint that corresponds to `cr` in the optimizer model. It throws NoOptimizer if no optimizer is set and throws an `ErrorException` if the optimizer is set but is not attached or if the constraint is bridged.

source

JuMP.constraint_object – Function.

```
constraint_object(con_ref::ConstraintRef)
```

Return the underlying constraint data for the constraint referenced by `ref`.

source

22.6 Start values

JuMP.set_dual_start_value – Function.

```
set_dual_start_value(con_ref::ConstraintRef, value)
```

Set the dual start value (MOI attribute `ConstraintDualStart`) of the constraint `con_ref` to `value`. To remove a dual start value set it to `nothing`.

See also dual_start_value.

source

JuMP.dual_start_value – Function.

```
dual_start_value(con_ref::ConstraintRef)
```

Return the dual start value (MOI attribute `ConstraintDualStart`) of the constraint `con_ref`.

Note: If no dual start value has been set, `dual_start_value` will return `nothing`.

See also set_dual_start_value.

source

22.7 Special sets

JuMP.SecondOrderCone – Type.

```
SecondOrderCone
```

Second order cone object that can be used to constrain the euclidean norm of a vector `x` to be less than or equal to a nonnegative scalar `t`. This is a shortcut for the `MOI.SecondOrderCone`.

Examples

The following constrains $\|(x-1, x-2)\|_2 \leq t$ and $t \geq 0$:

```
julia> model = Model();

julia> @variable(model, x)
x

julia> @variable(model, t)
t

julia> @constraint(model, [t, x-1, x-2] in SecondOrderCone())
[t, x - 1, x - 2] ∈ MathOptInterface.SecondOrderCone(3)
```

source

JuMP.RotatedSecondOrderCone – Type.

```
RotatedSecondOrderCone
```

Rotated second order cone object that can be used to constrain the square of the euclidean norm of a vector `x` to be less than or equal to $2tu$ where `t` and `u` are nonnegative scalars. This is a shortcut for the `MOI.RotatedSecondOrderCone`.

Examples

The following constrains $\|(x-1, x-2)\|_2 \leq 2tx$ and $t, x \geq 0$:

```
julia> model = Model();

julia> @variable(model, x)
x

julia> @variable(model, t)
t

julia> @constraint(model, [t, x, x-1, x-2] in RotatedSecondOrderCone())
[t, x, x - 1, x - 2] ∈ MathOptInterface.RotatedSecondOrderCone(4)
```

source

JuMP.PSDCone – Type.

```
PSDCone
```

Positive semidefinite cone object that can be used to constrain a square matrix to be positive semidefinite in the `@constraint` macro. If the matrix has type `Symmetric` then the columns vectorization (the

vector obtained by concatenating the columns) of its upper triangular part is constrained to belong to the `MOI.PositiveSemidefiniteConeTriangle` set, otherwise its column vectorization is constrained to belong to the `MOI.PositiveSemidefiniteConeSquare` set.

Examples

Consider the following example:

```
julia> model = Model();

julia> @variable(model, x)
x

julia> a = [ x 2x
             2x  x];

julia> b = [1 2
            2 4];

julia> cref = @SDconstraint(model, a  b)
[x - 1    2 x - 2;
 2 x - 2  x - 4  ] ∈ PSDCone()

julia> jump_function(constraint_object(cref))
4-element Array{GenericAffExpr{Float64,VariableRef},1}:
 x - 1
 2 x - 2
 2 x - 2
 x - 4

julia> moi_set(constraint_object(cref))
MathOptInterface.PositiveSemidefiniteConeSquare(2)
```

We see in the output of the last command that the matrix the vectorization of the matrix is constrained to belong to the `PositiveSemidefiniteConeSquare`.

```
julia> using LinearAlgebra # For Symmetric

julia> cref = @constraint(model, Symmetric(a - b) in PSDCone())
[x - 1    2 x - 2;
 2 x - 2  x - 4  ] ∈ PSDCone()

julia> jump_function(constraint_object(cref))
3-element Array{GenericAffExpr{Float64,VariableRef},1}:
 x - 1
 2 x - 2
 x - 4

julia> moi_set(constraint_object(cref))
MathOptInterface.PositiveSemidefiniteConeTriangle(2)
```

As we see in the output of the last command, the vectorization of only the upper triangular part of the matrix is constrained to belong to the `PositiveSemidefiniteConeSquare`.

source

`JuMP.SOS1` – Type.

```
SOS1
```

SOS1 (Special Ordered Sets type 1) object than can be used to constrain a vector x to a set where at most 1 variable can take a non-zero value, all others being at 0. The `weights`, when specified, induce an ordering of the variables; as such, they should be unique values. The kth element in the set corresponds to the kth weight in `weights`. See here for a description of SOS constraints and their potential uses. This is a shortcut for the `MathOptInterface.SOS1` set.

source

`JuMP.SOS2` – Type.

```
SOS2
```

SOS1 (Special Ordered Sets type 2) object than can be used to constrain a vector x to a set where at most 2 variables can take a non-zero value, all others being at 0. In addition, if two are non-zero these must be consecutive in their ordering. The `weights` induce an ordering of the variables; as such, they should be unique values. The kth element in the set corresponds to the kth weight in `weights`. See here for a description of SOS constraints and their potential uses. This is a shortcut for the `MathOptInterface.SOS2` set.

source

`JuMP.SkewSymmetricMatrixSpace` – Type.

```
SkewSymmetricMatrixSpace()
```

Use in the `@variable` macro to constrain a matrix of variables to be skew-symmetric.

Examples

```
@variable(model, Q[1:2, 1:2] in SkewSymmetricMatrixSpace())
```

source

`JuMP.SkewSymmetricMatrixShape` – Type.

```
SkewSymmetricMatrixShape
```

Shape object for a skew symmetric square matrix of `side_dimension` rows and columns. The vectorized form contains the entries of the upper-right triangular part of the matrix (without the diagonal) given column by column (or equivalently, the entries of the lower-left triangular part given row by row). The diagonal is zero.

source

`JuMP.moi_set` – Function.

```
moi_set(constraint::AbstractConstraint)
```

Return the set of the constraint `constraint` in the function-in-set form as a `MathOptInterface.AbstractSet`.

```
moi_set(s::AbstractVectorSet, dim::Int)
```

Returns the MOI set of dimension `dim` corresponding to the JuMP set `s`.

source

22.8 Printing

JuMP.function_string – Function.

```
function_string(print_mode::Type{<:JuMP.PrintMode},
                func::Union{JuMP.AbstractJuMPScalar,
                            Vector{<:JuMP.AbstractJuMPScalar}})
```

Return a `String` representing the function `func` using print mode `print_mode`.

source

JuMP.constraints_string – Function.

```
constraints_string(print_mode, model::AbstractModel)::Vector{String}
```

Return a list of `String`s describing each constraint of the model.

source

JuMP.in_set_string – Function.

```
in_set_string(print_mode::Type{<:PrintMode}, set)
```

Return a `String` representing the membership to the set `set` using print mode `print_mode`.

source

JuMP.show_constraints_summary – Function.

```
show_constraints_summary(io::IO, model::AbstractModel)
```

Write to `io` a summary of the number of constraints.

source

Chapter 23

Containers

More information can be found in the Containers section of the manual.

`JuMP.Containers` – Module.

```
Containers
```

Module defining the containers `DenseAxisArray` and `SparseAxisArray` that behaves as a regular `AbstractArray` but with custom indexes that are not necessarily integers.

source

`JuMP.Containers.DenseAxisArray` – Type.

```
DenseAxisArray(data::Array{T, N}, axes...) where {T, N}
```

Construct a JuMP array with the underlying data specified by the `data` array and the given axes. Exactly `N` axes must be provided, and their lengths must match `size(data)` in the corresponding dimensions.

Example

```
julia> array = JuMP.Containers.DenseAxisArray([1 2; 3 4], [:a, :b], 2:3)
2-dimensional DenseAxisArray{Int64,2,...} with index sets:
    Dimension 1, Symbol[:a, :b]
    Dimension 2, 2:3
And data, a 2×2 Array{Int64,2}:
 1  2
 3  4

julia> array[:b, 3]
4
```

source

```
DenseAxisArray{T}(undef, axes...) where T
```

Construct an uninitialized DenseAxisArray with element-type T indexed over the given axes.

Example

```
julia> array = JuMP.Containers.DenseAxisArray{Float64}(undef, [:a, :b], 1:2);

julia> fill!(array, 1.0)
```

```
2-dimensional DenseAxisArray{Float64,2,...} with index sets:
    Dimension 1, Symbol[:a, :b]
    Dimension 2, 1:2
And data, a 2×2 Array{Float64,2}:
 1.0  1.0
 1.0  1.0

julia> array[:a, 2] = 5.0
5.0

julia> array[:a, 2]
5.0

julia> array
2-dimensional DenseAxisArray{Float64,2,...} with index sets:
    Dimension 1, Symbol[:a, :b]
    Dimension 2, 1:2
And data, a 2×2 Array{Float64,2}:
 1.0  5.0
 1.0  1.0
```

source

JuMP.Containers.SparseAxisArray – Type.

```
struct SparseAxisArray{T,N,K<:NTuple{N, Any}} <: AbstractArray{T,N}
    data::Dict{K,T}
end
```

N-dimensional array with elements of type T where only a subset of the entries are defined. The entries with indices `idx = (i1, i2, ..., iN)` in `keys(data)` has value `data[idx]`. Note that as opposed to `SparseArrays.AbstractSparseArray`, the missing entries are not assumed to be `zero(T)`, they are simply not part of the array. This means that the result of `map(f, sa::SparseAxisArray)` or `f.(sa::SparseAxisArray)` has the same sparsity structure than `sa` even if `f(zero(T))` is not zero.

Examples

```
julia> dict = Dict((:a, 2) => 1.0, (:a, 3) => 2.0, (:b, 3) => 3.0)
Dict{Tuple{Symbol,Int64},Float64} with 3 entries:
  (:b, 3) => 3.0
  (:a, 2) => 1.0
  (:a, 3) => 2.0

julia> array = JuMP.Containers.SparseAxisArray(dict)
JuMP.Containers.SparseAxisArray{Float64,2,Tuple{Symbol,Int64}} with 3 entries:
  [b, 3]  =  3.0
  [a, 2]  =  1.0
  [a, 3]  =  2.0

julia> array[:b, 3]
3.0
```

source

JuMP.Containers.container – Function.

```
container(f::Function, indices, ::Type{C})
```

Create a container of type `C` with indices `indices` and values at given indices given by `f`.

```
container(f::Function, indices)
```

Create a container with indices `indices` and values at given indices given by `f`. The type of container used is determined by `default_container`.

Examples

```
julia> Containers.container((i, j) -> i + j, Containers.vectorized_product(Base.OneTo(3), Base.
    OneTo(3)))
3×3 Array{Int64,2}:
 2  3  4
 3  4  5
 4  5  6

julia> Containers.container((i, j) -> i + j, Containers.vectorized_product(1:3, 1:3))
2-dimensional DenseAxisArray{Int64,2,...} with index sets:
    Dimension 1, 1:3
    Dimension 2, 1:3
And data, a 3×3 Array{Int64,2}:
 2  3  4
 3  4  5
 4  5  6

julia> Containers.container((i, j) -> i + j, Containers.vectorized_product(2:3, Base.OneTo(3)))
2-dimensional DenseAxisArray{Int64,2,...} with index sets:
    Dimension 1, 2:3
    Dimension 2, Base.OneTo(3)
And data, a 2×3 Array{Int64,2}:
 3  4  5
 4  5  6

julia> Containers.container((i, j) -> i + j, Containers.nested(() -> 1:3, i -> i:3, condition = (
    i, j) -> isodd(i) || isodd(j)))
SparseAxisArray{Int64,2,Tuple{Int64,Int64}} with 5 entries:
  [1, 2]  =  3
  [2, 3]  =  5
  [3, 3]  =  6
  [1, 1]  =  2
  [1, 3]  =  4
```

source

`JuMP.Containers.default_container` – Function.

```
default_container(indices)
```

If `indices` is a `NestedIterator`, return a `SparseAxisArray`. Otherwise, `indices` should be a `VectorizedProductIterator` and the function returns `Array` if all iterators of the product are `Base.OneTo` and retunrs `DenseAxisArray` otherwise.

source

`JuMP.Containers.@container` – Macro.

```
@container([i=..., j=..., ...], expr[, container = :Auto])
```

Create a container with indices `i`, `j`, ... and values given by `expr` that may depend on the value of the indices.

```
@container(ref[i=..., j=..., ...], expr[, container = :Auto])
```

Same as above but the container is assigned to the variable of name `ref`.

The type of container can be controlled by the `container` keyword.

> **Note**
> When the index set is explicitly given as `1:n` for any expression `n`, it is transformed to `Base.OneTo(n)` before being given to `container`.

Examples

```
julia> Containers.@container([i = 1:3, j = 1:3], i + j)
3×3 Array{Int64,2}:
 2  3  4
 3  4  5
 4  5  6

julia> I = 1:3
1:3

julia> Containers.@container(x[i = I, j = I], i + j);

julia> x
2-dimensional DenseAxisArray{Int64,2,...} with index sets:
    Dimension 1, 1:3
    Dimension 2, 1:3
And data, a 3×3 Array{Int64,2}:
 2  3  4
 3  4  5
 4  5  6

julia> Containers.@container([i = 2:3, j = 1:3], i + j)
2-dimensional DenseAxisArray{Int64,2,...} with index sets:
    Dimension 1, 2:3
    Dimension 2, Base.OneTo(3)
And data, a 2×3 Array{Int64,2}:
 3  4  5
 4  5  6

julia> Containers.@container([i = 1:3, j = 1:3; i <= j], i + j)
JuMP.Containers.SparseAxisArray{Int64,2,Tuple{Int64,Int64}} with 6 entries:
  [1, 2]  =  3
  [2, 3]  =  5
  [3, 3]  =  6
  [2, 2]  =  4
  [1, 1]  =  2
  [1, 3]  =  4
```

source

`JuMP.Containers.VectorizedProductIterator` – Type.

```
struct VectorizedProductIterator{T}
    prod::Iterators.ProductIterator{T}
end
```

A wrapper type for `Iterators.ProuctIterator` that discards shape information and returns a `Vector`.

Construct a `VectorizedProductIterator` using `vectorized_product`.

source

`JuMP.Containers.vectorized_product` – Function.

```
vectorized_product(iterators...)
```

Created a `VectorizedProductIterator`.

Examples

```
vectorized_product(1:2, ["A", "B"])
```

source

`JuMP.Containers.NestedIterator` – Type.

```
struct NestedIterator{T}
    iterators::T # Tuple of functions
    condition::Function
end
```

Iterators over the tuples that are produced by a nested for loop.

Construct a `NestedIterator` using `nested`.

Example

If `length(iterators) == 3`:

```
x = NestedIterator(iterators, condition)
for (i1, i2, i3) in x
    # produces (i1, i2, i3)
end
```

is the same as

```
for i1 in iterators[1]()
    for i2 in iterator[2](i1)
        for i3 in iterator[3](i1, i2)
            if condition(i1, i2, i3)
                # produces (i1, i2, i3)
            end
        end
    end
end
```

source

`JuMP.Containers.nested` – Function.

```
nested(iterators...; condition = (args...) -> true)
```

Create a NestedIterator.

Example

```
nested(1:2, ["A", "B"]; condition = (i, j) -> isodd(i) || j == "B")
```

source

For advanced users, the following functions are provided to aid the writing of macros that use the container functionality.

JuMP.Containers.build_ref_sets – Function.

```
build_ref_sets(_error::Function, expr)
```

Helper function for macros to construct container objects.

> **Warning**
>
> This function is for advanced users implementing JuMP extensions. See container_code for more details.

Arguments

- _error: a function that takes a String and throws an error, potentially annotating the input string with extra information such as from which macro it was thrown from. Use error if you do not want a modified error message.
- expr: an Expr that specifies the container, e.g., :(x[i = 1:3, [:red, :blue], k = S; i + k <= 6])

Returns

1. index_vars: a Vector{Any} of names for the index variables, e.g., [:i, gensym(), :k]. These may also be expressions, like :((i, j)) from a call like :(x[(i, j) in S]).
2. indices: an iterator over the indices, e.g.,

```
Containers.NestedIterators(
    (1:3, [:red, :blue], S),
    (i, _, k) -> i + k <= 6,
)
```

Examples

See container_code for a worked example.

source

JuMP.Containers.container_code – Function.

```
container_code(
    index_vars::Vector{Any},
    indices::Expr,
    code,
    requested_container::Union{Symbol,Expr},
)
```

Used in macros to construct a call to `container`. This should be used in conjunction with `build_ref_sets`.

Arguments

- `index_vars::Vector{Any}`: a vector of names for the indices of the container. These may also be expressions, like `:((i, j))` from a call like `:(x[(i, j) in S])`.
- `indices::Expr`: an expression that evaluates to an iterator of the indices.
- `code`: an expression or literal constant for the value to be stored in the container as a function of the named `index_vars`.
- `requested_container`: passed to the third argument of `container`. For built-in JuMP types, choose one of `:Array`, `:DenseAxisArray`, `:SparseAxisArray`, or `:Auto`. For a user-defined container, this expression must evaluate to the correct type.

  Warning

  In most cases, you should `esc(code)` before passing it to `container_code`.

Examples

```
julia> macro foo(ref_sets, code)
           index_vars, indices = Containers.build_ref_sets(error, ref_sets)
           return Containers.container_code(
               index_vars,
               indices,
               esc(code),
               :Auto,
            )
       end
@foo (macro with 1 method)

julia> @foo(x[i=1:2, j=["A", "B"]], j^i)
2-dimensional DenseAxisArray{String,2,...} with index sets:
    Dimension 1, Base.OneTo(2)
    Dimension 2, ["A", "B"]
And data, a 2×2 Matrix{String}:
 "A"   "B"
 "AA"  "BB"
```

source

Chapter 24

Solutions

More information can be found in the Solutions section of the manual.

24.1 Basic utilities

JuMP.optimize! – Function.

```
optimize!(model::Model;
          ignore_optimize_hook=(model.optimize_hook === nothing),
          kwargs...)
```

Optimize the model. If an optimizer has not been set yet (see set_optimizer), a NoOptimizer error is thrown.

Keyword arguments kwargs are passed to the optimize_hook. An error is thrown if optimize_hook is nothing and keyword arguments are provided.

source

JuMP.NoOptimizer – Type.

```
struct NoOptimizer <: Exception end
```

No optimizer is set. The optimizer can be provided to the Model constructor or by calling set_optimizer.

source

JuMP.OptimizeNotCalled – Type.

```
struct OptimizeNotCalled <: Exception end
```

A result attribute cannot be queried before optimize! is called.

source

JuMP.solution_summary – Function.

```
solution_summary(model::Model; verbose::Bool = false)
```

Return a struct that can be used print a summary of the solution.

If verbose=true, write out the primal solution for every variable and the dual solution for every constraint, excluding those with empty names.

Examples

When called at the REPL, the summary is automatically printed:

```
julia> solution_summary(model)
[...]
```

Use `print` to force the printing of the summary from inside a function:

```
function foo(model)
    print(solution_summary(model))
    return
end
```

source

24.2 Termination status

`JuMP.termination_status` – Function.

```
termination_status(model::Model)
```

Return a `MOI.TerminationStatusCode` describing why the solver stopped (i.e., the `MOI.TerminationStatus` attribute).

source

`JuMP.raw_status` – Function.

```
raw_status(model::Model)
```

Return the reason why the solver stopped in its own words (i.e., the MathOptInterface model attribute `RawStatusString`).

source

`JuMP.result_count` – Function.

```
result_count(model::Model)
```

Return the number of results available to query after a call to `optimize!`.

source

24.3 Primal solutions

`JuMP.primal_status` – Function.

```
primal_status(model::Model; result::Int = 1)
```

Return a `MOI.ResultStatusCode` describing the status of the most recent primal solution of the solver (i.e., the `MOI.PrimalStatus` attribute) associated with the result index `result`.

See also: `result_count`.

source

JuMP.has_values – Function.

```
has_values(model::Model; result::Int = 1)
```

Return `true` if the solver has a primal solution in result index `result` available to query, otherwise return `false`.

See also `value` and `result_count`.

source

JuMP.value – Function.

```
value(con_ref::ConstraintRef; result::Int = 1)
```

Return the primal value of constraint `con_ref` associated with result index `result` of the most-recent solution returned by the solver.

That is, if `con_ref` is the reference of a constraint `func-in-set`, it returns the value of `func` evaluated at the value of the variables (given by `value(::VariableRef)`).

Use `has_values` to check if a result exists before asking for values.

See also: `result_count`.

Note

For scalar constraints, the constant is moved to the `set` so it is not taken into account in the primal value of the constraint. For instance, the constraint `@constraint(model, 2x + 3y + 1 == 5)` is transformed into `2x + 3y-in-MOI.EqualTo(4)` so the value returned by this function is the evaluation of `2x + 3y`. "'

source

```
value(var_value::Function, con_ref::ConstraintRef)
```

Evaluate the primal value of the constraint `con_ref` using `var_value(v)` as the value for each variable `v`.

source

```
value(v::VariableRef; result = 1)
```

Return the value of variable `v` associated with result index `result` of the most-recent returned by the solver.

Use `has_values` to check if a result exists before asking for values.

See also: `result_count`.

source

```
value(var_value::Function, v::VariableRef)
```

Evaluate the value of the variable `v` as `var_value(v)`.

source

```
value(var_value::Function, ex::GenericAffExpr)
```

Evaluate `ex` using `var_value(v)` as the value for each variable `v`.

source

```
value(v::GenericAffExpr; result::Int = 1)
```

Return the value of the `GenericAffExpr` `v` associated with result index `result` of the most-recent solution returned by the solver.

See also: `result_count`.

source

```
value(var_value::Function, ex::GenericQuadExpr)
```

Evaluate `ex` using `var_value(v)` as the value for each variable `v`.

source

```
value(v::GenericQuadExpr; result::Int = 1)
```

Return the value of the `GenericQuadExpr` `v` associated with result index `result` of the most-recent solution returned by the solver.

Replaces `getvalue` for most use cases.

See also: `result_count`.

source

```
value(p::NonlinearParameter)
```

Return the current value stored in the nonlinear parameter `p`.

Example

```
model = Model()
@NLparameter(model, p == 10)
value(p)

# output
10.0
```

source

```
value(var_value::Function, ex::NonlinearExpression)
```

Evaluate `ex` using `var_value(v)` as the value for each variable `v`.

source

```
value(ex::NonlinearExpression; result::Int = 1)
```

Return the value of the `NonlinearExpression` `ex` associated with result index `result` of the most-recent solution returned by the solver.

Replaces `getvalue` for most use cases.

See also: `result_count`.

source

24.4 Dual solutions

JuMP.dual_status – Function.

```
dual_status(model::Model; result::Int = 1)
```

Return a MOI.ResultStatusCode describing the status of the most recent dual solution of the solver (i.e., the MOI.DualStatus attribute) associated with the result index result.

See also: result_count.

source

JuMP.has_duals – Function.

```
has_duals(model::Model; result::Int = 1)
```

Return true if the solver has a dual solution in result index result available to query, otherwise return false.

See also dual, shadow_price, and result_count.

source

JuMP.dual – Function.

```
dual(con_ref::ConstraintRef; result::Int = 1)
```

Return the dual value of constraint con_ref associated with result index result of the most-recent solution returned by the solver.

Use has_dual to check if a result exists before asking for values.

See also: result_count, shadow_price.

source

```
dual(c::ConstraintRef{Model,NonlinearConstraintIndex})
```

Return the dual of the nonlinear constraint c.

source

JuMP.shadow_price – Function.

```
shadow_price(con_ref::ConstraintRef)
```

Return the change in the objective from an infinitesimal relaxation of the constraint.

This value is computed from dual and can be queried only when has_duals is true and the objective sense is MIN_SENSE or MAX_SENSE (not FEASIBILITY_SENSE). For linear constraints, the shadow prices differ at most in sign from the dual value depending on the objective sense.

See also reduced_cost.

Notes

- The function simply translates signs from dual and does not validate the conditions needed to guarantee the sensitivity interpretation of the shadow price. The caller is responsible, e.g., for checking whether the solver converged to an optimal primal-dual pair or a proof of infeasibility.

- The computation is based on the current objective sense of the model. If this has changed since the last solve, the results will be incorrect.
- Relaxation of equality constraints (and hence the shadow price) is defined based on which sense of the equality constraint is active.

source

JuMP.reduced_cost – Function.

```
reduced_cost(x::VariableRef)::Float64
```

Return the reduced cost associated with variable x.

Equivalent to querying the shadow price of the active variable bound (if one exists and is active).

See also: shadow_price.

source

24.5 Basic attributes

JuMP.objective_value – Function.

```
objective_value(model::Model; result::Int = 1)
```

Return the objective value associated with result index `result` of the most-recent solution returned by the solver.

See also: result_count.

source

JuMP.objective_bound – Function.

```
objective_bound(model::Model)
```

Return the best known bound on the optimal objective value after a call to `optimize!(model)`.

source

JuMP.dual_objective_value – Function.

```
dual_objective_value(model::Model; result::Int = 1)
```

Return the value of the objective of the dual problem associated with result index `result` of the most-recent solution returned by the solver.

Throws `MOI.UnsupportedAttribute{MOI.DualObjectiveValue}` if the solver does not support this attribute.

See also: result_count.

source

JuMP.solve_time – Function.

```
solve_time(model::Model)
```

If available, returns the solve time reported by the solver. Returns "ArgumentError: ModelLike of type `Solver.Optimizer` does not support accessing the attribute MathOptInterface.SolveTimeSec()" if the attribute is not implemented.

source

`JuMP.relative_gap` – Function.

```
relative_gap(model::Model)
```

Return the final relative optimality gap after a call to `optimize!(model)`. Exact value depends upon implementation of MathOptInterface.RelativeGap() by the particular solver used for optimization.

source

`JuMP.simplex_iterations` – Function.

```
simplex_iterations(model::Model)
```

Gets the cumulative number of simplex iterations during the most-recent optimization.

Solvers must implement `MOI.SimplexIterations()` to use this function.

source

`JuMP.barrier_iterations` – Function.

```
barrier_iterations(model::Model)
```

Gets the cumulative number of barrier iterations during the most recent optimization.

Solvers must implement `MOI.BarrierIterations()` to use this function.

source

`JuMP.node_count` – Function.

```
node_count(model::Model)
```

Gets the total number of branch-and-bound nodes explored during the most recent optimization in a Mixed Integer Program.

Solvers must implement `MOI.NodeCount()` to use this function.

source

24.6 Conflicts

`JuMP.compute_conflict!` – Function.

```
compute_conflict!(model::Model)
```

Compute a conflict if the model is infeasible. If an optimizer has not been set yet (see `set_optimizer`), a `NoOptimizer` error is thrown.

The status of the conflict can be checked with the `MOI.ConflictStatus` model attribute. Then, the status for each constraint can be queried with the `MOI.ConstraintConflictStatus` attribute.

source

JuMP.copy_conflict – Function.

```
copy_conflict(model::Model)
```

Return a copy of the current conflict for the model `model` and a `ReferenceMap` that can be used to obtain the variable and constraint reference of the new model corresponding to a given `model`'s reference.

This is a convenience function that provides a filtering function for `copy_model`.

Note

Model copy is not supported in `DIRECT` mode, i.e. when a model is constructed using the `direct_model` constructor instead of the `Model` constructor. Moreover, independently on whether an optimizer was provided at model construction, the new model will have no optimizer, i.e., an optimizer will have to be provided to the new model in the `optimize!` call.

Examples

In the following example, a model `model` is constructed with a variable `x` and two constraints `cref` and `cref2`. This model has no solution, as the two constraints are mutually exclusive. The solver is asked to compute a conflict with `compute_conflict!`. The parts of `model` participating in the conflict are then copied into a model `new_model`.

```
model = Model() # You must use a solver that supports conflict refining/IIS
# computation, like CPLEX or Gurobi
@variable(model, x)
@constraint(model, cref, x >= 2)
@constraint(model, cref2, x <= 1)

compute_conflict!(model)
if MOI.get(model, MOI.ConflictStatus()) != MOI.CONFLICT_FOUND
    error("No conflict could be found for an infeasible model.")
end

new_model, reference_map = copy_conflict(model)
```

source

24.7 Sensitivity

JuMP.lp_sensitivity_report – Function.

```
lp_sensitivity_report(model::Model; atol::Float64 = 1e-8)::SensitivityReport
```

Given a linear program `model` with a current optimal basis, return a `SensitivityReport` object, which maps:

- Every variable reference to a tuple `(d_lo, d_hi)::Tuple{Float64,Float64}`, explaining how much the objective coefficient of the corresponding variable can change by, such that the original basis remains optimal.
- Every constraint reference to a tuple `(d_lo, d_hi)::Tuple{Float64,Float64}`, explaining how much the right-hand side of the corresponding constraint can change by, such that the basis remains optimal.

Both tuples are relative, rather than absolute. So given a objective coefficient of `1.0` and a tuple `(-0.5, 0.5)`, the objective coefficient can range between `1.0 - 0.5` an `1.0 + 0.5`.

`atol` is the primal/dual optimality tolerance, and should match the tolerance of the solver used to compute the basis.

Note: interval constraints are NOT supported.

Example

```
model = Model(GLPK.Optimizer)
@variable(model, -1 <= x <= 2)
@objective(model, Min, x)
optimize!(model)
report = lp_sensitivity_report(model; atol = 1e-7)
dx_lo, dx_hi = report[x]
println(
    "The objective coefficient of `x` can decrease by $dx_lo or " *
    "increase by $dx_hi."
)
c = LowerBoundRef(x)
dRHS_lo, dRHS_hi = report[c]
println(
    "The lower bound of `x` can decrease by $dRHS_lo or increase " *
    "by $dRHS_hi."
)
```

source

JuMP.SensitivityReport – Type.

```
SensitivityReport
```

See lp_sensitivity_report.

source

24.8 Feasibility

JuMP.primal_feasibility_report – Function.

```
primal_feasibility_report(
    model::Model,
    point::AbstractDict{VariableRef,Float64} = _last_primal_solution(model),
    atol::Float64 = 0.0,
    skip_missing::Bool = false,
)::Dict{Any,Float64}
```

Given a dictionary `point`, which maps variables to primal values, return a dictionary whose keys are the constraints with an infeasibility greater than the supplied tolerance `atol`. The value corresponding to each key is the respective infeasibility. Infeasibility is defined as the distance between the primal value of the constraint (see `MOI.ConstraintPrimal`) and the nearest point by Euclidean distance in the corresponding set.

Notes

- If `skip_missing = true`, constraints containing variables that are not in `point` will be ignored.

- If `skip_missing = false` and a partial primal solution is provided, an error will be thrown.
- If no point is provided, the primal solution from the last time the model was solved is used.

Examples

```
julia> model = Model();

julia> @variable(model, 0.5 <= x <= 1);

julia> primal_feasibility_report(model, Dict(x => 0.2))
Dict{Any,Float64} with 1 entry:
  x ≥ 0.5 => 0.3
```

source

```
primal_feasibility_report(
    point::Function,
    model::Model;
    atol::Float64 = 0.0,
    skip_missing::Bool = false,
)
```

A form of `primal_feasibility_report` where a function is passed as the first argument instead of a dictionary as the second argument.

Examples

```
julia> model = Model();

julia> @variable(model, 0.5 <= x <= 1);

julia> primal_feasibility_report(model) do v
           return value(v)
       end
Dict{Any,Float64} with 1 entry:
    x ≥ 0.5 => 0.3
```

source

Chapter 25

Nonlinear Modeling

More information can be found in the Nonlinear Modeling section of the manual.

25.1 Constraints

JuMP.@NLconstraint – Macro.

```
@NLconstraint(m::Model, expr)
```

Add a constraint described by the nonlinear expression `expr`. See also `@constraint`. For example:

```
@NLconstraint(model, sin(x) <= 1)
@NLconstraint(model, [i = 1:3], sin(i * x) <= 1 / i)
```

source

JuMP.@NLconstraints – Macro.

```
@NLconstraints(model, args...)
```

Adds multiple nonlinear constraints to model at once, in the same fashion as the `@NLconstraint` macro.

The model must be the first argument, and multiple variables can be added on multiple lines wrapped in a `begin ... end` block.

Examples

```
@NLconstraints(model, begin
    t >= sqrt(x^2 + y^2)
    [i = 1:2], z[i] <= log(a[i])
end)
```

source

JuMP.NonlinearConstraintIndex – Type.

```
NonlinearConstraintIndex(index::Int64)
```

A struct to refer to the 1-indexed nonlinear constraint `index`.

source

JuMP.num_nl_constraints – Function.

```
num_nl_constraints(model::Model)
```

Returns the number of nonlinear constraints associated with the `model`.

source

JuMP.add_NL_constraint – Function.

```
add_NL_constraint(model::Model, expr::Expr)
```

Add a nonlinear constraint described by the Julia expression ex to `model`.

This function is most useful if the expression ex is generated programmatically, and you cannot use `@NLconstraint`.

Notes

- You must interpolate the variables directly into the expression `expr`.

Examples

```
julia> add_NL_constraint(model, :($(x) + $(x)^2 <= 1))
(x + x ^ 2.0) - 1.0 ≤ 0
```

source

25.2 Expressions

JuMP.@NLexpression – Macro.

```
@NLexpression(args...)
```

Efficiently build a nonlinear expression which can then be inserted in other nonlinear constraints and the objective. See also [@expression]. For example:

```
@NLexpression(model, my_expr, sin(x)^2 + cos(x^2))
@NLconstraint(model, my_expr + y >= 5)
@NLobjective(model, Min, my_expr)
```

Indexing over sets and anonymous expressions are also supported:

```
@NLexpression(m, my_expr_1[i=1:3], sin(i * x))
my_expr_2 = @NLexpression(m, log(1 + sum(exp(x[i])) for i in 1:2))
```

source

JuMP.@NLexpressions – Macro.

```
@NLexpressions(model, args...)
```

Adds multiple nonlinear expressions to model at once, in the same fashion as the `@NLexpression` macro.

The model must be the first argument, and multiple variables can be added on multiple lines wrapped in a `begin ... end` block.

Examples

```
@NLexpressions(model, begin
    my_expr, sqrt(x^2 + y^2)
    my_expr_1[i = 1:2], log(a[i]) - z[i]
end)
```

source

JuMP.NonlinearExpression – Type.

```
NonlinearExpression <: AbstractJuMPScalar
```

A struct to represent a nonlinear expression.

Create an expression using @NLexpression.

source

JuMP.add_NL_expression – Function.

```
add_NL_expression(model::Model, expr::Expr)
```

Add a nonlinear expression expr to model.

This function is most useful if the expression expr is generated programmatically, and you cannot use @NLexpression.

Notes

- You must interpolate the variables directly into the expression expr.

Examples

```
julia> add_NL_expression(model, :($(x) + $(x)^2))
"subexpression[1]: x + x ^ 2.0"
```

source

25.3 Objectives

JuMP.@NLobjective – Macro.

```
@NLobjective(model, sense, expression)
```

Add a nonlinear objective to model with optimization sense sense. sense must be Max or Min.

Example

```
@NLobjective(model, Max, 2x + 1 + sin(x))
```

source

JuMP.set_NL_objective – Function.

```
set_NL_objective(model::Model, sense::MOI.OptimizationSense, expr::Expr)
```

Set the nonlinear objective of model to the expression expr, with the optimization sense sense.

This function is most useful if the expression expr is generated programmatically, and you cannot use @NLobjective.

Notes

- You must interpolate the variables directly into the expression `expr`.
- You must use `MOI.MIN_SENSE` or `MOI.MAX_SENSE` instead of `Min` and `Max`.

Examples

```
julia> set_NL_objective(model, MOI.MIN_SENSE, :($(x) + $(x)^2))
```

source

25.4 Parameters

`JuMP.@NLparameter` – Macro.

```
@NLparameter(model, param == value)
```

Create and return a nonlinear parameter `param` attached to the model `model` with initial value set to `value`. Nonlinear parameters may be used only in nonlinear expressions.

Example

```
model = Model()
@NLparameter(model, x == 10)
value(x)

# output
10.0
```

```
@NLparameter(model, value = param_value)
```

Create and return an anonymous nonlinear parameter `param` attached to the model `model` with initial value set to `param_value`. Nonlinear parameters may be used only in nonlinear expressions.

Example

```
model = Model()
x = @NLparameter(model, value = 10)
value(x)

# output
10.0
```

```
@NLparameter(model, param_collection[...] == value_expr)
```

Create and return a collection of nonlinear parameters `param_collection` attached to the model `model` with initial value set to `value_expr` (may depend on index sets). Uses the same syntax for specifying index sets as `@variable`.

Example

```
model = Model()
@NLparameter(model, y[i = 1:10] == 2 * i)
value(y[9])

# output
18.0
```

```
@NLparameter(model, [...] == value_expr)
```

Create and return an anonymous collection of nonlinear parameters attached to the model `model` with initial value set to `value_expr` (may depend on index sets). Uses the same syntax for specifying index sets as `@variable`.

Example

```
model = Model()
y = @NLparameter(model, [i = 1:10] == 2 * i)
value(y[9])

# output
18.0
```

source

`JuMP.@NLparameters` – Macro.

```
@NLparameters(model, args...)
```

Create and return multiple nonlinear parameters attached to model `model`, in the same fashion as `@NLparameter` macro.

The model must be the first argument, and multiple parameters can be added on multiple lines wrapped in a `begin ... end` block. Distinct parameters need to be placed on separate lines as in the following example.

Example

```
model = Model()
@NLparameters(model, begin
    x == 10
    b == 156
end)
value(x)

# output
10.0
```

source

`JuMP.NonlinearParameter` – Type.

```
NonlinearParameter <: AbstractJuMPScalar
```

A struct to represent a nonlinear parameter.

Create a parameter using `@NLparameter`.

source

`JuMP.value` – Method.

```
value(p::NonlinearParameter)
```

Return the current value stored in the nonlinear parameter `p`.

Example

```
model = Model()
@NLparameter(model, p == 10)
value(p)

# output
10.0
```

source

`JuMP.set_value` – Method.

```
set_value(p::NonlinearParameter, v::Number)
```

Store the value `v` in the nonlinear parameter `p`.

Example

```
model = Model()
@NLparameter(model, p == 0)
set_value(p, 5)
value(p)

# output
5.0
```

source

25.5 User-defined functions

`JuMP.register` – Function.

```
register(
    model::Model,
    s::Symbol,
    dimension::Integer,
    f::Function;
    autodiff:Bool = false,
)
```

Register the user-defined function `f` that takes `dimension` arguments in `model` as the symbol `s`.

The function `f` must support all subtypes of `Real` as arguments. Do not assume that the inputs are `Float64`.

Notes

- For this method, you must explicitly set `autodiff = true`, because no user-provided gradient function `∇f` is given.
- Second-derivative information is only computed if `dimension == 1`.
- `s` does not have to be the same symbol as `f`, but it is generally more readable if it is.

Examples

```
model = Model()
@variable(model, x)
f(x::T) where {T<:Real} = x^2
register(model, :foo, 1, f; autodiff = true)
@NLobjective(model, Min, foo(x))
```

```
model = Model()
@variable(model, x[1:2])
g(x::T, y::T) where {T<:Real} = x * y
register(model, :g, 2, g; autodiff = true)
@NLobjective(model, Min, g(x[1], x[2]))
```

source

```
register(
    model::Model,
    s::Symbol,
    dimension::Integer,
    f::Function,
    ∇f::Function;
    autodiff:Bool = false,
)
```

Register the user-defined function `f` that takes `dimension` arguments in `model` as the symbol `s`. In addition, provide a gradient function `∇f`.

The functions `f` and `∇f` must support all subtypes of `Real` as arguments. Do not assume that the inputs are `Float64`.

Notes

- If the function `f` is univariate (i.e., `dimension == 1`), `∇f` must return a number which represents the first-order derivative of the function `f`.
- If the function `f` is multi-variate, `∇f` must have a signature matching `∇f(g::AbstractVector{T}, args::T...) where {T<:Real}`, where the first argument is a vector `g` that is modified in-place with the gradient.
- If `autodiff = true` and `dimension == 1`, use automatic differentiation to compute the second-order derivative information. If `autodiff = false`, only first-order derivative information will be used.
- `s` does not have to be the same symbol as `f`, but it is generally more readable if it is.

Examples

```
model = Model()
@variable(model, x)
f(x::T) where {T<:Real} = x^2
∇f(x::T) where {T<:Real} = 2 * x
register(model, :foo, 1, f, ∇f; autodiff = true)
@NLobjective(model, Min, foo(x))
```

```
model = Model()
@variable(model, x[1:2])
g(x::T, y::T) where {T<:Real} = x * y
function ∇g(g::AbstractVector{T}, x::T, y::T) where {T<:Real}
    g[1] = y
    g[2] = x
```

```
    return
end
register(model, :g, 2, g, ∇g; autodiff = true)
@NLobjective(model, Min, g(x[1], x[2]))
```

source

```
register(
    model::Model,
    s::Symbol,
    dimension::Integer,
    f::Function,
    ∇f::Function,
    ∇²f::Function,
)
```

Register the user-defined function `f` that takes `dimension` arguments in `model` as the symbol `s`. In addition, provide a gradient function `∇f` and a hessian function `∇²f`.

`∇f` and `∇²f` must return numbers corresponding to the first- and second-order derivatives of the function `f` respectively.

Notes

- Because automatic differentiation is not used, you can assume the inputs are all `Float64`.
- This method will throw an error if `dimension > 1`.
- `s` does not have to be the same symbol as `f`, but it is generally more readable if it is.

Examples

```
model = Model()
@variable(model, x)
f(x::Float64) = x^2
∇f(x::Float64) = 2 * x
∇²f(x::Float64) = 2.0
register(model, :foo, 1, f, ∇f, ∇²f)
@NLobjective(model, Min, foo(x))
```

source

25.6 Derivatives

`JuMP.NLPEvaluator` – Type.

```
NLPEvaluator(m::Model)
```

Return an `MOI.AbstractNLPEvaluator` constructed from the model `model`.

Before using, you must initialize the evaluator using `MOI.initialize`.

source

Chapter 26

Callbacks

More information can be found in the Callbacks section of the manual.

26.1 Macros

JuMP.@build_constraint – Macro.

```
@build_constraint(constraint_expr)
```

Constructs a `ScalarConstraint` or `VectorConstraint` using the same machinery as `@constraint` but without adding the constraint to a model.

Constraints using broadcast operators like `x .<= 1` are also supported and will create arrays of `ScalarConstraint` or `VectorConstraint`.

Examples

```
model = Model();
@variable(model, x);
@build_constraint(2x >= 1)

# output
ScalarConstraint{GenericAffExpr{Float64,VariableRef},MathOptInterface.GreaterThan{Float64}}(2 x,
↪  MathOptInterface.GreaterThan{Float64}(1.0))
```

source

26.2 Callback variable primal

JuMP.callback_value – Function.

```
callback_value(cb_data, x::VariableRef)
```

Return the primal solution of a variable inside a callback.

`cb_data` is the argument to the callback function, and the type is dependent on the solver.

source

```
callback_value(cb_data, expr::Union{GenericAffExpr, GenericQuadExpr})
```

Return the primal solution of an affine or quadratic expression inside a callback by getting the value for each variable appearing in the expression.

`cb_data` is the argument to the callback function, and the type is dependent on the solver.

source

26.3 Callback node status

`JuMP.callback_node_status` – Function.

```
callback_node_status(cb_data, model::Model)
```

Return an `MOI.CallbackNodeStatusCode` enum, indicating if the current primal solution available from `callback_value` is integer feasible.

source

Chapter 27

Extensions

More information can be found in the Extensions section of the manual.

27.1 Define a new set

JuMP.AbstractVectorSet – Type.

```
AbstractVectorSet
```

An abstract type for defining new sets in JuMP.

Implement `moi_set(::AbstractVectorSet, dim::Int)` to convert the type into an MOI set.

See also: moi_set.

source

27.2 Extend `@variable`

JuMP.ScalarVariable – Type.

```
ScalarVariable{S,T,U,V} <: AbstractVariable
```

A struct used when adding variables.

See also: add_variable.

source

JuMP.VariableInfo – Type.

```
VariableInfo{S,T,U,V}
```

A struct by JuMP internally when creating variables. This may also be used by JuMP extensions to create new types of variables.

See also: ScalarVariable.

source

JuMP.add_variable – Function.

```
add_variable(m::Model, v::AbstractVariable, name::String="")
```

Add a variable `v` to `Model m` and sets its name.

source

`JuMP.build_variable` – Function.

```
build_variable(_error::Function, variables, ::SymMatrixSpace)
```

Return a `VariablesConstrainedOnCreation` of shape `SymmetricMatrixShape` creating variables in `MOI.Reals`, i.e. "free" variables unless they are constrained after their creation.

This function is used by the `@variable` macro as follows:

```
@variable(model, Q[1:2, 1:2], Symmetric)
```

source

```
build_variable(_error::Function, variables, ::SkewSymmetricMatrixSpace)
```

Return a `VariablesConstrainedOnCreation` of shape `SkewSymmetricMatrixShape` creating variables in `MOI.Reals`, i.e. "free" variables unless they are constrained after their creation.

This function is used by the `@variable` macro as follows:

```
@variable(model, Q[1:2, 1:2] in SkewSymmetricMatrixSpace())
```

source

```
build_variable(_error::Function, variables, ::PSDCone)
```

Return a `VariablesConstrainedOnCreation` of shape `SymmetricMatrixShape` constraining the variables to be positive semidefinite.

This function is used by the `@variable` macro as follows:

```
@variable(model, Q[1:2, 1:2], PSD)
```

source

```
build_variable(
    _error::Function,
    info::VariableInfo,
    args...;
    kwargs...,
)
```

Return a new `AbstractVariable` object.

This method should only be implemented by developers creating JuMP extensions. It should never be called by users of JuMP.

Arguments

- `_error`: a function to call instead of `error`. `_error` annotates the error message with additional information for the user.
- `info`: an instance of `VariableInfo`. This has a variety of fields relating to the variable such as `info.lower_bound` and `info.binary`.
- `args`: optional additional positional arguments for extending the `@variable` macro.

- kwargs: optional keyword arguments for extending the `@variable` macro.

See also: `@variable`

> **Warning**
> Extensions should define a method with ONE positional argument to dispatch the call to a different method. Creating an extension that relies on multiple positional arguments leads to `MethodErrors` if the user passes the arguments in the wrong order.

Examples

```
@variable(model, x, Foo)
```

will call

```
build_variable(_error::Function, info::VariableInfo, ::Type{Foo})
```

Passing special-case positional arguments such as `Bin`, `Int`, and `PSD` is okay, along with keyword arguments:

```
@variable(model, x, Int, Foo(), mykwarg = true)
# or
@variable(model, x, Foo(), Int, mykwarg = true)
```

will call

```
build_variable(_error::Function, info::VariableInfo, ::Foo; mykwarg)
```

and `info.integer` will be true.

Note that the order of the positional arguments does not matter.

source

27.3 Extend `@constraint`

`JuMP.build_constraint` – Function.

```
build_constraint(
    _error::Function,
    f::AbstractVector{<:AbstractJuMPScalar},
    s::MOI.GreaterThan,
    extra::Union{MOI.AbstractVectorSet,AbstractVectorSet},
)
```

A helper method that re-writes

```
@constraint(model, X >= Y, extra)
```

into

```
@constraint(model, X - Y in extra)
```

source

```
build_constraint(
    _error::Function,
    f::AbstractVector{<:AbstractJuMPScalar},
    s::MOI.LessThan,
    extra::Union{MOI.AbstractVectorSet,AbstractVectorSet},
)
```

A helper method that re-writes

```
@constraint(model, Y <= X, extra)
```

into

```
@constraint(model, X - Y in extra)
```

source

```
build_constraint(_error::Function, Q::Symmetric{V, M},
                 ::PSDCone) where {V <: AbstractJuMPScalar,
                                   M <: AbstractMatrix{V}}
```

Return a `VectorConstraint` of shape `SymmetricMatrixShape` constraining the matrix `Q` to be positive semidefinite.

This function is used by the `@constraint` macros as follows:

```
@constraint(model, Symmetric(Q) in PSDCone())
```

The form above is usually used when the entries of `Q` are affine or quadratic expressions, but it can also be used when the entries are variables to get the reference of the semidefinite constraint, e.g.,

```
@variable model Q[1:2,1:2] Symmetric
# The type of `Q` is `Symmetric{VariableRef, Matrix{VariableRef}}`
var_psd = @constraint model Q in PSDCone()
# The `var_psd` variable contains a reference to the constraint
```

source

```
build_constraint(
    _error::Function,
    Q::AbstractMatrix{<:AbstractJuMPScalar},
    ::PSDCone,
)
```

Return a `VectorConstraint` of shape `SquareMatrixShape` constraining the matrix `Q` to be symmetric and positive semidefinite.

This function is used by the `@constraint` macro as follows:

```
@constraint(model, Q in PSDCone())
```

source

`JuMP.add_constraint` – Function.

```
add_constraint(model::Model, con::AbstractConstraint, name::String="")
```

Add a constraint con to `Model model` and sets its name.

source

`JuMP.AbstractShape` – Type.

```
AbstractShape
```

Abstract vectorizable shape. Given a flat vector form of an object of shape shape, the original object can be obtained by `reshape_vector`.

source

`JuMP.shape` – Function.

```
shape(c::AbstractConstraint)::AbstractShape
```

Return the shape of the constraint `c`.

source

`JuMP.reshape_vector` – Function.

```
reshape_vector(vectorized_form::Vector, shape::AbstractShape)
```

Return an object in its original shape shape given its vectorized form `vectorized_form`.

Examples

Given a `SymmetricMatrixShape` of vectorized form `[1, 2, 3]`, the following code returns the matrix `Symmetric(Matrix[1 2; 2 3])`:

```
julia> reshape_vector([1, 2, 3], SymmetricMatrixShape(2))
2×2 LinearAlgebra.Symmetric{Int64,Array{Int64,2}}:
 1  2
 2  3
```

source

`JuMP.reshape_set` – Function.

```
reshape_set(vectorized_set::MOI.AbstractSet, shape::AbstractShape)
```

Return a set in its original shape shape given its vectorized form `vectorized_form`.

Examples

Given a `SymmetricMatrixShape` of vectorized form `[1, 2, 3] in MOI.PositiveSemidefinieConeTriangle(2)`, the following code returns the set of the original constraint `Symmetric(Matrix[1 2; 2 3]) in PSDCone()`:

```
julia> reshape_set(MOI.PositiveSemidefiniteConeTriangle(2), SymmetricMatrixShape(2))
PSDCone()
```

source

`JuMP.dual_shape` – Function.

```
dual_shape(shape::AbstractShape)::AbstractShape
```

Returns the shape of the dual space of the space of objects of shape `shape`. By default, the `dual_shape` of a shape is itself. See the examples section below for an example for which this is not the case.

Examples

Consider polynomial constraints for which the dual is moment constraints and moment constraints for which the dual is polynomial constraints. Shapes for polynomials can be defined as follows:

```
struct Polynomial
    coefficients::Vector{Float64}
    monomials::Vector{Monomial}
end
struct PolynomialShape <: AbstractShape
    monomials::Vector{Monomial}
end
JuMP.reshape_vector(x::Vector, shape::PolynomialShape) = Polynomial(x, shape.monomials)
```

and a shape for moments can be defined as follows:

```
struct Moments
    coefficients::Vector{Float64}
    monomials::Vector{Monomial}
end
struct MomentsShape <: AbstractShape
    monomials::Vector{Monomial}
end
JuMP.reshape_vector(x::Vector, shape::MomentsShape) = Moments(x, shape.monomials)
```

Then `dual_shape` allows the definition of the shape of the dual of polynomial and moment constraints:

```
dual_shape(shape::PolynomialShape) = MomentsShape(shape.monomials)
dual_shape(shape::MomentsShape) = PolynomialShape(shape.monomials)
```

source

`JuMP.ScalarShape` – Type.

```
ScalarShape
```

Shape of scalar constraints.

source

`JuMP.VectorShape` – Type.

```
VectorShape
```

Vector for which the vectorized form corresponds exactly to the vector given.

source

`JuMP.SquareMatrixShape` – Type.

```
SquareMatrixShape
```

Shape object for a square matrix of `side_dimension` rows and columns. The vectorized form contains the entries of the the matrix given column by column (or equivalently, the entries of the lower-left triangular part given row by row).

source

JuMP.SymmetricMatrixShape – Type.

```
SymmetricMatrixShape
```

Shape object for a symmetric square matrix of `side_dimension` rows and columns. The vectorized form contains the entries of the upper-right triangular part of the matrix given column by column (or equivalently, the entries of the lower-left triangular part given row by row).

source

JuMP.operator_to_set – Function.

```
operator_to_set(_error::Function, ::Val{sense_symbol})
```

Converts a sense symbol to a set set such that `@constraint(model, func sense_symbol 0)` is equivalent to `@constraint(model, func in set)` for any `func::AbstractJuMPScalar`.

Example

Once a custom set is defined you can directly create a JuMP constraint with it:

```
julia> struct CustomSet{T} <: MOI.AbstractScalarSet
           value::T
       end

julia> Base.copy(x::CustomSet) = CustomSet(x.value)

julia> model = Model();

julia> @variable(model, x)
x

julia> cref = @constraint(model, x in CustomSet(1.0))
x ∈ CustomSet{Float64}(1.0)
```

However, there might be an appropriate sign that could be used in order to provide a more convenient syntax:

```
julia> JuMP.operator_to_set(::Function, ::Val{:}) = CustomSet(0.0)

julia> MOIU.shift_constant(set::CustomSet, value) = CustomSet(set.value + value)

julia> cref = @constraint(model, x  1)
x ∈ CustomSet{Float64}(1.0)
```

Note that the whole function is first moved to the right-hand side, then the sign is transformed into a set with zero constant and finally the constant is moved to the set with `MOIU.shift_constant`.

source

JuMP.parse_constraint – Function.

```
parse_constraint(_error::Function, expr::Expr)
```

The entry-point for all constraint-related parsing.

Arguments

- The `_error` function is passed everywhere to provide better error messages
- `expr` comes from the `@constraint` macro. There are two possibilities:
  - `@constraint(model, expr)`
  - `@constraint(model, name[args], expr)`

  In both cases, `expr` is the main component of the constraint.

Supported syntax

JuMP currently supports the following `expr` objects:

- `lhs <= rhs`
- `lhs == rhs`
- `lhs >= rhs`
- `l <= body <= u`
- `u >= body >= l`
- `lhs ⊥ rhs`
- `lhs in rhs`
- `lhs ∈ rhs`
- `z => {constraint}`
- `!z => {constraint}`

as well as all broadcasted variants.

Extensions

The infrastructure behind `parse_constraint` is extendable. See `parse_constraint_head` and `parse_constraint_call` for details.

source

`JuMP.parse_constraint_head` – Function.

```
parse_constraint_head(_error::Function, ::Val{head}, args...)
```

Implement this method to intercept the parsing of an expression with head `head`.

> **Warning**
> Extending the constraint macro at parse time is an advanced operation and has the potential to interfere with existing JuMP syntax. Please discuss with the developer chatroom before publishing any code that implements these methods.

Arguments

- `_error`: a function that accepts a `String` and throws the string as an error, along with some descriptive information of the macro from which it was thrown.
- `head`: the `.head` field of the `Expr` to intercept
- `args...`: the `.args` field of the `Expr`.

Returns

This function must return:

- `is_vectorized::Bool`: whether the expression represents a broadcasted expression like `x .<= 1`
- `parse_code::Expr`: an expression containing any setup or rewriting code that needs to be called before `build_constraint`
- `build_code::Expr`: an expression that calls `build_constraint(` or `build_constraint.(` depending on `is_vectorized`.

Existing implementations

JuMP currently implements:

- `::Val{:call}`, which forwards calls to `parse_constraint_call`
- `::Val{:comparison}`, which handles the special case of `l <= body <= u`.

See also: `parse_constraint_call`, `build_constraint`

source

`JuMP.parse_constraint_call` – Function.

```
parse_constraint_call(
    _error::Function,
    is_vectorized::Bool,
    ::Val{op},
    args...,
)
```

Implement this method to intercept the parsing of a `:call` expression with operator `op`.

> **Warning**
> Extending the constraint macro at parse time is an advanced operation and has the potential to interfere with existing JuMP syntax. Please discuss with the developer chatroom before publishing any code that implements these methods.

Arguments

- `_error`: a function that accepts a `String` and throws the string as an error, along with some descriptive information of the macro from which it was thrown.
- `is_vectorized`: a boolean to indicate if `op` should be broadcast or not
- `op`: the first element of the `.args` field of the `Expr` to intercept
- `args...`: the `.args` field of the `Expr`.

Returns

This function must return:

- `parse_code::Expr`: an expression containing any setup or rewriting code that needs to be called before `build_constraint`
- `build_code::Expr`: an expression that calls `build_constraint(` or `build_constraint.(` depending on `is_vectorized`.

See also: `parse_constraint_head`, `build_constraint`

source

```
parse_constraint_call(
    _error::Function,
    vectorized::Bool,
    ::Val{op},
    lhs,
    rhs,
) where {op}
```

Fallback handler for binary operators. These might be infix operators like `@constraint(model, lhs op rhs)`, or normal operators like `@constraint(model, op(lhs, rhs))`.

In both cases, we rewrite as `lhs - rhs in operator_to_set(_error, op)`.

See `operator_to_set` for details.

source

Part V

Background Information

Chapter 28

Should I use JuMP?

JuMP is an algebraic modeling language for mathematical optimization written in the Julia language.

28.1 When should I use JuMP?

You should use JuMP if you have a constrained optimization problem for which you can formulate a set of decision variables, a scalar objective function, and a set of constraints.

Key reasons to use JuMP include:

- User friendliness
  - Syntax that mimics natural mathematical expressions. (See the section on algebraic modeling languages.)
- Speed
  - Benchmarking has shown that JuMP can create problems at similar speeds to special-purpose modeling languages such as AMPL.
  - JuMP communicates with most solvers in memory, avoiding the need to write intermediary files.
- Solver independence
  - JuMP uses a generic solver-independent interface provided by the MathOptInterface package, making it easy to change between a number of open-source and commercial optimization software packages ("solvers"). The Supported solvers section contains a table of the currently supported solvers.
- Access to advanced algorithmic techniques
  - Efficient in-memory LP re-solves which previously required using solver-specific and/or low-level C++ libraries.
  - Access to solver-independent and solver-dependent Callbacks.
- Ease of embedding
  - JuMP itself is written purely in Julia. Solvers are the only binary dependencies.
  - Automated install of many solver dependencies.

    * JuMP provides automatic installation of many open-source solvers. This is different to modeling languages in Python which require you to download and install a solver yourself.
- Being embedded in a general-purpose programming language makes it easy to solve optimization problems as part of a larger workflow (e.g., inside a simulation, behind a web server, or as a subproblem in a decomposition algorithm).
    * As a trade-off, JuMP's syntax is constrained by the syntax available in Julia.
- JuMP is MPL licensed, meaning that it can be embedded in commercial software that complies with the terms of the license.

28.2 When should I not use JuMP?

JuMP supports a broad range of optimization classes. However, there are still some that it doesn't support, or that are better supported by other software packages.

Black-box, derivative free, or unconstrained optimization

JuMP does support nonlinear programs with constraints and objectives containing user-defined functions. However, the functions must be automatically differentiable, or need to provide explicit derivatives. (See User-defined Functions for more information.)

If your function is a black-box that is non-differentiable (e.g., the output of a simulation written in C++), JuMP is not the right tool for the job. This also applies if you want to use a derivative free method.

Even if your problem is differentiable, if it is unconstrained there is limited benefit (and downsides in the form of more overhead) to using JuMP over tools which are only concerned with function minimization.

Alternatives to consider are:

- Optim.jl
- NLopt.jl

Multiobjective programs

If your problem has more than one objective, JuMP is not the right tool for the job. However, we're working on fixing this!.

Alternatives to consider are:

- vOptGeneric.jl

Disciplined convex programming

JuMP does not support disciplined convex programming (DCP).

Alternatives to consider are:

- Convex.jl

Note

`Convex.jl` is also built on MathOptInterface, and shares the same set of underlying solvers. However, you input problems differently, and Convex.jl checks that the problem is DCP.

Chapter 29

Algebraic modeling languages

29.1 What is an algebraic modeling language?

If you have taken a class in mixed-integer linear programming, you will have seen a formulation like:

$$\begin{aligned} \min\ & c^\top x \\ \text{s.t.}\ & Ax = b \\ & x \geq 0 \\ & x_i \in \mathbb{Z}, \quad \forall i \in \mathcal{I} \end{aligned}$$

where c, A, and b are appropriately sized vectors and matrices of data, and $\mathcal{I}$ denotes the set of variables that are integer.

Solvers expect problems in a standard form like this because it limits the types of constraints that they need to consider. This makes writing a solver much easier.

> **What is a solver?**
>
> A solver is a software package that computes solutions to one or more classes of problems.
>
> For example, GLPK is a solver for linear programming (LP) and mixed integer programming (MIP) problems. It incorporates algorithms such as the simplex method and the interior-point method.
>
> JuMP currently supports a number of open-source and commercial solvers, which can be viewed in the Supported-solvers table.

However, you probably formulated problems algebraically like so:

$$\begin{aligned} \min\ & \sum_{i=1}^{n} c_i x_i \\ \text{s.t.}\ & \sum_{i=1}^{n} w_i x_i \leq b \\ & x_i \geq 0 \quad \forall i = 1, \ldots, n \\ & x_i \in \mathbb{Z} \quad \forall i = 1, \ldots, n. \end{aligned}$$

Info

Do you recognize this formulation? It's the knapsack problem.

Users prefer to write problems in algebraic form because it is more convenient. For example, we just used $\leq b$, even though the standard form only supported constraints of the form $Ax = b$.

We could convert our knapsack problem into the standard form by adding a new slack variable x_0 like so:

$$\begin{aligned} \min \quad & \sum_{i=1}^{n} c_i x_i \\ \text{s.t.} \quad & x_0 + \sum_{i=1}^{n} w_i x_i = b \\ & x_i \geq 0 \quad \forall i = 0, \ldots, n \\ & x_i \in \mathbb{Z} \quad \forall i = 1, \ldots, n. \end{aligned}$$

However, as models get more complicated, this manual conversion becomes more and more error-prone.

An algebraic modeling language is a tool that simplifies the translation between the algebraic form of the modeler, and the standard form of the solver.

Each algebraic modeling language has two main parts:

1. A domain specific language for the user to write down problems in algebraic form.
2. A converter from the algebraic form into a standard form supported by the solver (and back again).

29.2 Part I: writing in algebraic form

JuMP provides the first part of an algebraic modeling language using the `@variable`, `@objective`, and `@constraint` macros.

For example, here's how we would write the knapsack problem in JuMP:

```
julia> function algebraic_knapsack(c, w, b)
           n = length(c)
           model = Model()
           @variable(model, x[1:n] >= 0, Int)
           @objective(model, Min, sum(c[i] * x[i] for i = 1:n))
           @constraint(model, sum(w[i] * x[i] for i = 1:n) <= b)
           return print(model)
       end
algebraic_knapsack (generic function with 1 method)

julia> algebraic_knapsack([1, 2], [0.5, 0.5], 1.25)
Min x[1] + 2 x[2]
Subject to
 0.5 x[1] + 0.5 x[2] ≤ 1.25
 x[1] ≥ 0.0
 x[2] ≥ 0.0
 x[1] integer
 x[2] integer
```

This formulation is compact, and it closely matches the algebraic formulation of the model we wrote out above.

Here's what the JuMP code would look like if we didn't use macros:

```
julia> function nonalgebraic_knapsack(c, w, b)
           n = length(c)
           model = Model()
           x = [VariableRef(model) for i = 1:n]
           for i = 1:n
               set_lower_bound(x[i], 0)
               set_integer(x[i])
               set_name(x[i], "x[$i]")
           end
           obj = AffExpr(0.0)
           for i = 1:n
               add_to_expression!(obj, c[i], x[i])
           end
           set_objective(model, MOI.MIN_SENSE, obj)
           lhs = AffExpr(0.0)
           for i = 1:n
               add_to_expression!(lhs, w[i], x[i])
           end
           con = build_constraint(error, lhs, MOI.LessThan(b))
           add_constraint(model, con)
           return print(model)
       end
nonalgebraic_knapsack (generic function with 1 method)

julia> nonalgebraic_knapsack([1, 2], [0.5, 0.5], 1.25)
Min x[1] + 2 x[2]
Subject to
 0.5 x[1] + 0.5 x[2] ≤ 1.25
 x[1] ≥ 0.0
 x[2] ≥ 0.0
 x[1] integer
 x[2] integer
```

Hopefully you agree that the macro version is much easier to read!

29.3 Part II: talking to solvers

Now that we have the algebraic problem from the user, we need a way of communicating the problem to the solver, and a way of returning the solution from the solver back to the user.

This is less trivial than it might seem, because each solver has a unique application programming interface (API) and data structures for representing optimization models and obtaining results.

JuMP uses the MathOptInterface.jl package to abstract these differences between solvers.

What is MathOptInterface?

MathOptInterface (MOI) is an abstraction layer designed to provide an interface to mathematical optimization solvers so that users do not need to understand multiple solver-specific APIs. MOI can be used directly, or through a higher-level modeling interface like JuMP.

There are three main parts to MathOptInterface:

1. A solver-independent API that abstracts concepts such as adding and deleting variables and constraints, setting and getting parameters, and querying results. For more information on the MathOptInterface API, read the documentation.

2. An automatic rewriting system based on equivalent formulations of a constraint. For more information on this rewriting system, read the LazyBridgeOptimizer section of the manual, and our paper on arXiv.

3. Utilities for managing how and when models are copied to solvers. For more information on this, read the CachingOptimizer section of the manual.

Part VI

Developer Docs

Chapter 30

Contributing

30.1 How to contribute to JuMP

Welcome! This document explains some of the ways you can contribute to JuMP.

Code of Conduct

This project and everyone participating in it is governed by the JuMP Code of Conduct. By participating, you are expected to uphold this code.

Join the community forum

First up, join the community forum.

The forum is a good place to ask questions about how to use JuMP. You can also use the forum to discuss possible feature requests and bugs before raising a GitHub issue (more on this below).

Aside from asking questions, the easiest way you can contribute to JuMP is to help answer questions on the forum!

Join the developer chatroom

If you're interested in contributing code to JuMP, the next place to join is the developer chatroom. Let us know what you have in mind, and we can point you in the right direction.

Improve the documentation

Chances are, if you asked (or answered) a question on the community forum, then it is a sign that the documentation could be improved. Moreover, since it is your question, you are probably the best-placed person to improve it!

The docs are written in Markdown and are built using Documenter.jl. You can find the source of all the docs here.

If your change is small (like fixing typos, or one or two sentence corrections), the easiest way to do this is via GitHub's online editor. (GitHub has help on how to do this.)

If your change is larger, or touches multiple files, you will need to make the change locally and then use Git to submit a pull request. (See Contribute code to JuMP below for more on this.)

> **Tip**
>
> If you need any help, come join the developer chatroom and we will walk you through the process.

File a bug report

Another way to contribute to JuMP is to file bug reports.

Make sure you read the info in the box where you write the body of the issue before posting. You can also find a copy of that info here.

> **Tip**
>
> If you're unsure whether you have a real bug, post on the community forum first. Someone will either help you fix the problem, or let you know the most appropriate place to open a bug report.

Contribute code to JuMP

Finally, you can also contribute code to JuMP!

> **Warning**
>
> If you do not have experience with Git, GitHub, and Julia development, the first steps can be a little daunting. However, there are lots of tutorials available online, including these for:
>
> - GitHub
> - Git and GitHub
> - Git
> - Julia package development
>
> If you need any help, come join the developer chatroom and we will walk you through the process.

Once you are familiar with Git and GitHub, the workflow for contributing code to JuMP is along the lines of the following:

Step 1: decide what to work on

The first step is to find an open issue (or open a new one) for the problem you want to solve. Then, before spending too much time on it, discuss what you are planning to do in the issue to see if other contributors are fine with your proposed changes. Getting feedback early can improve code quality, and avoid time spent writing code that does not get merged into JuMP.

> **Tip**
>
> At this point, remember to be patient and polite; you may get a lot of comments on your issue! However, do not be afraid! Comments mean that people are willing to help you improve the code that you are contributing to JuMP.

Step 2: fork JuMP

Go to https://github.com/jump-dev/JuMP.jl and click the "Fork" button in the top-right corner. This will create a copy of JuMP under your GitHub account.

Step 3: install JuMP locally

Open Julia and run:

```
] dev JuMP
```

This will download the JuMP Git repository to `~/.julia/dev/JuMP`. If you're on Windows, this will be `C:\\Users\\<my_name>\\.jul`

> **Warning**
>
> `]` command means "first type `]` to enter the Julia pkg mode, then type the rest. Don't copy-paste the code directly.

Step 4: checkout a new branch

> **Note**
>
> In the following, replace any instance of `GITHUB_ACCOUNT` with your GitHub user name.

The next step is to checkout a development branch. In a terminal (or command prompt on Windows), run:

```
$ cd ~/.julia/dev/JuMP

$ git remote add GITHUB_ACCOUNT https://github.com/GITHUB_ACCOUNT/JuMP.jl.git

$ git checkout master

$ git pull

$ git checkout -b my_new_branch
```

> **Tip**
>
> Lines starting with $ mean "run these in a terminal (command prompt on Windows)."

Step 5: make changes

Now make any changes to the source code inside the `~/.julia/dev/JuMP` directory.

Make sure you:

- Follow the Style guide
- Add tests and documentation for any changes or new features

> **Tip**
>
> When you change the source code, you'll need to restart Julia for the changes to take effect. This is a pain, so install Revise.jl.

Step 6a: test your code changes

To test that your changes work, run the JuMP test-suite by opening Julia and running:

```
cd("~/.julia/dev/JuMP")
] activate .
] test
```

> **Warning**
>
> Running the tests might take a long time (~10-15 minutes).

> **Tip**
>
> If you're using Revise.jl, you can also run the tests by calling `include`:
>
> ```
> include("test/runtests.jl")
> ```
>
> This can be faster if you want to re-run the tests multiple times.

Step 6b: test your documentation changes

Open Julia, then run:

```
cd("~/.julia/dev/JuMP/docs")
] activate .
include("src/make.jl")
```

> **Warning**
>
> Building the documentation might take a long time (~10 minutes).

> **Tip**
>
> If there's a problem with the tests that you don't know how to fix, don't worry. Continue on to step 5, and one of the JuMP contributors will comment on your pull request telling you how to fix things.

Step 7: make a pull request

Once you've made changes, you're ready to push the changes to GitHub. Run:

```
$ cd ~/.julia/dev/JuMP

$ git add .

$ git commit -m "A descriptive message of the changes"

$ git push -u GITHUB_ACCOUNT my_new_branch
```

Then go to https://github.com/jump-dev/JuMP.jl and follow the instructions that pop up to open a pull request.

Step 8: respond to comments

At this point, remember to be patient and polite; you may get a lot of comments on your pull request! However, do not be afraid! A lot of comments means that people are willing to help you improve the code that you are contributing to JuMP.

To respond to the comments, go back to step 5, make any changes, test the changes in step 6, and then make a new commit in step 7. Your PR will automatically update.

Step 9: cleaning up

Once the PR is merged, you should clean-up your Git repository ready for the next contribution!

```
$ cd ~/.julia/dev/JuMP

$ git checkout master

$ git pull
```

Note

If you have suggestions to improve this guide, please make a pull request! It's particularly helpful if you do this after your first pull request because you'll know all the parts that could be explained better.

Thanks for contributing to JuMP!

Chapter 31

Extensions

31.1 Extensions

JuMP provides a variety of ways to extend the basic modeling functionality.

> **Tip**
>
> This documentation in this section is still a work-in-progress. The best place to look for ideas and help when writing a new JuMP extension are existing JuMP extensions. Examples include:
>
> - BilevelJuMP.jl
> - Coluna.jl
> - InfiniteOpt.jl
> - Plasmo.jl
> - PolyJuMP.jl
> - SDDP.jl
> - StochasticPrograms.jl
> - SumOfSquares.jl
> - vOptGeneric.jl

Define a new set

To define a new set for JuMP, subtype `MOI.AbstractScalarSet` or `MOI.AbstractVectorSet` and implement `Base.copy` for the set. That's it!

```
struct _NewVectorSet <: MOI.AbstractVectorSet
    dimension::Int
end
Base.copy(x::_NewVectorSet) = x

model = Model()
@variable(model, x[1:2])
@constraint(model, x in _NewVectorSet(2))

# output

[x[1], x[2]] ∈ _NewVectorSet(2)
```

However, for vector-sets, this requires the user to specify the dimension argument to their set, even though we could infer it from the length of x!

You can make a more user-friendly set by subtyping `AbstractVectorSet` and implementing `moi_set`.

```
struct NewVectorSet <: JuMP.AbstractVectorSet end
JuMP.moi_set(::NewVectorSet, dim::Int) = _NewVectorSet(dim)

model = Model()
@variable(model, x[1:2])
@constraint(model, x in NewVectorSet())

# output

[x[1], x[2]] ∈ _NewVectorSet(2)
```

Extend `@variable`

Just as `Bin` and `Int` create binary and integer variables, you can extend the `@variable` macro to create new types of variables. Here is an explanation by example, where we create a `AddTwice` type, that creates a tuple of two JuMP variables instead of a single variable.

First, create a new struct. This can be anything. Our struct holds a `VariableInfo` object that stores bound information, and whether the variable is binary or integer.

```
julia> struct AddTwice
           info::JuMP.VariableInfo
       end
```

Second, implement `build_variable`, which takes `::Type{AddTwice}` as an argument, and returns an instance of `AddTwice`. Note that you can also receive keyword arguments.

```
julia> function JuMP.build_variable(
           _err::Function,
           info::JuMP.VariableInfo,
           ::Type{AddTwice};
           kwargs...
       )
           println("Can also use $kwargs here.")
           return AddTwice(info)
       end
```

Third, implement `add_variable`, which takes the instance of `AddTwice` from the previous step, and returns something. Typically, you will want to call `add_variable` here. For example, our `AddTwice` call is going to add two JuMP variables.

```
julia> function JuMP.add_variable(
           model::JuMP.Model,
           duplicate::AddTwice,
           name::String,
       )
           a = JuMP.add_variable(
               model,
```

```
            JuMP.ScalarVariable(duplicate.info),
            name * "_a",
        )
        b = JuMP.add_variable(
            model,
            JuMP.ScalarVariable(duplicate.info),
            name * "_b",
        )
        return (a, b)
    end
```

Now `AddTwice` can be passed to `@variable` just like `Bin` or `Int`. However, now it adds two variables instead of one!

```
julia> model = Model();

julia> @variable(model, x[i=1:2], AddTwice, kw=i)
Can also use Base.Iterators.Pairs(:kw => 1) here.
Can also use Base.Iterators.Pairs(:kw => 2) here.
2-element Vector{Tuple{VariableRef, VariableRef}}:
 (x[1]_a, x[1]_b)
 (x[2]_a, x[2]_b)

julia> num_variables(model)
4

julia> first(x[1])
x[1]_a

julia> last(x[2])
x[2]_b
```

Extend `@constraint`

The `@constraint` macro has three steps that can be intercepted and extended: parse time, build time, and add time.

Parse

To extend the `@constraint` macro at parse time, implement one of the following methods:

- `parse_constraint_head`
- `parse_constraint_call`

> **Warning**
>
> Extending the constraint macro at parse time is an advanced operation and has the potential to interfere with existing JuMP syntax. Please discuss with the developer chatroom before publishing any code that implements these methods.

`parse_constraint_head` should be implemented to intercept an expression based on the `.head` field of `Base.Expr`. For example:

```
julia> using JuMP

julia> const MutableArithmetics = JuMP._MA;

julia> model = Model(); @variable(model, x);

julia> function JuMP.parse_constraint_head(
           _error::Function,
           ::Val{:(:=)},
           lhs,
           rhs,
       )
           println("Rewriting := as ==")
           new_lhs, parse_code = MutableArithmetics.rewrite(lhs)
           build_code = :(
               build_constraint($(_error), $(new_lhs), MOI.EqualTo($(rhs)))
           )
           return false, parse_code, build_code
       end

julia> @constraint(model, x + x := 1.0)
Rewriting := as ==
2 x = 1.0
```

parse_constraint_call should be implemented to intercept an expression of the form `Expr(:call, op, args...)`. For example:

```
julia> using JuMP

julia> const MutableArithmetics = JuMP._MA;

julia> model = Model(); @variable(model, x);

julia> function JuMP.parse_constraint_call(
           _error::Function,
           is_vectorized::Bool,
           ::Val{:my_equal_to},
           lhs,
           rhs,
       )
           println("Rewriting my_equal_to to ==")
           new_lhs, parse_code = MutableArithmetics.rewrite(lhs)
           build_code = if is_vectorized
               :(build_constraint($(_error), $(new_lhs), MOI.EqualTo($(rhs))))
           else
               :(build_constraint.($(_error), $(new_lhs), MOI.EqualTo($(rhs))))
           end
           return parse_code, build_code
       end

julia> @constraint(model, my_equal_to(x + x, 1.0))
Rewriting my_equal_to to ==
2 x = 1.0
```

Tip

When parsing a constraint you can recurse into sub-constraint (e.g., the `{expr}` in `z => {x <= 1}`) by calling `parse_constraint`.

Build

To extend the `@constraint` macro at build time, implement a new `build_constraint` method.

This may mean implementing a method for a specific function or set created at parse time, or it may mean implementing a method which handles additional positional arguments.

`build_constraint` must return an `AbstractConstraint`, which can either be an `AbstractConstraint` already supported by JuMP, e.g., `ScalarConstraint` or `VectorConstraint`, or a custom `AbstractConstraint` with a corresponding `add_constraint` method (see Add).

Tip

The easiest way to extend `@constraint` is via an additional positional argument to `build_constraint`.

Here is an example of adding extra arguments to `build_constraint`:

```
julia> model = Model(); @variable(model, x);

julia> struct MyConstrType end

julia> function JuMP.build_constraint(
            _error::Function,
            f::JuMP.GenericAffExpr,
            set::MOI.EqualTo,
            extra::Type{MyConstrType};
            d = 0,
       )
            new_set = MOI.LessThan(set.value + d)
            return JuMP.build_constraint(_error, f, new_set)
       end

julia> @constraint(model, my_con, x == 0, MyConstrType, d = 2)
my_con : x ≤ 2.0
```

Note

Only a single positional argument can be given to a particular constraint. Extensions that seek to pass multiple arguments (e.g., `Foo` and `Bar`) should combine them into one argument type (e.g., `FooBar`).

Add

`build_constraint` returns an `AbstractConstraint` object. To extend `@constraint` at add time, define a subtype of `AbstractConstraint`, implement `build_constraint` to return an instance of the new type, and then implement `add_constraint`.

Here is an example:

```
julia> model = Model(); @variable(model, x);

julia> struct MyTag
```

```
            name::String
        end

julia> struct MyConstraint{S} <: AbstractConstraint
            name::String
            f::AffExpr
            s::S
        end

julia> function JuMP.build_constraint(
             _error::Function,
             f::AffExpr,
             set::MOI.AbstractScalarSet,
             extra::MyTag,
        )
             return MyConstraint(extra.name, f, set)
        end

julia> function JuMP.add_constraint(
             model::Model,
             con::MyConstraint,
             name::String,
        )
             return add_constraint(
                 model,
                 ScalarConstraint(con.f, con.s),
                 "$(con.name)[$(name)]",
             )
        end

julia> @constraint(model, my_con, 2x <= 1, MyTag("my_prefix"))
my_prefix[my_con] : 2 x - 1 ≤ 0.0
```

The extension dictionary

Every JuMP model has a field `.ext::Dict{Symbol,Any}` that can be used by extensions. This is useful if your extensions to `@variable` and `@constraint` need to store information between calls.

The most common way to initialize a model with information in the `.ext` dictionary is to provide a new constructor:

```
julia> function MyModel()
            model = Model()
            model.ext[:MyModel] = 1
            return model
        end
MyModel (generic function with 1 method)

julia> model = MyModel()
A JuMP Model
Feasibility problem with:
Variables: 0
Model mode: AUTOMATIC
CachingOptimizer state: NO_OPTIMIZER
Solver name: No optimizer attached.
```

```
julia> model.ext
Dict{Symbol, Any} with 1 entry:
  :MyModel => 1
```

Defining new JuMP models

If extending individual calls to `@variable` and `@constraint` is not sufficient, it is possible to implement a new model via a subtype of `AbstractModel`. You can also define new `AbstractVariableRef`s to create different types of JuMP variables.

> **Warning**
>
> Extending JuMP in this manner is an advanced operation. We strongly encourage you to consider how you can use the methods mentioned in the previous sections to achieve your aims instead of defining new model and variable types. Consult the developer chatroom before starting work on this.

If you define new types, you will need to implement a considerable number of methods, and doing so will require a detailed understanding of the JuMP internals. Therefore, the list of methods to implement is currently undocumented.

The easiest way to extend JuMP by defining a new model type is to follow an existing example. A simple example to follow is the JuMPExtension module in the JuMP test suite. The best example of an external JuMP extension that implements an `AbstractModel` is InfiniteOpt.jl.

Chapter 32

Style Guide

32.1 Style guide and design principles

Style guide

This section describes the coding style rules that apply to JuMP code and that we recommend for JuMP models and surrounding Julia code. The motivations for a style guide include:

- conveying best practices for writing readable and maintainable code
- reducing the amount of time spent on bike-shedding by establishing basic naming and formatting conventions
- lowering the barrier for new contributors by codifying the existing practices (e.g., you can be more confident your code will pass review if you follow the style guide)

In some cases, the JuMP style guide diverges from the Julia style guide. All such cases will be explicitly noted and justified.

The JuMP style guide adopts many recommendations from the Google style guides.

> **Info**
>
> The style guide is always a work in progress, and not all JuMP code follows the rules. When modifying JuMP, please fix the style violations of the surrounding code (i.e., leave the code tidier than when you started). If large changes are needed, consider separating them into another PR.

JuliaFormatter

JuMP uses JuliaFormatter.jl as an autoformatting tool.

We use the options contained in `.JuliaFormatter.toml`.

To format code, `cd` to the JuMP directory, then run:

```
] add JuliaFormatter@0.13.2
using JuliaFormatter
format("src")
format("test")
```

Info

A continuous integration check verifies that all PRs made to JuMP have passed the formatter.

The following sections outline extra style guide points that are not fixed automatically by JuliaFormatter.

Whitespace

For conciseness, never use more than one blank line within a function, and never begin a function with a blank line.

Bad:

```
function foo(x)
    y = 2 * x


    return y
end

function foo(x)

    y = 2 * x
    return y
end
```

Juxtaposed multiplication

Only use juxtaposed multiplication when the right-hand side is a symbol.

Good:

```
2x  # Acceptable if there are space constraints.
2 * x  # This is preferred if space is not an issue.
2 * (x + 1)
```

Bad:

```
2(x + 1)
```

Empty vectors

For a type `T`, `T[]` and `Vector{T}()` are equivalent ways to create an empty vector with element type `T`. Prefer `T[]` because it is more concise.

Comments

For non-native speakers and for general clarity, comments in code must be proper English sentences with appropriate punctuation.

Good:

```
# This is a comment demonstrating a good comment.
```

Bad:

```
# a bad comment
```

JuMP macro syntax

For consistency, always use parentheses.

Good:

```
@variable(model, x >= 0)
```

Bad:

```
@variable model x >= 0
```

For consistency, always use `constant * variable` as opposed to `variable * constant`. This makes it easier to read models in ambiguous cases like `a * x`.

Good:

```
a = 4
@constraint(model, 3 * x <= 1)
@constraint(model, a * x <= 1)
```

Bad:

```
a = 4
@constraint(model, x * 3 <= 1)
@constraint(model, x * a <= 1)
```

In order to reduce boilerplate code, prefer the plural form of macros over lots of repeated calls to singular forms.

Good:

```
@variables(model, begin
    x >= 0
    y >= 1
    z <= 2
end)
```

Bad:

```
@variable(model, x >= 0)
@variable(model, y >= 1)
@variable(model, z <= 2)
```

An exception is made for calls with many keyword arguments, since these need to be enclosed in parentheses in order to parse properly.

Acceptable:

```
@variable(model, x >= 0, start = 0.0, base_name = "my_x")
@variable(model, y >= 1, start = 2.0)
@variable(model, z <= 2, start = -1.0)
```

Also acceptable:

```
@variables(model, begin
    x >= 0, (start = 0.0, base_name = "my_x")
    y >= 1, (start = 2.0)
    z <= 2, (start = -1.0)
end)
```

While we always use `in` for `for`-loops, it is acceptable to use = in the container declarations of JuMP macros.

Okay:

```
@variable(model, x[i=1:3])
```

Also okay:

```
@variable(model, x[i in 1:3])
```

Naming

```
module SomeModule end
function some_function end
const SOME_CONSTANT = ...
struct SomeStruct
  some_field::SomeType
end
@enum SomeEnum ENUM_VALUE_A ENUM_VALUE_B
some_local_variable = ...
some_file.jl # Except for ModuleName.jl.
```

Exported and non-exported names

Begin private module level functions and constants with an underscore. All other objects in the scope of a module should be exported. (See JuMP.jl for an example of how to do this.)

Names beginning with an underscore should only be used for distinguishing between exported (public) and non-exported (private) objects. Therefore, never begin the name of a local variable with an underscore.

```
module MyModule

export public_function, PUBLIC_CONSTANT

function _private_function()
    local_variable = 1
    return
end

function public_function end

const _PRIVATE_CONSTANT = 3.14159
const PUBLIC_CONSTANT = 1.41421

end
```

Use of underscores within names

The Julia style guide recommends avoiding underscores "when readable", for example, `haskey`, `isequal`, `remotecall`, and `remotecall_fetch`. This convention creates the potential for unnecessary bikeshedding and also forces the user to recall the presence/absence of an underscore, e.g., "was that argument named `basename` or `base_name`?". For consistency, always use underscores in variable names and function names to separate words.

Use of `!`

Julia has a convention of appending `!` to a function name if the function modifies its arguments. We recommend to:

- Omit `!` when the name itself makes it clear that modification is taking place, e.g., `add_constraint` and `set_name`. We depart from the Julia style guide because `!` does not provide a reader with any additional information in this case, and adherence to this convention is not uniform even in base Julia itself (consider `Base.println` and `Base.finalize`).
- Use `!` in all other cases. In particular it can be used to distinguish between modifying and non-modifying variants of the same function like `scale` and `scale!`.

Note that `!` is not a self-documenting feature because it is still ambiguous which arguments are modified when multiple arguments are present. Be sure to document which arguments are modified in the method's docstring.

See also the Julia style guide recommendations for ordering of function arguments.

Abbreviations

Abbreviate names to make the code more readable, not to save typing. Don't arbitrarily delete letters from a word to abbreviate it (e.g., `indx`). Use abbreviations consistently within a body of code (e.g., do not mix `con` and `constr`, `idx` and `indx`).

Common abbreviations:

- `num` for `number`
- `con` for `constraint`

No one-letter variable names

Where possible, avoid one-letter variable names.

Use `model = Model()` instead of `m = Model()`

Exceptions are made for indices in loops.

User-facing `MethodError`

Specifying argument types for methods is mostly optional in Julia, which means that it's possible to find out that you are working with unexpected types deep in the call chain. Avoid this situation or handle it with a helpful error message. A user should see a `MethodError` only for methods that they called directly.

Bad:

```
_internal_function(x::Integer) = x + 1
# The user sees a MethodError for _internal_function when calling
# public_function("a string"). This is not very helpful.
public_function(x) = _internal_function(x)
```

Good:

```
_internal_function(x::Integer) = x + 1
# The user sees a MethodError for public_function when calling
# public_function("a string"). This is easy to understand.
public_function(x::Integer) = _internal_function(x)
```

If it is hard to provide an error message at the top of the call chain, then the following pattern is also ok:

```
_internal_function(x::Integer) = x + 1
function _internal_function(x)
    error(
        "Internal error. This probably means that you called " *
        "public_function() with the wrong type.",
    )
end
public_function(x) = _internal_function(x)
```

`@enum` vs. `Symbol`

The `@enum` macro lets you define types with a finite number of values that are explicitly enumerated (like `enum` in C/C++). `Symbol`s are lightweight strings that are used to represent identifiers in Julia (for example, `:x`).

`@enum` provides type safety and can have docstrings attached to explain the possible values. Use `@enum`s when applicable, e.g., for reporting statuses. Use strings to provide long-form additional information like error messages.

Use of `Symbol` should typically be reserved for identifiers, e.g., for lookup in the JuMP model (`model[:my_variable]`).

`using` vs. `import`

`using ModuleName` brings all symbols exported by the module `ModuleName` into scope, while `import ModuleName` brings only the module itself into scope. (See the Julia manual) for examples and more details.

For the same reason that `from <module> import *` is not recommended in python (PEP 8), avoid `using ModuleName` except in throw-away scripts or at the REPL. The `using` statement makes it harder to track where symbols come from and exposes the code to ambiguities when two modules export the same symbol.

Prefer `using ModuleName: x, p` to `import ModuleName.x, ModuleName.p` and `import MyModule: x, p` because the `import` versions allow method extension without qualifying with the module name.

Similarly, `using ModuleName: ModuleName` is an acceptable substitute for `import ModuleName`, because it does not bring all symbols exported by `ModuleName` into scope. However, we prefer `import ModuleName` for consistency.

Documentation

This section describes the writing style that should be used when writing documentation for JuMP (and supporting packages).

We can recommend the documentation style guides by Divio, Google, and Write the Docs as general reading for those writing documentation. This guide delegates a thorough handling of the topic to those guides and instead elaborates on the points more specific to Julia and documentation that use Documenter.

- Be concise
- Use lists instead of long sentences
- Use numbered lists when describing a sequence, e.g., (1) do X, (2) then Y
- Use bullet points when the items are not ordered
- Example code should be covered by doctests
- When a word is a Julia symbol and not an English word, enclose it with backticks. In addition, if it has a docstring in this doc add a link using `@ref`. If it is a plural, add the "s" after the closing backtick. For example,

  ```
  [`VariableRef`](@ref)s
  ```

- Use `@meta` blocks for TODOs and other comments that shouldn't be visible to readers. For example,

  ````
  ```@meta
  # TODO: Mention also X, Y, and Z.
  ```
  ````

Docstrings

- Every exported object needs a docstring
- All examples in docstrings should be `jldoctests`
- Always use complete English sentences with proper punctuation
- Do not terminate lists with punctuation (e.g., as in this doc)

Here is an example:

````
"""
    signature(args; kwargs...)

Short sentence describing the function.

Optional: add a slightly longer paragraph describing the function.

## Notes

 - List any notes that the user should be aware of

## Examples

```jldoctest
julia> 1 + 1
2
```
"""
````

Testing

Use a module to encapsulate tests, and structure all tests as functions. This avoids leaking local variables between tests.

Here is a basic skeleton:

```
module TestPkg

using Test

_helper_function() = 2

function test_addition()
    @test 1 + 1 == _helper_function()
end

function runtests()
    for name in names(@__MODULE__; all = true)
        if startswith("$(name)", "test_")
            @testset "$(name)" begin
                getfield(@__MODULE__, name)()
            end
        end
    end
end

end # TestPkg

TestPkg.runtests()
```

Break the tests into multiple files, with one module per file, so that subsets of the codebase can be tested by calling `include` with the relevant file.

Design principles

TODO: How to structure and test large JuMP models, libraries that use JuMP.

For how to write a solver, see MOI.

Chapter 33

Roadmap

33.1 Development roadmap

This page is not JuMP documentation per se but are notes for the JuMP community. The JuMP developers have compiled this roadmap document to share their plans and goals. Contributions to roadmap issues are especially invited.

JuMP 1.0

JuMP 1.0 will be ready to release roughly when all of these tasks are completed. Some but not all of these tasks are summarized in the JuMP 1.0 milestone.

- Create a website for JuMP (**Done**: jump.dev)
- Deprecate the JuliaOpt organization and move repositories to the JuMP-dev organization (**Done**)
- Address major regressions from JuMP 0.18
  - Performance (#1403, #1654, #1607)
  - Callbacks (**Done**: see `examples/callbacks.jl`)
  - Column generation syntax (**Done**: see `examples/cutting_stock_column_generation.jl`)
  - Support for second-order cones in Gurobi, CPLEX, and Xpress (**Done**)
- Fix issues that we promised MOI would fix
  - Checking feasibility of solutions (**Done**: #2466)
  - Accessing IIS (**Done**: see Conflicts)
  - Accessing multiple results from solvers (**Done**: Gurobi#392)
  - Dual warm-starts (**Done**: #2214)
- Address "easy" usability issues
  - Line numbers in error messages (**Done**: #2276)
  - LP sensitivity summary (**Done**: see Sensitivity analysis for LP)
  - Inferred element types for collections in macros (**Done**: #2070)
  - Expose solver-independent options from JuMP (**Done**: see `set_silent` etc.)

- Improve the documentation (#1062)
  - Separate how-to, concept explanation, and technical reference following the Divio recommendations (**Done**)
  - Fully integrate JuMPTutorials with JuMP's documentation (**Done**)
- Developer experience
  - Get JuMP's unit tests running faster. See #1745. (**Done**)
- All solvers should complete the transition to MOI (**Done**)
- Provide packages for installing Bonmin and Couenne (**Done**)
- MathOptFormat 1.0 (**Done**)

MOI 1.0

Beyond JuMP 1.0

Part VII

Release Notes

Chapter 34

Release notes

34.1 Version 0.22.0 (In development)

JuMP v0.22 is a breaking release

Breaking changes

JuMP 0.22 contains a number of breaking changes. However, these should be invisible for the majority of users. You will mostly encounter these breaking changes if you: wrote a JuMP extension, accessed `backend(model)`, or called `@SDconstraint`.

The breaking changes are as follows:

- MathOptInterface has been updated to v0.10.3. For users who have interacted with the MOI backend, this contains a large number of breaking changes. Read the MathOptInterface release notes for more details.
- The `bridge_constraints` keyword argument to `Model` and `set_optimizer` has been renamed `add_bridges` to reflect that more thing were bridged than just constraints.
- The `backend(model)` field now contains a concrete instance of a `MOI.Utilities.CachingOptimizer` instead of one with an abstractly typed optimizer field. In most cases, this will lead to improved performance. However, calling `set_optimizer` after `backend` invalidates the old backend. For example:

```
model = Model()
b = backend(model)
set_optimizer(model, GLPK.Optimizer)
@variable(model, x)
# b is not updated with `x`! Get a new b by calling `backend` again.
new_b = backend(model)
```

- All usages of `@SDconstraint` are deprecated. The new syntax is `@constraint(model, X >= Y, PSDCone())`.
- Creating a `DenseAxisArray` with a `Number` as an axis will now display a warning. This catches a common error in which users write `@variable(model, x[length(S)])` instead of `@variable(model, x[1:length(S)])`.
- The `caching_mode` argument to `Model`, e.g., `Model(caching_mode = MOIU.MANUAL)` mode has been removed. For more control over the optimizer, use `direct_model` instead.
- The previously deprecated `lp_objective_perturbation_range` and `lp_rhs_perturbation_range` functions have been removed. Use `lp_sensitivity_report` instead.

- The `.m` fields of `NonlinearExpression` and `NonlinearParameter` have been renamed to `.model`.
- Infinite variable bounds are now ignored. Thus, `@variable(model, x <= Inf)` will show `has_upper_bound(x) == false`. Previously, these bounds were passed through to the solvers which caused numerical issues for solvers expecting finite bounds.
- The `variable_type` and `constraint_type` functions were removed. This should only affect users who previously wrote JuMP extensions. The functions can be deleted without consequence.
- The internal functions `moi_mode`, `moi_bridge_constraints`, `moi_add_constraint`, and `moi_add_to_function_constant` are no longer exported.
- The un-used method `Containers.generate_container` has been deleted.
- The `Containers` API has been refactored, and `_build_ref_sets` is now public as `Containers.build_ref_sets`.
- The `parse_constraint_` methods for extending `@constraint` at parse time have been refactored in a breaking way. Consult the Extensions documentation for more details and examples.

New features

- Copy a `x::DenseAxisArray` to an `Array` by calling `Array(x)`.
- `NonlinearExpression` is now a subtype of `AbstractJuMPScalar`
- Constraints such as `@constraint(model, x + 1 in MOI.Integer())` are now supported.
- `primal_feasibility_report` now accepts a function as the first argument.
- Scalar variables `@variable(model, x[1:2] in MOI.Integer())` creates two variables, both of which are constrained to be in the set `MOI.Integer`.
- Conic constraints can now be specified as inequalities under a different partial ordering. So `@constraint(model, x - y in MOI.Nonnegatives())` can now be written as `@constraint(model, x >= y, MOI.Nonnegatives())`.
- Names are now set for vectorized constraints.

Documentation, maintenance and performance

- The documentation now includes a full copy of the MathOptInterface documentation to make it easy to link concepts between the docs. (The MathOptInterface documentation has also been significantly improved.)
- The documentation contains a large number of improvements and clarifications on a range of topics. Thanks to @sshin23, @DilumAluthge, and @jlwether.
- The documentation is now built with Julia 1.6 instead of 1.0.
- Various error messages have been improved to be more readable.
- Fixed a performance issue when `show` was called on a `SparseAxisArray` with a large number of elements.
- Fixed a bug displaying barrier and simplex iterations in `solution_summary`.
- Fixed a bug by implementing `hash` for `DenseAxisArray` and `SparseAxisArray`.
- Names are now only set if the solver supports them. Previously, this prevented solvers such as Ipopt from being used with `direct_model`.

- `MutableArithmetics.Zero` is converted into a `0.0` before being returned to the user. Previously, some calls to `@expression` would return the undocumented `MutableArithmetics.Zero()` object. One example is summing over an empty set `@expression(model, sum(x[i] for i in 1:0))`. You will now get `0.0` instead.
- `AffExpr` and `QuadExpr` can now be used with `== 0` instead of `iszero`. This fixes a number of issues relating to Julia standard libraries such as `LinearAlgebra` and `SparseArrays`.
- Fixed a bug when registering a user-defined function with splatting.

34.2 Version 0.21.10 (September 4, 2021)

For a detailed list of the closed issues and pull requests from this release, see the tag notes. A summary of changes are as follows:

- New features:
  - Add `add_NL_expression`
  - `add_NL_xxx` functions now support `AffExpr` and `QuadExpr` as terms
- Documentation, maintenance and performance:
  - Fix bug in `solution_summary`
  - Fix bug in `relax_integrality`
  - Improve error message in `lp_sensitivity_report`

34.3 Version 0.21.9 (August 1, 2021)

For a detailed list of the closed issues and pull requests from this release, see the tag notes. A summary of changes are as follows:

- New features:
  - Containers now support arbitrary container types by passing the type to the `container` keyword and overloading `Containers.container`.
  - `is_valid` now supports nonlinear constraints
  - Added `unsafe_backend` for querying the inner-most optimizer of a JuMP model.
  - Nonlinear parameters now support the plural `@NLparameters` macro.
  - Containers (e.g., `DenseAxisArray`) can now be used in vector-valued constraints.
- Documentation, maintenance and performance:
  - Various improvements to the documentation.

34.4 Version 0.21.8 (May 8, 2021)

For a detailed list of the closed issues and pull requests from this release, see the tag notes. A summary of changes are as follows:

- New features:
  - The `@constraint` macro is now extendable in the same way as `@variable`.
  - `AffExpr` and `QuadExpr` can now be used in nonlinear macros.
- Bug fixes:
  - Fixed a bug in `lp_sensitivity_report`.
  - Fixed an inference issue when creating empty `SparseAxisArrays`.

34.5 Version 0.21.7 (April 12, 2021)

For a detailed list of the closed issues and pull requests from this release, see the tag notes. A summary of changes are as follows:

- New features:
  - Added `primal_feasibility_report`, which can be used to check whether a primal point satisfies primal feasibility.
  - Added `coefficient`, which returns the coefficient associated with a variable in affine and quadratic expressions.
  - Added `copy_conflict`, which returns the IIS of an infeasible model.
  - Added `solution_summary`, which returns (and prints) a struct containing a summary of the solution.
  - Allow `AbstractVector` in vector constraints instead of just `Vector`.
  - Added `latex_formulation(model)` which returns an object representing the latex formulation of a model. Use `print(latex_formulation(model))` to print the formulation as a string.
  - User-defined functions in nonlinear expressions are now automatically registered to aid quick model prototyping. However, a warning is printed to encourage the manual registration.
  - DenseAxisArray's now support broadcasting over multiple arrays.
  - Container indices can now be iterators of `Base.SizeUnknown`.
- Bug fixes:
  - Fixed bug in `rad2deg` and `deg2rad` in nonlinear expressions.
  - Fixed a MethodError bug in `Containers` when forcing container type.
  - Allow partial slicing of a DenseAxisArray, resolving an issue from 2014!
  - Fixed a bug printing variable names in IJulia.
  - Ending an IJulia cell with `model` now prints a summary of the model (like in the REPL) not the latex formulation. Use `print(model)` to print the latex formulation.
  - Fixed a bug when copying models containing nested arrays.
- Documentation, performance improvements, and general maintenance:

  - Tutorials are now part of the documentation, and more refactoring has taken place.
  - Added JuliaFormatter added as a code formatter.
  - Added some precompilation statements to reduce initial latency.
  - Various improvements to error messages to make them more helpful.
  - Improved performance of `value(::NonlinearExpression)`.
  - Improved performance of `fix(::VariableRef)`.

34.6 Version 0.21.6 (January 29, 2021)

For a detailed list of the closed issues and pull requests from this release, see the tag notes. A summary of changes are as follows:

- New features:
  - Added support for skew symmetric variables via `@variable(model, X[1:2, 1:2] in SkewSymmetricMatrixSpace(`
  - `lp_sensitivity_report` has been added which significantly improves the performance of querying the sensitivity summary of an LP. `lp_objective_perturbation_range` and `lp_rhs_perturbation_range` are deprecated.
  - Dual warm-starts are now supported with `set_dual_start_value` and `dual_start_value`.
  - ∈ (`\in<tab>`) can now be used in macros instead of = or `in`.
  - Use `haskey(model::Model, key::Symbol)` to check if a name key is registered in a model.
  - Added `unregister(model::Model, key::Symbol)` to unregister a name key from `model`.
  - Added `callback_node_status` for use in callbacks.
  - Added `print_bridge_graph` to visualize the bridging graph generated by MathOptInterface.
  - Improved error message for containers with duplicate indices.
- Bug fixes:
  - Various fixes to pass tests on Julia 1.6.
  - Fixed a bug in the printing of nonlinear expressions in IJulia.
  - Fixed a bug when nonlinear expressions are passed to user-defined functions.
  - Some internal functions that were previously exported are now no longer exported.
  - Fixed a bug when relaxing a fixed binary variable.
  - Fixed a `StackOverflowError` that occurred when `SparseAxisArrays` had a large number of elements.
  - Removed an unnecessary type assertion in `list_of_constraint_types`.
  - Fixed a bug when copying models with registered expressions.
- Documentation and general maintenance:
  - The documentation has been significantly overhauled. It now has distinct sections for the manual, API reference, and examples. The existing examples in `/examples` have now been moved to `/docs/src/examples` and rewritten using `Literate.jl`, and they are now included in the documentation.
  - JuliaFormatter has been applied to most of the codebase. This will continue to roll out over time, as we fix upstream issues in the formatter, and will eventually become compulsory.
  - The root cause of a large number of method invalidations has been resolved.
  - We switched continuous integration from Travis and Appveyor to Github Actions.

34.7 Version 0.21.5 (September 18, 2020)

For a detailed list of the closed issues and pull requests from this release, see the tag notes. A summary of changes are as follows:

- Fix deprecation warnings
- Throw `DimensionMismatch` for incompatibly sized functions and sets
- Unify treatment of `keys(x)` on JuMP containers

34.8 Version 0.21.4 (September 14, 2020)

For a detailed list of the closed issues and pull requests from this release, see the tag notes. A summary of changes are as follows:

- New features:
  - Add debug info when adding unsupported constraints
  - Add `relax_integrality` for solving continuous relaxation
  - Allow querying constraint conflicts
- Bug fixes:
  - Dispatch on `Real` for `MOI.submit`
  - Implement `copy` for `CustomSet` in tests
  - Don't export private macros
  - Fix invalid assertion in nonlinear
- Error if constraint has `NaN` right-hand side
- Improve speed of tests
  - Lots of work modularizing files in `/test`
- Improve line numbers in macro error messages
- Print nonlinear subexpressions
- Various documentation updates
- Dependency updates:
  - Datastructures 0.18
  - MathOptFormat v0.5
  - Prep for MathOptInterface 0.9.15

34.9 Version 0.21.3 (June 18, 2020)

- Added Special Order Sets (SOS1 and SOS2) to JuMP with default weights to ease the creation of such constraints (#2212).
- Added functions `simplex_iterations`, `barrier_iterations` and `node_count` (#2201).
- Added function `reduced_cost` (#2205).
- Implemented `callback_value` for affine and quadratic expressions (#2231).
- Support `MutableArithmetics.Zero` in objective and constraints (#2219).
- Documentation improvements:
  - Mention tutorials in the docs (#2223).
  - Update COIN-OR links (#2242).
  - Explicit link to the documentation of `MOI.FileFormats` (#2253).
  - Typo fixes (#2261).
- Containers improvements:
  - Fix `Base.map` for `DenseAxisArray` (#2235).
  - Throw `BoundsError` if number of indices is incorrect for `DenseAxisArray` and `SparseAxisArray` (#2240).
- Extensibility improvements:
  - Implement a `set_objective` method fallback that redirects to `set_objective_sense` and `set_objective_function` (#2247).
  - Add `parse_constraint` method with arbitrary number of arguments (#2051).
  - Add `parse_constraint_expr` and `parse_constraint_head` (#2228).

34.10 Version 0.21.2 (April 2, 2020)

- Added `relative_gap()` to access `MOI.RelativeGap()` attribute (#2199).
- Documentation fixes:
  - Added link to source for docstrings in the documentation (#2207).
  - Added docstring for `@variables` macro (#2216).
  - Typo fixes (#2177, #2184, #2182).
- Implementation of methods for Base functions:
  - Implemented `Base.empty!` for `JuMP.Model` (#2198).
  - Implemented `Base.conj` for JuMP scalar types (#2209).
- Bug fixes:
  - Fixed sum of expression with scalar product in macro (#2178).
  - Fixed writing of nonlinear models to MathOptFormat (#2181).
  - Fixed construction of empty SparseAxisArray (#2179).
  - Fixed constraint with zero function (#2188).

34.11 Version 0.21.1 (Feb 18, 2020)

- Improved the clarity of the `with_optimizer` deprecation warning.

34.12 Version 0.21 (Feb 16, 2020)

Breaking changes:

- Deprecated `with_optimizer` (#2090, #2084, #2141). You can replace `with_optimizer` by either nothing, `optimizer_with_attributes` or a closure:
  - replace `with_optimizer(Ipopt.Optimizer)` by `Ipopt.Optimizer`.
  - replace `with_optimizer(Ipopt.Optimizer, max_cpu_time=60.0)` by `optimizer_with_attributes(Ipopt.Optimi "max_cpu_time" => 60.0)`.
  - replace `with_optimizer(Gurobi.Optimizer, env)` by `() -> Gurobi.Optimizer(env)`.
  - replace `with_optimizer(Gurobi.Optimizer, env, Presolve=0)` by `optimizer_with_attributes(() -> Gurobi.Optimizer(env), "Presolve" => 0)`.

  alternatively to `optimizer_with_attributes`, you can also set the attributes separately with `set_optimizer_attribute`.
- Renamed `set_parameter` and `set_parameters` to `set_optimizer_attribute` and `set_optimizer_attributes` (#2150).
- Broadcast should now be explicit inside macros. `@SDconstraint(model, x >= 1)` and `@constraint(model, x + 1 in SecondOrderCone())` now throw an error instead of broadcasting 1 along the dimension of `x` (#2107).
- `@SDconstraint(model, x >= 0)` is now equivalent to `@constraint(model, x in PSDCone())` instead of `@constraint(model, (x .- 0) in PSDCone())` (#2107).
- The macros now create the containers with `map` instead of `for` loops, as a consequence, containers created by `@expression` can now have any element type and containers of constraint references now have concrete element types when possible. This fixes a long-standing issue where `@expression` could only be used to generate a collection of linear expressions. Now it works for quadratic expressions as well (#2070).
- Calling `deepcopy(::AbstractModel)` now throws an error.
- The constraint name is now printed in the model string (#2108).

New features:

- Added support for solver-independent and solver-specific callbacks (#2101).
- Added `write_to_file` and `read_from_file`, supported formats are CBF, LP, MathOptFormat, MPS and SDPA (#2114).
- Added support for complementarity constraints (#2132).
- Added support for indicator constraints (#2092).
- Added support for querying multiple solutions with the `result` keyword (#2100).

- Added support for constraining variables on creation (#2128).
- Added method `delete` that deletes a vector of variables at once if it is supported by the underlying solver (#2135).
- The arithmetic between JuMP expression has be refactored into the MutableArithmetics package (#2107).
- Improved error on complex values in NLP (#1978).
- Added an example of column generation (#2010).

Bug fixes:

- Incorrect coefficients generated when using Symmetric variables (#2102)

34.13 Version 0.20.1 (Oct 18, 2019)

- Add sections on `@variables` and `@constraints` in the documentation (#2062).
- Fixed product of sparse matrices for Julia v1.3 (#2063).
- Added `set_objective_coefficient` to modify the coefficient of a linear term of the objective function (#2008).
- Added `set_time_limit_sec`, `unset_time_limit_sec` and `time_limit_sec` to set and query the time limit for the solver in seconds (#2053).

34.14 Version 0.20.0 (Aug 24, 2019)

- Documentation updates.
- Numerous bug fixes.
- Better error messages (#1977, #1978, #1997, #2017).
- Performance improvements (#1947, #2032).
- Added LP sensitivity summary functions `lp_objective_perturbation_range` and `lp_rhs_perturbation_range` (#1917).
- Added functions `dual_objective_value`, `raw_status` and `set_parameter`.
- Added function `set_objective_coefficient` to modify the coefficient of a linear term of the objective (#2008).
- Added functions `set_normalized_rhs`, `normalized_rhs`, and `add_to_function_constant` to modify and get the constant part of a constraint (#1935, #1960).
- Added functions `set_normalized_coefficient` and `normalized_coefficient` to modify and get the coefficient of a linear term of a constraint (#1935, #1960).
- Numerous other improvements in MOI 0.9, see the `NEWS.md` file of MOI for more details.

34.15 Version 0.19.2 (June 8, 2019)

- Fix a bug in derivatives that could arise in models with nested nonlinear subexpressions.

34.16 Version 0.19.1 (May 12, 2019)

- Usability and performance improvements.
- Bug fixes.

34.17 Version 0.19.0 (February 15, 2019)

JuMP 0.19 contains significant breaking changes.

Breaking changes:

- JuMP's abstraction layer for communicating with solvers changed from MathProgBase (MPB) to MathOptInterface (MOI). MOI addresses many longstanding design issues. (See @mlubin's slides from JuMP-dev 2018.) JuMP 0.19 is compatible only with solvers that have been updated for MOI. See the installation guide for a list of solvers that have and have not yet been updated.
- Most solvers have been renamed to `PackageName.Optimizer`. For example, `GurobiSolver()` is now `Gurobi.Optimizer`.
- Solvers are no longer added to a model via `Model(solver = XXX(kwargs...))`. Instead use `Model(with_optimizer(XXX, kwargs...))`. For example, `Model(with_optimizer(Gurobi.Optimizer, OutputFlag=0))`.
- JuMP containers (e.g., the objects returned by `@variable`) have been redesigned. `Containers.SparseAxisArray` replaces `JuMPDict`, `JuMPArray` was rewritten (inspired by `AxisArrays`) and renamed `Containers.DenseAxisArray`, and you can now request a container type with the `container=` keyword to the macros. See the corresponding documentation for more details.
- The statuses returned by solvers have changed. See the possible status values here. The MOI statuses are much richer than the MPB statuses and can be used to distinguish between previously indistinguishable cases (e.g. did the solver have a feasible solution when it stopped because of the time limit?).
- Starting values are separate from result values. Use `value` to query the value of a variable in a solution. Use `start_value` and `set_start_value` to get and set an initial starting point provided to the solver. The solutions from previous solves are no longer automatically set as the starting points for the next solve.
- The data structures for affine and quadratic expressions `AffExpr` and `QuadExpr` have changed. Internally, terms are stored in dictionaries instead of lists. Duplicate coefficients can no longer exist. Accessors and iteration methods have changed.
- `JuMPNLPEvaluator` no longer includes the linear and quadratic parts of the model in the evaluation calls. These are now handled separately to allow NLP solvers that support various types of constraints.
- JuMP solver-independent callbacks have been replaced by solver-specific callbacks. See your favorite solver for more details. (See the note below: No solver-specific callbacks are implemented yet.)
- The `norm()` syntax is no longer recognized inside macros. Use the `SecondOrderCone()` set instead.
- JuMP no longer performs automatic transformation between special quadratic forms and second-order cone constraints. Support for these constraint classes depends on the solver.

- The symbols `:Min` and `:Max` are no longer used as optimization senses. Instead, JuMP uses the `OptimizationSense` enum from MathOptInterface. `@objective(model, Max, ...)`, `@objective(model, Min, ...)`, `@NLobjective(model, Max, ...)`, and `@objective(model, Min, ...)` remain valid, but `@objective(m, :Max, ...)` is no longer accepted.
- The sign conventions for duals has changed in some cases for consistency with conic duality (see the documentation). The `shadow_price` helper method returns duals with signs that match conventional LP interpretations of dual values as sensitivities of the objective value to relaxations of constraints.
- `@constraintref` is no longer defined. Instead, create the appropriate container to hold constraint references manually. For example,

```
constraints = Dict() # Optionally, specify types for improved performance.
for i in 1:N
  constraints[i] = @constraint(model, ...)
end
```

- The `lowerbound`, `upperbound`, and `basename` keyword arguments to the `@variable` macro have been renamed to `lower_bound`, `upper_bound`, and `base_name`, for consistency with JuMP's new style recommendations.
- We rely on broadcasting syntax to apply accessors to collections of variables, e.g., `value.(x)` instead of `getvalue(x)` for collections. (Use `value(x)` when `x` is a scalar object.)

New features:

- Splatting (like `f(x...)`) is recognized in restricted settings in nonlinear expressions.
- Support for deleting constraints and variables.
- The documentation has been completely rewritten using docstrings and Documenter.
- Support for modeling mixed conic and quadratic models (e.g., conic models with quadratic objectives and bi-linear matrix inequalities).
- Significantly improved support for modeling new types of constraints and for extending JuMP's macros.
- Support for providing dual warm starts.
- Improved support for accessing solver-specific attributes (e.g., the irreducible inconsistent subsystem).
- Explicit control of whether symmetry-enforcing constraints are added to PSD constraints.
- Support for modeling exponential cones.
- Significant improvements in internal code quality and testing.
- Style and naming guidelines.
- Direct mode and manual mode provide explicit control over when copies of a model are stored and/or regenerated. See the corresponding documentation.

There are known regressions from JuMP 0.18 that will be addressed in a future release (0.19.x or later):

- Performance regressions in model generation (issue). Please file an issue anyway if you notice a significant performance regression. We have plans to address a number of performance issues, but we might not be aware of all of them.

- Fast incremental NLP solves are not yet reimplemented (issue).
- We do not yet have an implementation of solver-specific callbacks.
- The column generation syntax in `@variable` has been removed (i.e., the `objective`, `coefficients`, and `inconstraints` keyword arguments). Support for column generation will be re-introduced in a future release.
- The ability to solve the continuous relaxation (i.e. via `solve(model; relaxation = true)`) is not yet reimplemented (issue).

34.18 Version 0.18.5 (December 1, 2018)

- Support views in some derivative evaluation functions.
- Improved compatibility with PackageCompiler.

34.19 Version 0.18.4 (October 8, 2018)

- Fix a bug in model printing on Julia 0.7 and 1.0.

34.20 Version 0.18.3 (October 1, 2018)

- Add support for Julia v1.0 (Thanks @ExpandingMan)
- Fix matrix expressions with quadratic functions (#1508)

34.21 Version 0.18.2 (June 10, 2018)

- Fix a bug in second-order derivatives when expressions are present (#1319)
- Fix a bug in `@constraintref` (#1330)

34.22 Version 0.18.1 (April 9, 2018)

- Fix for nested tuple destructuring (#1193)
- Preserve internal model when relaxation=true (#1209)
- Minor bug fixes and updates for example

34.23 Version 0.18.0 (July 27, 2017)

- Drop support for Julia 0.5.
- Update for ForwardDiff 0.5.
- Minor bug fixes.

34.24 Version 0.17.1 (June 9, 2017)

- Use of `constructconstraint!` in `@SDconstraint`.
- Minor bug fixes.

34.25 Version 0.17.0 (May 27, 2017)

- **Breaking change**: Mixing quadratic and conic constraints is no longer supported.
- **Breaking change**: The $getvariable$ and $getconstraint$ functions are replaced by indexing on the corresponding symbol. For instance, to access the variable with name x, one should now write $m[:x]$ instead of $getvariable(m, :x)$. As a consequence, creating a variable and constraint with the same name now triggers a warning, and accessing one of them afterwards throws an error. This change is breaking only in the latter case.
- Addition of the $getobjectivebound$ function that mirrors the functionality of the MathProgBase $getobjbound$ function except that it takes into account transformations performed by JuMP.
- Minor bug fixes.

The following changes are primarily of interest to developers of JuMP extensions:

- The new syntax $@constraint(model, exprinCone)$ creates the constraint ensuring that $expr$ is inside $Cone$. The $Cone$ argument is passed to $constructconstraint!$ which enables the call to the dispatched to an extension.
- The $@variable$ macro now calls $constructvariable!$ instead of directly calling the $Variable$ constructor. Extra arguments and keyword arguments passed to $@variable$ are passed to $constructvariable!$ which enables the call to be dispatched to an extension.
- Refactor the internal function $conicdata$ (used build the MathProgBase conic model) into smaller subfunctions to make these parts reusable by extensions.

34.26 Version 0.16.2 (March 28, 2017)

- Minor bug fixes and printing tweaks
- Address deprecation warnings for Julia 0.6

34.27 Version 0.16.1 (March 7, 2017)

- Better support for $AbstractArray$ in JuMP (Thanks @tkoolen)
- Minor bug fixes

34.28 Version 0.16.0 (February 23, 2017)

- **Breaking change**: JuMP no longer has a mechanism for selecting solvers by default (the previous mechanism was flawed and incompatible with Julia 0.6). Not specifying a solver before calling $solve()$ will result in an error.
- **Breaking change**: User-defined functions are no longer global. The first argument to $JuMP.register$ is now a JuMP $Model$ object within whose scope the function will be registered. Calling $JuMP.register$ without a $Model$ now produces an error.
- **Breaking change**: Use the new $JuMP.fix$ method to fix a variable to a value or to update the value to which a variable is fixed. Calling $setvalue$ on a fixed variable now results in an error in order to avoid silent behavior changes. (Thanks @joaquimg)
- Nonlinear expressions now print out similarly to linear/quadratic expressions (useful for debugging!)
- New $category$ keyword to $@variable$. Used for specifying categories of anonymous variables.
- Compatibility with Julia 0.6-dev.
- Minor fixes and improvements (Thanks @cossio, @ccoffrin, @blegat)

34.29 Version 0.15.1 (January 31, 2017)

- Bugfix for $@LinearConstraints$ and friends

34.30 Version 0.15.0 (December 22, 2016)

- Julia 0.5.0 is the minimum required version for this release.
- Document support for BARON solver
- Enable info callbacks in more states than before, e.g. for recording solutions. New $when$ argument to $addinfocallback$ (#814, thanks @yeesian)
- Improved support for anonymous variables. This includes new warnings for potentially confusing use of the traditional non-anonymous syntax:
  - When multiple variables in a model are given the same name
  - When non-symbols are used as names, e.g., $@variable(m, x[1][1:N])$
- Improvements in iterating over JuMP containers (#836, thanks @IssamT)
- Support for writing variable names in .lp file output (Thanks @leethargo)
- Support for querying duals to SDP problems (Thanks @blegat)
- The comprehension syntax with curly braces sum, $prod$, and $norm2$ has been deprecated in favor of Julia's native comprehension syntax $sum()$, $prod()$ and $norm()$ as previously announced. (For early adopters of the new syntax, $norm2()$ was renamed to $norm()$ without deprecation.)
- Unit tests rewritten to use Base.Test instead of FactCheck
- Improved support for operations with matrices of JuMP types (Thanks @ExpandingMan)
- The syntax to halt a solver from inside a callback has changed from $throw(CallbackAbort())$ to $returnJuMP.StopTheSolver$
- Minor bug fixes

34.31 Version 0.14.2 (December 12, 2016)

- Allow singeton anonymous variables (includes bugfix)

34.32 Version 0.14.1 (September 12, 2016)

- More consistent handling of states in informational callbacks, includes a new $when$ parameter to $addinfocallback$ for specifying in which state an informational callback should be called.

34.33 Version 0.14.0 (August 7, 2016)

- Compatibility with Julia 0.5 and ForwardDiff 0.2
- Support for "anonymous" variables, constraints, expressions, and parameters, e.g., $x = @variable(m, [1 : N])$ instead of $@variable(m, x[1 : N])$
- Support for retrieving constraints from a model by name via $getconstraint$
- $@NLconstraint$ now returns constraint references (as expected).
- Support for vectorized expressions within lazy constraints
- On Julia 0.5, parse new comprehension syntax $sum(x[i]foriin1 : Nifisodd(i))$ instead of $sumx[i], iin1 : N; isodd(i)$. The old syntax with curly braces will be deprecated in JuMP 0.15.
- Now possible to provide nonlinear expressions as "raw" Julia $Expr$ objects instead of using JuMP's nonlinear macros. This input format is useful for programmatically generated expressions.
- $s/MathematicalProgramming/MathematicalOptimization/$
- Support for local cuts (Thanks to @madanim, Mehdi Madani)
- Document Xpress interface developed by @joaquimg, Joaquim Dias Garcia
- Minor bug and deprecation fixes (Thanks @odow, @jrevels)

34.34 Version 0.13.2 (May 16, 2016)

- Compatibility update for MathProgBase

34.35 Version 0.13.1 (May 3, 2016)

- Fix broken deprecation for $registerNLfunction$.

34.36 Version 0.13.0 (April 29, 2016)

- Most exported methods and macros have been renamed to avoid camelCase. See the list of changes here. There is a 1-1 mapping from the old names to the new, and it is safe to simply replace the names to update existing models.
- Specify variable lower/upper bounds in $@variable$ using the $lowerbound$ and $upperbound$ keyword arguments.

- Change name printed for variable using the $basename$ keyword argument to $@variable$.
- New $@variables$ macro allows multiline declaration of groups of variables.
- A number of solver methods previously available only through MathProgBase are now exposed directly in JuMP. The fix was recorded live!
- Compatibility fixes with Julia 0.5.
- The "end" indexing syntax is no longer supported within JuMPArrays which do not use 1-based indexing until upstream issues are resolved, see here.

34.37 Version 0.12.2 (March 9, 2016)

- Small fixes for nonlinear optimization

34.38 Version 0.12.1 (March 1, 2016)

- Fix a regression in slicing for JuMPArrays (when not using 1-based indexing)

34.39 Version 0.12.0 (February 27, 2016)

- The automatic differentiation functionality has been completely rewritten with a number of user-facing changes:
  - $@defExpr$ and $@defNLExpr$ now take the model as the first argument. The previous one-argument version of $@defExpr$ is deprecated; all expressions should be named. E.g., replace $@defExpr(2x + y)$ with $@defExpr(jump_model, my_expr, 2x + y)$.
  - JuMP no longer uses Julia's variable binding rules for efficiently re-solving a sequence of nonlinear models. Instead, we have introduced nonlinear parameters. This is a breaking change, so we have added a warning message when we detect models that may depend on the old behavior.
  - Support for user-defined functions integrated within nonlinear JuMP expressions.
- Replaced iteration over $AffExpr$ with $Number$-like scalar iteration; previous iteration behavior is now available via $linearterms(::AffExpr)$.
- Stopping the solver via $throw(CallbackAbort())$ from a callback no longer triggers an exception. Instead, $solve()$ returns $UserLimit$ status.
- $getDual()$ now works for conic problems (Thanks @emreyamangil.)

34.40 Version 0.11.3 (February 4, 2016)

- Bug-fix for problems with quadratic objectives and semidefinite constraints

34.41 Version 0.11.2 (January 14, 2016)

- Compatibility update for Mosek

34.42 Version 0.11.1 (December 1, 2015)

- Remove usage of `@compat` in tests.
- Fix updating quadratic objectives for nonlinear models.

34.43 Version 0.11.0 (November 30, 2015)

- Julia 0.4.0 is the minimum required version for this release.
- Fix for scoping semantics of index variables in sum{}. Index variables no longer leak into the surrounding scope.
- Addition of the $solve(m :: Model, relaxation = true)$ keyword argument to solve the standard continuous realaxation of model m
- The $getConstraintBounds()$ method allows access to the lower and upper bounds of all constraints in a (nonlinear) model.
- Update for breaking changes in MathProgBase

34.44 Version 0.10.3 (November 20, 2015)

- Fix a rare error when parsing quadratic expressions
- Fix $Variable()$ constructor with default arguments
- Detect unrecognized keywords in $solve()$

34.45 Version 0.10.2 (September 28, 2015)

- Fix for deprecation warnings

34.46 Version 0.10.1 (September 3, 2015)

- Fixes for ambiguity warnings.
- Fix for breaking change in precompilation syntax in Julia 0.4-pre

34.47 Version 0.10.0 (August 31, 2015)

- Support (on Julia 0.4 and later) for conditions in indexing $@defVar$ and $@addConstraint$ constructs, e.g. $@defVar(m, x[i = 1:5, j = 1:5; i + j >= 3])$
- Support for vectorized operations on Variables and expressions. See the documentation for details.
- New $getVar()$ method to access variables in a model by name
- Support for semidefinite programming.

- Dual solutions are now available for general nonlinear problems. You may call $getDual$ on a reference object for a nonlinear constraint, and $getDual$ on a variable object for Lagrange multipliers from active bounds.
- Introduce warnings for two common performance traps: too many calls to $getValue()$ on a collection of variables and use of the $+$ operator in a loop to sum expressions.
- Second-order cone constraints can be written directly with the $norm()$ and $norm2$ syntax.
- Implement MathProgBase interface for querying Hessian-vector products.
- Iteration over $JuMPContainer$s is deprecated; instead, use the $keys$ and $values$ functions, and $zip(keys(d), values(d))$ for the old behavior.
- $@defVar$ returns $ArrayVariable, N$ when each of N index sets are of the form $1 : n$.
- Module precompilation: on Julia 0.4 and later, $usingJuMP$ is now much faster.

34.48 Version 0.9.3 (August 11, 2015)

- Fixes for FactCheck testing on julia v0.4.

34.49 Version 0.9.2 (June 27, 2015)

- Fix bug in @addConstraints.

34.50 Version 0.9.1 (April 25, 2015)

- Fix for Julia 0.4-dev.
- Small infrastructure improvements for extensions.

34.51 Version 0.9.0 (April 18, 2015)

- Comparison operators for constructing constraints (e.g. $2x >= 1$) have been deprecated. Instead, construct the constraints explicitly in the $@addConstraint$ macro to add them to the model, or in the $@LinearConstraint$ macro to create a stand-alone linear constraint instance.
- $getValue()$ method implemented to compute the value of a nonlinear subexpression
- JuMP is now released under the Mozilla Public License version 2.0 (was previously LGPL). MPL is a copyleft license which is less restrictive than LGPL, especially for embedding JuMP within other applications.
- A number of performance improvements in ReverseDiffSparse for computing derivatives.
- $MathProgBase.getsolvetime(m)$ now returns the solution time reported by the solver, if available. (Thanks @odow, Oscar Dowson)
- Formatting fix for LP format output. (Thanks @sbebo, Leonardo Taccari).

34.52 Version 0.8.0 (February 17, 2015)

- Nonlinear subexpressions now supported with the $@defNLExpr$ macro.
- SCS supported for solving second-order conic problems.
- $setXXXCallback$ family deprecated in favor of $addXXXCallback$.
- Multiple callbacks of the same type can be registered.
- Added support for informational callbacks via $addInfoCallback$.
- A $CallbackAbort$ exception can be thrown from callback to safely exit optimization.

34.53 Version 0.7.4 (February 4, 2015)

- Reduced costs and linear constraint duals are now accessible when quadratic constraints are present.
- Two-sided nonlinear constraints are supported.
- Methods for accessing the number of variables and constraints in a model are renamed.
- New default procedure for setting initial values in nonlinear optimization: project zero onto the variable bounds.
- Small bug fixes.

34.54 Version 0.7.3 (January 14, 2015)

- Fix a method ambiguity conflict with Compose.jl (cosmetic fix)

34.55 Version 0.7.2 (January 9, 2015)

- Fix a bug in $sum(::JuMPDict)$
- Added the $setCategory$ function to change a variables category (e.g. continuous or binary)

after construction, and $getCategory$ to retrieve the variable category.

34.56 Version 0.7.1 (January 2, 2015)

- Fix a bug in parsing linear expressions in macros. Affects only Julia 0.4 and later.

34.57 Version 0.7.0 (December 29, 2014)

Linear/quadratic/conic programming

- **Breaking change**: The syntax for column-wise model generation has been changed to use keyword arguments in $@defVar$.
- On Julia 0.4 and later, variables and coefficients may be multiplied in any order within macros. That is, variable*coefficient is now valid syntax.
- ECOS supported for solving second-order conic problems.

Nonlinear programming

- Support for skipping model generation when solving a sequence of nonlinear models with changing data.
- Fix a memory leak when solving a sequence of nonlinear models.
- The $@addNLConstraint$ macro now supports the three-argument version to define sets of nonlinear constraints.
- KNITRO supported as a nonlinear solver.
- Speed improvements for model generation.
- The $@addNLConstraints$ macro supports adding multiple (groups of) constraints at once. Syntax is similar to $@addConstraints$.
- Discrete variables allowed in nonlinear problems for solvers which support them (currently only KNITRO).

General

- Starting values for variables may now be specified with $@defVar(m, x, start = value)$.
- The $setSolver$ function allows users to change the solver subsequent to model creation.
- Support for "fixed" variables via the $@defVar(m, x == 1)$ syntax.
- Unit tests rewritten to use FactCheck.jl, improved testing across solvers.

34.58 Version 0.6.3 (October 19, 2014)

- Fix a bug in multiplying two AffExpr objects.

34.59 Version 0.6.2 (October 11, 2014)

- Further improvements and bug fixes for printing.
- Fixed a bug in $@defExpr$.
- Support for accessing expression graphs through the MathProgBase NLP interface.

34.60 Version 0.6.1 (September 19, 2014)

- Improvements and bug fixes for printing.

34.61 Version 0.6.0 (September 9, 2014)

- Julia 0.3.0 is the minimum required version for this release.
- $buildInternalModel(m :: Model)$ added to build solver-level model in memory without optimizing.
- Deprecate $load_model_only$ keyword argument to $solve$.
- Add groups of constraints with $@addConstraints$ macro.

- Unicode operators now supported, including for sum, for $prod$, and /
- Quadratic constraints supported in $@addConstraint$ macro.
- Quadratic objectives supported in $@setObjective$ macro.
- MathProgBase solver-independent interface replaces Ipopt-specific interface for nonlinear problems
  - **Breaking change**: $IpoptOptions$ no longer supported to specify solver options, use $m = Model(solver = IpoptSolver(options...))$ instead.
- New solver interfaces: ECOS, NLopt, and nonlinear support for MOSEK
- New option to control whether the lazy constraint callback is executed at each node in the B&B tree or just when feasible solutions are found
- Add support for semicontinuous and semi-integer variables for those solvers that support them.
- Add support for index dependencies (e.g. triangular indexing) in $@defVar$, $@addConstraint$, and $@defExpr$ (e.g. $@defVar(m, x[i = 1 : 10, j = i : 10])$).
  - This required some changes to the internal structure of JuMP containers, which may break code that explicitly stored $JuMPDict$ objects.

34.62 Version 0.5.8 (September 24, 2014)

- Fix a bug with specifying solvers (affects Julia 0.2 only)

34.63 Version 0.5.7 (September 5, 2014)

- Fix a bug in printing models

34.64 Version 0.5.6 (September 2, 2014)

- Add support for semicontinuous and semi-integer variables for those solvers that support them.
  - **Breaking change**: Syntax for $Variable()$ constructor has changed (use of this interface remains discouraged)
- Update for breaking changes in MathProgBase

34.65 Version 0.5.5 (July 6, 2014)

- Fix bug with problem modification: adding variables that did not appear in existing constraints or objective.

34.66 Version 0.5.4 (June 19, 2014)

- Update for breaking change in MathProgBase which reduces loading times for $using JuMP$
- Fix error when MIPs not solved to optimality

34.67 Version 0.5.3 (May 21, 2014)

- Update for breaking change in ReverseDiffSparse

34.68 Version 0.5.2 (May 9, 2014)

- Fix compatibility with Julia 0.3 prerelease

34.69 Version 0.5.1 (May 5, 2014)

- Fix a bug in coefficient handling inside lazy constraints and user cuts

34.70 Version 0.5.0 (May 2, 2014)

- Support for nonlinear optimization with exact, sparse second-order derivatives automatically computed. Ipopt is currently the only solver supported.
- $getValue$ for $AffExpr$ and $QuadExpr$
- **Breaking change**: $getSolverModel$ replaced by $getInternalModel$, which returns the internal MathProgBase-level model
- Groups of constraints can be specified with $@addConstraint$ (see documentation for details). This is not a breaking change.
- $dot(::JuMPDictVariable, ::JuMPDictVariable)$ now returns the corresponding quadratic expression.

34.71 Version 0.4.1 (March 24, 2014)

- Fix bug where change in objective sense was ignored when re-solving a model.
- Fix issue with handling zero coefficients in AffExpr.

34.72 Version 0.4.0 (March 10, 2014)

- Support for SOS1 and SOS2 constraints.
- Solver-independent callback for user heuristics.
- dot and sum implemented for $JuMPDict$ objects. Now you can say $@addConstraint(m, dot(a, x) <= b)$.
- Developers: support for extensions to JuMP. See definition of Model in $src/JuMP.jl$ for more details.
- Option to construct the low-level model before optimizing.

34.73 Version 0.3.2 (February 17, 2014)

- Improved model printing
  - Preliminary support for IJulia output

34.74 Version 0.3.1 (January 30, 2014)

- Documentation updates
  - Support for MOSEK
  - CPLEXLink renamed to CPLEX

34.75 Version 0.3.0 (January 21, 2014)

- Unbounded/infeasibility rays: getValue() will return the corresponding components of an unbounded ray when a model is unbounded, if supported by the selected solver. getDual() will return an infeasibility ray (Farkas proof) if a model is infeasible and the selected solver supports this feature.
- Solver-independent callbacks for user generated cuts.
- Use new interface for solver-independent QCQP.
- $setlazycallback$ renamed to $setLazyCallback$ for consistency.

34.76 Version 0.2.0 (December 15, 2013)

- **Breaking change**: Objective sense is specified in setObjective instead of in the Model constructor.
- **Breaking change**: $lpsolver$ and $mipsolver$ merged into single $solver$ option.
- Problem modification with efficient LP restarts and MIP warm-starts.
- Relatedly, column-wise modeling now supported.
- Solver-independent callbacks supported. Currently we support only a "lazy constraint" callback, which works with Gurobi, CPLEX, and GLPK. More callbacks coming soon.

34.77 Version 0.1.2 (November 16, 2013)

- Bug fixes for printing, improved error messages.
- Allow $AffExpr$ to be used in macros; e.g., $ex = y + z; @addConstraint(m, x + 2 * ex <= 3)$

34.78 Version 0.1.1 (October 23, 2013)

- Update for solver specification API changes in MathProgBase.

34.79 Version 0.1.0 (October 3, 2013)

- Initial public release.

Part VIII

MathOptInterface

Chapter 35

Introduction

35.1 Introduction

> **Warning**
>
> This documentation in this section is a copy of the official MathOptInterface documentation available at https://jump.dev/MathOptInterface.jl/v0.10.4. It is included here to make it easier to link concepts between JuMP and MathOptInterface.

What is MathOptInterface?

MathOptInterface.jl (MOI) is an abstraction layer designed to provide a unified interface to mathematical optimization solvers so that users do not need to understand multiple solver-specific APIs.

> **Tip**
>
> This documentation is aimed at developers writing software interfaces to solvers and modeling languages using the MathOptInterface API. If you are a user interested in solving optimization problems, we encourage you instead to use MOI through a higher-level modeling interface like JuMP or Convex.jl.

How the documentation is structured

Having a high-level overview of how this documentation is structured will help you know where to look for certain things.

- The **Background** section contains articles on the motivation and theory behind MathOptInterface. Look here if you want to understand why, rather than how.
- The **Tutorials** section contains articles on how to use and implement the MathOptInteraface API. Look here if you want to write a model in MOI, or write an interface to a new solver.
- The **Manual** contains short code-snippets that explain how to use the MOI API. Look here for more details on particular areas of MOI.
- The **API Reference** contains a complete list of functions and types that comprise the MOI API. Look here is you want to know how to use (or implement) a particular function.
- The **Submodules** section contains stand-alone documentation for each of the submodules within MOI. These submodules are not required to interface a solver with MOI, but they make the job much easier.

Citing MathOptInterface

A paper describing the design and features of MathOptInterface is available on arXiv.

If you find MathOptInterface useful in your work, we kindly request that you cite the following paper:

```
@misc{
    legat2020mathoptinterface,
    title = {MathOptInterface: a data structure for mathematical optimization problems},
    author = {Beno{\^i}t Legat and Oscar Dowson and Joaquim Dias Garcia and Miles Lubin},
    year = {2020},
    eprint = {2002.03447},
    archivePrefix = {arXiv},
    primaryClass = {math.OC},
    url = {https://arxiv.org/abs/2002.03447},
}
```

Chapter 36

Background

36.1 Motivation

MOI has been designed to replace MathProgBase, which was been used by modeling packages such as JuMP and Convex.jl.

This second-generation abstraction layer addresses a number of limitations of MathProgBase.

MOI is designed to:

- Be simple and extensible, unifying linear, quadratic, and conic optimization, and seamlessly facilitate extensions to essentially arbitrary constraints and functions (e.g., indicator constraints, complementarity constraints, and piecewise-linear functions)
- Be fast by allowing access to a solver's in-memory representation of a problem without writing intermediate files (when possible) and by using multiple dispatch and avoiding requiring containers of nonconcrete types
- Allow a solver to return multiple results (e.g., a pool of solutions)
- Allow a solver to return extra arbitrary information via attributes (e.g., variable- and constraint-wise membership in an irreducible inconsistent subset for infeasibility analysis)
- Provide a greatly expanded set of status codes explaining what happened during the optimization procedure
- Enable a solver to more precisely specify which problem classes it supports
- Enable both primal and dual warm starts
- Enable adding and removing both variables and constraints by indices that are not required to be consecutive
- Enable any modification that the solver supports to an existing model
- Avoid requiring the solver wrapper to store an additional copy of the problem data

36.2 Duality

Conic duality is the starting point for MOI's duality conventions. When all functions are affine (or coordinate projections), and all constraint sets are closed convex cones, the model may be called a conic optimization problem.

For a minimization problem in geometric conic form, the primal is:

$$\min_{x \in \mathbb{R}^n} \quad a_0^T x + b_0 \tag{36.1}$$

$$\text{s.t.} \quad A_i x + b_i \in \mathcal{C}_i \qquad i = 1 \dots m \tag{36.2}$$

and the dual is a maximization problem in standard conic form:

$$\max_{y_1, \dots, y_m} \quad -\sum_{i=1}^{m} b_i^T y_i + b_0 \tag{36.3}$$

$$\text{s.t.} \quad a_0 - \sum_{i=1}^{m} A_i^T y_i = 0 \tag{36.4}$$

$$y_i \in \mathcal{C}_i^* \qquad i = 1 \dots m \tag{36.5}$$

where each $\mathcal{C}_i$ is a closed convex cone and $\mathcal{C}_i^*$ is its dual cone.

For a maximization problem in geometric conic form, the primal is:

$$\max_{x \in \mathbb{R}^n} \quad a_0^T x + b_0 \tag{36.6}$$

$$\text{s.t.} \quad A_i x + b_i \in \mathcal{C}_i \qquad i = 1 \dots m \tag{36.7}$$

and the dual is a minimization problem in standard conic form:

$$\min_{y_1, \dots, y_m} \quad \sum_{i=1}^{m} b_i^T y_i + b_0 \tag{36.8}$$

$$\text{s.t.} \quad a_0 + \sum_{i=1}^{m} A_i^T y_i = 0 \tag{36.9}$$

$$y_i \in \mathcal{C}_i^* \qquad i = 1 \dots m \tag{36.10}$$

A linear inequality constraint $a^T x + b \geq c$ should be interpreted as $a^T x + b - c \in \mathbb{R}_+$, and similarly $a^T x + b \leq c$ should be interpreted as $a^T x + b - c \in \mathbb{R}_-$. Variable-wise constraints should be interpreted as affine constraints with the appropriate identity mapping in place of A_i.

For the special case of minimization LPs, the MOI primal form can be stated as:

$$\min_{x \in \mathbb{R}^n} \quad a_0^T x + b_0 \tag{36.11}$$

$$\text{s.t.} \quad A_1 x \geq b_1 \tag{36.12}$$

$$A_2 x \leq b_2 \tag{36.13}$$

$$A_3 x = b_3 \tag{36.14}$$

By applying the stated transformations to conic form, taking the dual, and transforming back into linear inequality form, one obtains the following dual:

$$\begin{aligned} \max_{y_1,y_2,y_3} \quad & b_1^T y_1 + b_2^T y_2 + b_3^T y_3 + b_0 && (36.15)\\ \text{s.t.} \quad & A_1^T y_1 + A_2^T y_2 + A_3^T y_3 = a_0 && (36.16)\\ & y_1 \geq 0 && (36.17)\\ & y_2 \leq 0 && (36.18) \end{aligned}$$

For maximization LPs, the MOI primal form can be stated as:

$$\begin{aligned} \max_{x \in \mathbb{R}^n} \quad & a_0^T x + b_0 && (36.19)\\ \text{s.t.} \quad & A_1 x \geq b_1 && (36.20)\\ & A_2 x \leq b_2 && (36.21)\\ & A_3 x = b_3 && (36.22) \end{aligned}$$

and similarly, the dual is:

$$\begin{aligned} \min_{y_1,y_2,y_3} \quad & -b_1^T y_1 - b_2^T y_2 - b_3^T y_3 + b_0 && (36.23)\\ \text{s.t.} \quad & A_1^T y_1 + A_2^T y_2 + A_3^T y_3 = -a_0 && (36.24)\\ & y_1 \geq 0 && (36.25)\\ & y_2 \leq 0 && (36.26) \end{aligned}$$

Warning

For the LP case, the signs of the feasible dual variables depend only on the sense of the corresponding primal inequality and not on the objective sense.

Duality and scalar product

The scalar product is different from the canonical one for the sets `PositiveSemidefiniteConeTriangle`, `LogDetConeTriangle`, `RootDetConeTriangle`.

If the set C_i of the section Duality is one of these three cones, then the rows of the matrix A_i corresponding to off-diagonal entries are twice the value of the `coefficients` field in the `VectorAffineFunction` for the corresponding rows. See `PositiveSemidefiniteConeTriangle` for details.

Dual for problems with quadratic functions

Given a problem with quadratic functions:

$$\begin{aligned} \min_{x \in \mathbb{R}^n} \quad & \frac{1}{2} x^T Q_0 x + a_0^T x + b_0 \\ \text{s.t.} \quad & \frac{1}{2} x^T Q_i x + a_i^T x + b_i \in \mathcal{C}_i \qquad i = 1 \ldots m \end{aligned}$$

with cones $\mathcal{C}_i \subseteq \mathbb{R}$ for $i = 1 \ldots m$, consider the Lagrangian function

$$L(x,y) = \frac{1}{2}x^T Q_0 x + a_0^T x + b_0 - \sum_{i=1}^{m} y_i (\frac{1}{2}x^T Q_i x + a_i^T x + b_i)$$

A pair of primal-dual variables $(x^\star, y^\star)$ is optimal if

- $x^\star$ is a minimizer of

$$\min_{x \in \mathbb{R}^n} L(x, y^\star).$$

  That is,

$$0 = \nabla_x L(x, y^\star) = Q_0 x + a_0 - \sum_{i=1}^{m} y_i^\star (Q_i x + a_i).$$

- and $y^\star$ is a maximizer of

$$\max_{y_i \in \mathcal{C}_i^*} L(x^\star, y).$$

  That is, for all $i = 1, \ldots, m$, $\frac{1}{2}x^T Q_i x + a_i^T x + b_i$ is either zero or in the normal cone of $\mathcal{C}_i^*$ at $y^\star$. For instance, if $\mathcal{C}_i$ is $\{x \in \mathbb{R} : x \leq 0\}$, this means that if $\frac{1}{2}x^T Q_i x + a_i^T x + b_i$ is nonzero at $x^\star$ then $y_i^\star = 0$. This is the classical complementary slackness condition.

If $\mathcal{C}_i$ is a vector set, the discussion remains valid with $y_i(\frac{1}{2}x^T Q_i x + a_i^T x + b_i)$ replaced with the scalar product between y_i and the vector of scalar-valued quadratic functions.

Note

For quadratic programs with only affine constraints, the optimality condition $\nabla_x L(x, y^\star) = 0$ can be simplified as follows:

$$0 = \nabla_x L(x, y^\star) = Q_0 x + a_0 - \sum_{i=1}^{m} y_i^\star a_i$$

which gives

$$Q_0 x = \sum_{i=1}^{m} y_i^\star a_i - a_0.$$

The Lagrangian function

$$L(x,y) = \frac{1}{2}x^T Q_0 x + a_0^T x + b_0 - \sum_{i=1}^{m} y_i (a_i^T x + b_i)$$

can be rewritten as

$$L(x, y) = \frac{1}{2}x^T Q_0 x - (\sum_{i=1}^{m} y_i a_i^T - a_0^T)x + b_0 - \sum_{i=1}^{m} y_i(a_i^T x + b_i)$$

which, using the optimality condition $\nabla_x L(x, y^\star) = 0$, can be simplified as

$$L(x, y) = -\frac{1}{2}x^T Q_0 x + b_0 - \sum_{i=1}^{m} y_i(a_i^T x + b_i)$$

36.3 Naming conventions

MOI follows several conventions for naming functions and structures. These should also be followed by packages extending MOI.

Sets

Sets encode the structure of constraints. Their names should follow the following conventions:

- Abstract types in the set hierarchy should begin with `Abstract` and end in `Set`, e.g., `AbstractScalarSet`, `AbstractVectorSet`.
- Vector-valued conic sets should end with `Cone`, e.g., `NormInfinityCone`, `SecondOrderCone`.
- Vector-valued Cartesian products should be plural and not end in `Cone`, e.g., `Nonnegatives`, not `NonnegativeCone`.
- Matrix-valued conic sets should provide two representations: `ConeSquare` and `ConeTriangle`, e.g., `RootDetConeTriangle` and `RootDetConeSquare`. See Matrix cones for more details.
- Scalar sets should be singular, not plural, e.g., `Integer`, not `Integers`.
- As much as possible, the names should follow established conventions in the domain where this set is used: for instance, convex sets should have names close to those of CVX, and constraint-programming sets should follow MiniZinc's constraints.

Chapter 37

Tutorials

37.1 Solving a problem using MathOptInterface

In this example, we want to solve a binary-constrained knapsack problem:

$$\begin{aligned} \max\ & c^\top x \\ s.t.\ & w^\top x \leq C \\ & x_i \in \{0,1\}, \quad \forall i = 1, \ldots, n \end{aligned}$$

Load the MathOptInterface module and define the shorthand `MOI`:

```
using MathOptInterface
const MOI = MathOptInterface
```

As an optimizer, we choose GLPK:

```
using GLPK
optimizer = GLPK.Optimizer()
```

Define the data

We first define the constants of the problem:

```
julia> c = [1.0, 2.0, 3.0]
3-element Vector{Float64}:
 1.0
 2.0
 3.0

julia> w = [0.3, 0.5, 1.0]
3-element Vector{Float64}:
 0.3
 0.5
 1.0

julia> C = 3.2
3.2
```

Add the variables

```
julia> x = MOI.add_variables(optimizer, length(c));
```

Set the objective

```
julia> MOI.set(
           optimizer,
           MOI.ObjectiveFunction{MOI.ScalarAffineFunction{Float64}}(),
           MOI.ScalarAffineFunction(MOI.ScalarAffineTerm.(c, x), 0.0),
       );

julia> MOI.set(optimizer, MOI.ObjectiveSense(), MOI.MAX_SENSE)
```

> **Tip**
>
> `MOI.ScalarAffineTerm.(c, x)` is a shortcut for `[MOI.ScalarAffineTerm(c[i], x[i]) for i = 1:3]`. This is Julia's broadcast syntax in action, and is used quite often throughout MOI.

Add the constraints

We add the knapsack constraint and integrality constraints:

```
julia> MOI.add_constraint(
           optimizer,
           MOI.ScalarAffineFunction(MOI.ScalarAffineTerm.(w, x), 0.0),
           MOI.LessThan(C),
       );
```

Add integrality constraints:

```
julia> for x_i in x
           MOI.add_constraint(optimizer, x_i, MOI.ZeroOne())
       end
```

Optimize the model

```
julia> MOI.optimize!(optimizer)
```

Understand why the solver stopped

The first thing to check after optimization is why the solver stopped, e.g., did it stop because of a time limit or did it stop because it found the optimal solution?

```
julia> MOI.get(optimizer, MOI.TerminationStatus())
OPTIMAL::TerminationStatusCode = 1
```

Looks like we found an optimal solution!

Understand what solution was returned

```
julia> MOI.get(optimizer, MOI.ResultCount())
1

julia> MOI.get(optimizer, MOI.PrimalStatus())
FEASIBLE_POINT::ResultStatusCode = 1

julia> MOI.get(optimizer, MOI.DualStatus())
NO_SOLUTION::ResultStatusCode = 0
```

Query the objective

What is its objective value?

```
julia> MOI.get(optimizer, MOI.ObjectiveValue())
6.0
```

Query the primal solution

And what is the value of the variables x?

```
julia> MOI.get(optimizer, MOI.VariablePrimal(), x)
3-element Vector{Float64}:
 1.0
 1.0
 1.0
```

37.2 Implementing a solver interface

This guide outlines the basic steps to implement an interface to MathOptInterface for a new solver.

> **Warning**
>
> Implementing an interface to MathOptInterface for a new solver is a lot of work. Before starting, we recommend that you join the Developer chatroom and explain a little bit about the solver you are wrapping. If you have questions that are not answered by this guide, please ask them in the Developer chatroom so we can improve this guide!

A note on the API

The API of MathOptInterface is large and varied. In order to support the diversity of solvers and use-cases, we make heavy use of duck-typing. That is, solvers are not expected to implement the full API, nor is there a well-defined minimal subset of what must be implemented. Instead, you should implement the API as necessary in order to make the solver function as you require.

The main reason for using duck-typing is that solvers work in different ways and target different use-cases.

For example:

- Some solvers support incremental problem construction, support modification after a solve, and have native support for things like variable names.
- Other solvers are "one-shot" solvers that require all of the problem data to construct and solve the problem in a single function call. They do not support modification or things like variable names.

- Other "solvers" are not solvers at all, but things like file readers. These may only support functions like `read_from_file`, and may not even support the ability to add variables or constraints directly!
- Finally, some "solvers" are layers which take a problem as input, transform it according to some rules, and pass the transformed problem to an inner solver.

Preliminaries

Decide if MathOptInterface is right for you

The first step in writing a wrapper is to decide whether implementing an interface is the right thing to do.

MathOptInterface is an abstraction layer for unifying constrained mathematical optimization solvers. If your solver doesn't fit in the category, i.e., it implements a derivative-free algorithm for unconstrained objective functions, MathOptInterface may not be the right tool for the job.

> **Tip**
>
> If you're not sure whether you should write an interface, ask in the Developer chatroom.

Find a similar solver already wrapped

The next step is to find (if possible) a similar solver that is already wrapped. Although not strictly necessary, this will be a good place to look for inspiration when implementing your wrapper.

The JuMP documentation has a good list of solvers, along with the problem classes they support.

> **Tip**
>
> If you're not sure which solver is most similar, ask in the Developer chatroom.

Create a low-level interface

Before writing a MathOptInterface wrapper, you first need to be able to call the solver from Julia.

Wrapping solvers written in Julia If your solver is written in Julia, there's nothing to do here! Go to the next section.

Wrapping solvers written in C Julia is well suited to wrapping solvers written in C.

> **Info**
>
> This is not true for C++. If you have a solver written in C++, first write a C interface, then wrap the C interface.

Before writing a MathOptInterface wrapper, there are a few extra steps.

Create a JLL If the C code is publicly available under an open-source license, create a JLL package via Yggdrasil. The easiest way to do this is to copy an existing solver. Good examples to follow are the COIN-OR solvers.

> **Warning**
>
> Building the solver via Yggdrasil is non-trivial. Please ask the Developer chatroom for help.

If the code is commercial or not publicly available, the user will need to manually install the solver. See Gurobi.jl or CPLEX.jl for examples of how to structure this.

Use Clang.jl to wrap the C API The next step is to use Clang.jl to automatically wrap the C API. The easiest way to do this is to follow an example. Good examples to follow are Cbc.jl and HiGHS.jl.

Sometimes, you will need to make manual modifications to the resulting files.

Solvers written in other languages Ask the Developer chatroom for advice. You may be able to use one of the JuliaInterop packages to call out to the solver.

For example, SeDuMi.jl uses MATLAB.jl to call the SeDuMi solver written in MATLAB.

Structuring the package

Structure your wrapper as a Julia package. Consult the Julia documentation if you haven't done this before.

MOI solver interfaces may be in the same package as the solver itself (either the C wrapper if the solver is accessible through C, or the Julia code if the solver is written in Julia, for example), or in a separate package which depends on the solver package.

> **Note**
>
> The JuMP core contributors request that you do not use "JuMP" in the name of your package without prior consent.

Your package should have the following structure:

```
/.github
    /workflows
        ci.yml
        format_check.yml
        TagBot.yml
/gen
    gen.jl  # Code to wrap the C API
/src
    NewSolver.jl
    /gen
        libnewsolver_api.jl
        libnewsolver_common.jl
    /MOI_wrapper
        MOI_wrapper.jl
        other_files.jl
/test
    runtests.jl
    /MOI_wrapper
        MOI_wrapper.jl
.gitignore
.JuliaFormatter.toml
README.md
LICENSE.md
Project.toml
```

- The `/.github` folder contains the scripts for GitHub actions. The easiest way to write these is to copy the ones from an existing solver.
- The `/gen` and `/src/gen` folders are only needed if you are wrapping a solver written in C.
- The `/src/MOI_wrapper` folder contains the Julia code for the MOI wrapper.

- The `/test` folder contains code for testing your package. See Setup tests for more information.
- The `.JuliaFormatter.toml` and `.github/workflows/format_check.yml` enforce code formatting using JuliaFormatter.jl. Check existing solvers or JuMP.jl for details.

Setup tests

The best way to implement an interface to MathOptInterface is via test-driven development.

The `MOI.Test` submodule contains a large test suite to help check that you have implemented things correctly.

Follow the guide How to test a solver to set up the tests for your package.

> **Tip**
>
> Run the tests frequently when developing. However, at the start there is going to be a lot of errors! Start by excluding large classes of tests (e.g., `exclude = ["test_basic_", "test_model_"]`, implement any missing methods until the tests pass, then remove an exclusion and repeat.

Initial code

By this point, you should have a package setup with tests, formatting, and access to the underlying solver. Now it's time to start writing the wrapper.

The `Optimizer` object

The first object to create is a subtype of `AbstractOptimizer`. This type is going to store everything related to the problem.

By convention, these optimizers should not be exported and should be named `PackageName.Optimizer`.

```
import MathOptInterface
const MOI = MathOptInterface

struct Optimizer <: MOI.AbstractOptimizer
    # Fields go here
end
```

Optimizer objects for C solvers

> **Warning**
>
> This section is important if you wrap a solver written in C.

Wrapping a solver written in C will require the use of pointers, and for you to manually free the solver's memory when the `Optimizer` is garbage collected by Julia.

Never pass a pointer directly to a Julia `ccall` function.

Instead, store the pointer as a field in your `Optimizer`, and implement `Base.cconvert` and `Base.unsafe_convert`. Then you can pass `Optimizer` to any `ccall` function that expects the pointer.

In addition, make sure you implement a `finalizer` for each model you create.

If `newsolver_createProblem()` is the low-level function that creates the problem pointer in C, and `newsolver_freeProblem(::Pt` is the low-level function that frees memory associated with the pointer, your `Optimizer()` function should look like this:

```
struct Optimizer <: MOI.AbstractOptimizer
    ptr::Ptr{Cvoid}

    function Optimizer()
        ptr = newsolver_createProblem()
        model = Optimizer(ptr)
        finalizer(model) do m
            newsolver_freeProblem(m)
            return
        end
        return model
    end
end

Base.cconvert(::Type{Ptr{Cvoid}}, model::Optimizer) = model
Base.unsafe_convert(::Type{Ptr{Cvoid}}, model::Optimizer) = model.ptr
```

Implement methods for `Optimizer`

All `Optimizers` must implement the following methods:

- `empty!`
- `is_empty`
- `optimize!`

Other methods, detailed below, are optional or depend on how you implement the interface.

> **Tip**
>
> For this and all future methods, read the docstrings to understand what each method does, what it expects as input, and what it produces as output. If it isn't clear, let us know and we will improve the docstrings! It is also very helpful to look at an existing wrapper for a similar solver.

You should also implement `Base.show(::IO, ::Optimizer)` to print a nice string when someone prints your model. For example

```
function Base.show(io::IO, model::Optimizer)
    return print(io, "NewSolver with the pointer $(model.ptr)")
end
```

Implement attributes

MathOptInterface uses attributes to manage different aspects of the problem.

For each attribute

- `get` gets the current value of the attribute
- `set` sets a new value of the attribute. Not all attributes can be set. For example, the user can't modify the `SolverName`.
- `supports` returns a `Bool` indicating whether the solver supports the attribute.

Info

Use `attribute_value_type` to check the value expected by a given attribute. You should make sure that your `get` function correctly infers to this type (or a subtype of it).

Each column in the table indicates whether you need to implement the particular method for each attribute.

Attribute	`get`	`set`	`supports`
`SolverName`	Yes	No	No
`SolverVersion`	Yes	No	No
`RawSolver`	Yes	No	No
`Name`	Yes	Yes	Yes
`Silent`	Yes	Yes	Yes
`TimeLimitSec`	Yes	Yes	Yes
`RawOptimizerAttribute`	Yes	Yes	Yes
`NumberOfThreads`	Yes	Yes	Yes

For example:

```
function MOI.get(model::Optimizer, ::MOI.Silent)
    return # true if MOI.Silent is set
end

function MOI.set(model::Optimizer, ::MOI.Silent, v::Bool)
    if v
        # Set a parameter to turn off printing
    else
        # Restore the default printing
    end
    return
end

MOI.supports(::Optimizer, ::MOI.Silent) = true
```

Define `supports_constraint`

The next step is to define which constraints and objective functions you plan to support.

For each function-set constraint pair, define `supports_constraint`:

```
function MOI.supports_constraint(
    ::Optimizer,
    ::Type{MOI.VariableIndex},
    ::Type{MOI.ZeroOne},
)
    return true
end
```

To make this easier, you may want to use `Union`s:

```
function MOI.supports_constraint(
    ::Optimizer,
    ::Type{MOI.VariableIndex},
```

```
    ::Type{<:Union{MOI.LessThan,MOI.GreaterThan,MOI.EqualTo}},
)
    return true
end
```

> **Tip**
>
> Only support a constraint if your solver has native support for it.

The big decision: copy-to or incremental modifications?

Now you need to decide whether to support incremental modification or not.

Incremental modification means that the user can add variables and constraints one-by-one without needing to rebuild the entire problem, and they can modify the problem data after an `optimize!` call. Supporting incremental modification means implementing functions like `add_variable` and `add_constraint`.

The alternative is to accept the problem data in a single `copy_to` function call, afterwhich it cannot be modified. Because `copy_to` sees all of the data at once, it can typically call a more efficient function to load data into the underlying solver.

Good examples of solvers supporting incremental modification are MILP solvers like GLPK.jl and Gurobi.jl. Examples of `copy_to` solvers are AmplNLWriter.jl and SCS.jl

It is possible to implement both approaches, but you should probably start with one for simplicity.

> **Tip**
>
> Only support incremental modification if your solver has native support for it.

In general, supporting incremental modification is more work, and it usually requires some extra book-keeping. However, it provides a more efficient interface to the solver if the problem is going to be resolved multiple times with small modifications. Moreover, once you've implemented incremental modification, it's usually not much extra work to add a `copy_to` interface. The converse is not true.

> **Tip**
>
> If this is your first time writing an interface, start with `copy_to`.

The `copy_to` interface

To implement the `copy_to` interface, implement the following function:

- `copy_to`

The incremental interface

> **Warning**
>
> Writing this interface is a lot of work. The easiest way is to consult the source code of a similar solver!

To implement the incremental interface, implement the following functions:

- `add_variable`

- `add_variables`
- `add_constraint`
- `add_constraints`
- `is_valid`
- `delete`

Info

Solvers do not have to support `AbstractScalarFunction` in `GreaterThan`, `LessThan`, `EqualTo`, or `Interval` with a nonzero constant in the function. Throw `ScalarFunctionConstantNotZero` if the function constant is not zero.

In addition, you should implement the following model attributes:

Attribute	`get`	`set`	`supports`
`ListOfModelAttributesSet`	Yes	No	No
`ObjectiveFunctionType`	Yes	No	No
`ObjectiveFunction`	Yes	Yes	Yes
`ObjectiveSense`	Yes	Yes	Yes
`Name`	Yes	Yes	Yes

Variable-related attributes:

Attribute	`get`	`set`	`supports`
`ListOfVariableAttributesSet`	Yes	No	No
`NumberOfVariables`	Yes	No	No
`ListOfVariableIndices`	Yes	No	No

Constraint-related attributes:

Attribute	`get`	`set`	`supports`
`ListOfConstraintAttributesSet`	Yes	No	No
`NumberOfConstraints`	Yes	No	No
`ListOfConstraintTypesPresent`	Yes	No	No
`ConstraintFunction`	Yes	Yes	No
`ConstraintSet`	Yes	Yes	No

Modifications If your solver supports modifying data in-place, implement `modify` for the following `AbstractModifications`:

- `ScalarConstantChange`
- `ScalarCoefficientChange`
- `VectorConstantChange`
- `MultirowChange`

Variables constrained on creation Some solvers require variables be associated with a set when they are created. This conflicts with the incremental modification approach, since you cannot first add a free variable and then constrain it to the set.

If this is the case, implement:

- `add_constrained_variable`
- `add_constrained_variables`
- `supports_add_constrained_variables`

By default, MathOptInterface assumes solvers support free variables. If your solver does not support free variables, define:

```
MOI.supports_add_constrained_variables(::Optimizer, ::Type{Reals}) = false
```

Incremental and `copy_to`

If you implement the incremental interface, you have the option of also implementing `copy_to`.

If you don't want to implement `copy_to`, e.g., because the solver has no API for building the problem in a single function call, define the following fallback:

```
MOI.supports_incremental_interface(::Optimizer) = true

function MOI.copy_to(dest::Optimizer, src::MOI.ModelLike)
    return MOI.Utilities.default_copy_to(dest, src)
end
```

Names

Regardless of which interface you implement, you have the option of implementing the `Name` attribute for variables and constraints:

Attribute	`get`	`set`	`supports`
`VariableName`	Yes	Yes	Yes
`ConstraintName`	Yes	Yes	Yes

If you implement names, you must also implement the following three methods:

```
function MOI.get(model::Optimizer, ::Type{MOI.VariableIndex}, name::String)
    return # The variable named `name`.
end

function MOI.get(model::Optimizer, ::Type{MOI.ConstraintIndex}, name::String)
    return # The constraint any type named `name`.
end

function MOI.get(
    model::Optimizer,
    ::Type{MOI.ConstraintIndex{F,S}},
    name::String,
) where {F,S}
    return # The constraint of type F-in-S named `name`.
end
```

These methods have the following rules:

- If there is no variable or constraint with the name, return `nothing`
- If there is a single variable or constraint with that name, return the variable or constraint
- If there are multiple variables or constraints with the name, throw an error.

Warning

You should not implement `ConstraintName` for `VariableIndex` constraints. If you implement `ConstraintName` for other constraints, you can add the following two methods to disable `ConstraintName` for `VariableIndex` constraints.

```
function MOI.supports(
    ::Optimizer,
    ::MOI.ConstraintName,
    ::Type{<:MOI.ConstraintIndex{MOI.VariableIndex,<:MOI.AbstractScalarSet}},
)
    return throw(MOI.VariableIndexConstraintNameError())
end
function MOI.set(
    ::Optimizer,
    ::MOI.ConstraintName,
    ::MOI.ConstraintIndex{MOI.VariableIndex,<:MOI.AbstractScalarSet},
    ::String,
)
    return throw(MOI.VariableIndexConstraintNameError())
end
```

Solutions

Implement `optimize!` to solve the model:

- `optimize!`

All `Optimizers` must implement the following attributes:

- `DualStatus`
- `PrimalStatus`
- `RawStatusString`
- `ResultCount`
- `TerminationStatus`

Info

You only need to implement `get` for solution attributes. Don't implement `set` or `supports`.

Note

Solver wrappers should document how the low-level statuses map to the MOI statuses. Statuses like `NEARLY_FEASIBLE_POINT` and `INFEASIBLE_POINT`, are designed to be used when the solver explicitly indicates that relaxed tolerances are satisfied or the returned point is infeasible, respectively.

You should also implement the following attributes:

- `ObjectiveValue`
- `SolveTimeSec`
- `VariablePrimal`

Tip

Attributes like `VariablePrimal` and `ObjectiveValue` are indexed by the result count. Use `MOI.check_result_index_bound attr)` to throw an error if the attribute is not available.

If your solver returns dual solutions, implement:

- `ConstraintDual`
- `DualObjectiveValue`

For integer solvers, implement:

- `ObjectiveBound`
- `RelativeGap`

If applicable, implement:

- `SimplexIterations`
- `BarrierIterations`
- `NodeCount`

If your solver uses the Simplex method, implement:

- `ConstraintBasisStatus`

If your solver accepts primal or dual warm-starts, implement:

- `VariablePrimalStart`
- `ConstraintDualStart`

Other tips

Unsupported constraints at runtime

In some cases, your solver may support a particular type of constraint (e.g., quadratic constraints), but only if the data meets some condition (e.g., it is convex).

In this case, declare that you `supports` the constraint, and throw `AddConstraintNotAllowed`.

Dealing with multiple variable bounds

MathOptInterface uses `VariableIndex` constraints to represent variable bounds. Defining multiple variable bounds on a single variable is not allowed.

Throw `LowerBoundAlreadySet` or `UpperBoundAlreadySet` if the user adds a constraint that results in multiple bounds.

Only throw if the constraints conflict. It is okay to add `VariableIndex`-in-`GreaterThan` and then `VariableIndex`-in-`LessThan`, but not `VariableIndex`-in-`Interval` and then `VariableIndex`-in-`LessThan`,

Expect duplicate coefficients

Solvers should expect that functions such as `ScalarAffineFunction` and `VectorQuadraticFunction` may contain duplicate coefficents.

For example, `ScalarAffineFunction([ScalarAffineTerm(x, 1), ScalarAffineTerm(x, 1)], 0.0)`.

Use `Utilities.canonical` to return a new function with the duplicate coefficients aggregated together.

Don't modify user-data

All data passed to the solver should be copied immediately to internal data structures. Solvers may not modify any input vectors and should assume that input vectors may be modified by users in the future.

This applies, for example, to the `terms` vector in `ScalarAffineFunction`. Vectors returned to the user, e.g., via `ObjectiveFunction` or `ConstraintFunction` attributes, should not be modified by the solver afterwards. The in-place version of `get!` can be used by users to avoid extra copies in this case.

Column Generation

There is no special interface for column generation. If the solver has a special API for setting coefficients in existing constraints when adding a new variable, it is possible to queue modifications and new variables and then call the solver's API once all of the new coefficients are known.

Extra: solver-specific attributes

You don't need to restrict yourself to the attributes defined in the MathOptInterface.jl package.

Solver-specific attributes should be specified by creating an appropriate subtype of `AbstractModelAttribute`, `AbstractOptimizerAttribute`, `AbstractVariableAttribute`, or `AbstractConstraintAttribute`.

For example, Gurobi.jl adds attributes for multiobjective optimization by defining:

```
struct NumberOfObjectives <: MOI.AbstractModelAttribute end

function MOI.set(model::Optimizer, ::NumberOfObjectives, n::Integer)
    # Code to set NumberOfObjectives
    return
```

```
end

function MOI.get(model::Optimizer, ::NumberOfObjectives)
    n = # Code to get NumberOfObjectives
    return n
end
```

Then, the user can write:

```
model = Gurobi.Optimizer()
MOI.set(model, Gurobi.NumberofObjectives(), 3)
```

37.3 Transitioning from MathProgBase

MathOptInterface is a replacement for MathProgBase.jl. However, it is not a direct replacement.

Transitioning a solver interface

MathOptInterface is more extensive than MathProgBase which may make its implementation seem daunting at first. There are however numerous utilities in MathOptInterface that should hopefully make the implementation as simple or simpler than MathProgBase.

For more information, read Implementing a solver interface.

Transitioning the high-level functions

MathOptInterface doesn't provide replacements for the high-level interfaces in MathProgBase. We recommend you use JuMP as a modeling interface instead.

> **Tip**
>
> If you haven't used JuMP before, start with the tutorial Getting started with JuMP

linprog

Here is one way of transitioning from `linprog`:

```
using JuMP

function linprog(c, A, sense, b, l, u, solver)
    N = length(c)
    model = Model(solver)
    @variable(model, l[i] <= x[i=1:N] <= u[i])
    @objective(model, Min, c' * x)
    eq_rows, ge_rows, le_rows = sense .== '=', sense .== '>', sense .== '<'
    @constraint(model, A[eq_rows, :] * x .== b[eq_rows])
    @constraint(model, A[ge_rows, :] * x .>= b[ge_rows])
    @constraint(model, A[le_rows, :] * x .<= b[le_rows])
    optimize!(model)
    return (
        status = termination_status(model),
        objval = objective_value(model),
        sol = value.(x)
    )
end
```

mixintprog

Here is one way of transitioning from `mixintprog`:

```
using JuMP

function mixintprog(c, A, rowlb, rowub, vartypes, lb, ub, solver)
    N = length(c)
    model = Model(solver)
    @variable(model, lb[i] <= x[i=1:N] <= ub[i])
    for i in 1:N
        if vartypes[i] == :Bin
            set_binary(x[i])
        elseif vartypes[i] == :Int
            set_integer(x[i])
        end
    end
    @objective(model, Min, c' * x)
    @constraint(model, rowlb .<= A * x .<= rowub)
    optimize!(model)
    return (
        status = termination_status(model),
        objval = objective_value(model),
        sol = value.(x)
    )
end
```

quadprog

Here is one way of transitioning from quadprog:

```
using JuMP

function quadprog(c, Q, A, rowlb, rowub, lb, ub, solver)
    N = length(c)
    model = Model(solver)
    @variable(model, lb[i] <= x[i=1:N] <= ub[i])
    @objective(model, Min, c' * x + 0.5 * x' * Q * x)
    @constraint(model, rowlb .<= A * x .<= rowub)
    optimize!(model)
    return (
        status = termination_status(model),
        objval = objective_value(model),
        sol = value.(x)
    )
end
```

37.4 Implementing a constraint bridge

This guide outlines the basic steps to create a new bridge from a constraint expressed in the formalism Function-in-Set.

Preliminaries

First, decide on the set you want to bridge. Then, study its properties: the most important one is whether the set is scalar or vector, which impacts the dimensionality of the functions that can be used with the set.

- A scalar function only has one dimension. MOI defines three types of scalar functions: a variable (`VariableIndex`), an affine function (`ScalarAffineFunction`), or a quadratic function (`ScalarQuadraticFunction`).
- A vector function has several dimensions (at least one). MOI defines three types of vector functions: several variables (`VectorOfVariables`), an affine function (`VectorAffineFunction`), or a quadratic function (`VectorQuadraticFunction`). The main difference with scalar functions is that the order of dimensions can be very important: for instance, in an indicator constraint (`Indicator`), the first dimension indicates whether the constraint about the second dimension is active.

To explain how to implement a bridge, we present the example of `Bridges.Constraint.FlipSignBridge`. This bridge maps <= (`LessThan`) constraints to >= (`GreaterThan`) constraints. This corresponds to reversing the sign of the inequality. We focus on scalar affine functions (we disregard the cases of a single variable or of quadratic functions). This example is a simplified version of the code included in MOI.

Four mandatory parts in a constraint bridge

The first part of a constraint bridge is a new concrete subtype of `Bridges.Constraint.AbstractBridge`. This type must have fields to store all the new variables and constraints that the bridge will add. Typically, these types are parametrized by the type of the coefficients in the model.

Then, three sets of functions must be defined:

1. `Bridges.Constraint.bridge_constraint`: this function implements the bridge and creates the required variables and constraints.
2. `supports_constraint`: these functions should return `true` when the combination of function and set is supported by the bridge. By default, the base implementation always returns `false` and the bridge does not have to provide this implementation.
3. `Bridges.added_constrained_variable_types` and `Bridges.added_constraint_types`: these functions return the types of variables and constraints that this bridge adds. They are used to compute the set of other bridges that are required to use the one you are defining, if need be.

More functions can be implemented, for instance to retrieve properties from the bridge or deleting a bridged constraint.

1. Structure for the bridge

A typical `struct` behind a bridge depends on the type of the coefficients that are used for the model (typically `Float64`, but coefficients might also be integers or complex numbers).

This structure must hold a reference to all the variables and the constraints that are created as part of the bridge.

The type of this structure is used throughout MOI as an identifier for the bridge. It is passed as argument to most functions related to bridges.

The best practice is to have the name of this type end with `Bridge`.

In our example, the bridge should be able to map any `ScalarAffineFunction{T}`-in-`LessThan{T}` constraint to a single `ScalarAffineFunction{T}`-in-`GreaterThan{T}` constraint. The affine function has coefficients of type T. The bridge is parametrized with T, so that the constraint that the bridge creates also has coefficients of type T.

```
struct SignBridge{T<:Number} <: Bridges.Constraint.AbstractBridge
    constraint::ConstraintIndex{ScalarAffineFunction{T}, GreaterThan{T}}
end
```

2. Bridge creation

The function `Bridges.Constraint.bridge_constraint` is called whenever the bridge should be instantiated for a specific model, with the given function and set. The arguments to `bridge_constraint` are similar to `add_constraint`, with the exception of the first argument: it is the Type of the struct defined in the first step (for our example, `Type{SignBridge{T}}`).

`bridge_constraint` returns an instance of the struct defined in the first step. the first step.

In our example, the bridge constraint could be defined as:

```
function Bridges.Constraint.bridge_constraint(
    ::Type{SignBridge{T}}, # Bridge to use.
    model::ModelLike, # Model to which the constraint is being added.
    f::ScalarAffineFunction{T}, # Function to rewrite.
    s::LessThan{T}, # Set to rewrite.
) where {T}
    # Create the variables and constraints required for the bridge.
    con = add_constraint(model, -f, GreaterThan(-s.upper))

    # Return an instance of the bridge type with a reference to all the
    # variables and constraints that were created in this function.
    return SignBridge(con)
end
```

3. Supported constraint types

The function `supports_constraint` determines whether the bridge type supports a given combination of function and set.

This function must closely match `bridge_constraint`, because it will not be called if `supports_constraint` returns `false`.

```
function supports_constraint(
    ::Type{SignBridge{T}}, # Bridge to use.
    ::Type{ScalarAffineFunction{T}}, # Function to rewrite.
    ::Type{LessThan{T}}, # Set to rewrite.
) where {T}
    # Do some computation to ensure that the constraint is supported.
    # Typically, you can directly return true.
    return true
end
```

4. Metadata about the bridge

To determine whether a bridge can be used, MOI uses a shortest-path algorithm that uses the variable types and the constraints that the bridge can create. This information is communicated from the bridge to MOI using

the functions `Bridges.added_constrained_variable_types` and `Bridges.added_constraint_types`. Both return lists of tuples: either a list of 1-tuples containing the variable types (typically, `ZeroOne` or `Integer`) or a list of 2-tuples contained the functions and sets (like `ScalarAffineFunction{T}`-`GreaterThan`).

For our example, the bridge does not create any constrained variables, and only `ScalarAffineFunction{T}`-in-`GreaterThan{T}` constraints:

```
function Bridges.added_constrained_variable_types(::Type{SignBridge{T}}) where {T}
    # The bridge does not create variables, return an empty list of tuples:
    return Tuple{Type}[]
end

function Bridges.added_constraint_types(::Type{SignBridge{T}}) where {T}
    return Tuple{Type,Type}[
        # One element per F-in-S the bridge creates.
        (ScalarAffineFunction{T}, GreaterThan{T}),
    ]
end
```

A bridge that creates binary variables would rather have this definition of `added_constrained_variable_types`:

```
function Bridges.added_constrained_variable_types(::Type{SomeBridge{T}}) where {T}
    # The bridge only creates binary variables:
    return Tuple{Type}[(ZeroOne,)]
end
```

> **Warning**
>
> If you declare the creation of constrained variables in `added_constrained_variable_types`, the corresponding constraint type `VariableIndex` should not be indicated in `added_constraint_types`. This would restrict the use of the bridge to solvers that can add such a constraint after the variable is created.
>
> More concretely, if you declare in `added_constrained_variable_types` that your bridge creates binary variables (`ZeroOne`), and if you never add such a constraint afterward (you do not call `add_constraint(model, var, ZeroOne())`), then you should not list `(VariableIndex, ZeroOne)` in `added_constraint_types`.

Typically, the function `Bridges.Constraint.concrete_bridge_type` does not have to be defined for most bridges.

Bridge registration

For a bridge to be used by MOI, it must be known by MOI.

`SingleBridgeOptimizer`

The first way to do so is to create a single-bridge optimizer. This type of optimizer wraps another optimizer and adds the possibility to use only one bridge. It is especially useful when unit testing bridges.

It is common practice to use the same name as the type defined for the bridge (`SignBridge`, in our example) without the suffix `Bridge`.

```
const Sign{T,OT<: ModelLike} =
    SingleBridgeOptimizer{SignBridge{T}, OT}
```

In the context of unit tests, this bridge is used in conjunction with a `Utilities.MockOptimizer`:

```
mock = Utilities.MockOptimizer(
    Utilities.UniversalFallback(Utilities.Model{Float64}()),
)
bridged_mock = Sign{Float64}(mock)
```

New bridge for a `LazyBridgeOptimizer`

Typical user-facing models for MOI are based on `Bridges.LazyBridgeOptimizer`. For instance, this type of model is returned by `Bridges.full_bridge_optimizer`. These models can be added more bridges by using `Bridges.add_bridge`:

```
inner_optimizer = Utilities.Model{Float64}()
optimizer = Bridges.full_bridge_optimizer(inner_optimizer, Float64)
Bridges.add_bridge(optimizer, SignBridge{Float64})
```

Bridge improvements

Attribute retrieval

Like models, bridges have attributes that can be retrieved using `get` and `set`. The most important ones are the number of variables and constraints, but also the lists of variables and constraints.

In our example, we only have one constraint and only have to implement the `NumberOfConstraints` and `ListOfConstraintIndices` attributes:

```
function get(
    ::SignBridge{T},
    ::NumberOfConstraints{
        ScalarAffineFunction{T},
        GreaterThan{T},
    },
) where {T}
    return 1
end

function get(
    bridge::SignBridge{T},
    ::ListOfConstraintIndices{
        ScalarAffineFunction{T},
        GreaterThan{T},
    },
) where {T}
    return [bridge.constraint]
end
```

You must implement one such pair of functions for each type of constraint the bridge adds to the model.

> **Warning**
>
> Avoid returning a list from the bridge object without copying it. Users should be able to change the contents of the returned list without altering the bridge object.

For variables, the situation is simpler. If your bridge creates new variables, you should implement the `NumberOfVariables` and `ListOfVariableIndices` attributes. However, these attributes do not have parameters, unlike their constraint counterparts. Only two functions suffice:

```
function get(
    ::SignBridge{T},
    ::NumberOfVariables,
) where {T}
    return 0
end

function get(
    ::SignBridge{T},
    ::ListOfVariableIndices,
) where {T}
    return VariableIndex[]
end
```

Model modifications

To avoid copying the model when the user request to change a constraint, MOI provides `modify`. Bridges can also implement this API to allow certain changes, such as coefficient changes.

In our case, a modification of a coefficient in the original constraint (i.e. replacing the value of the coefficient of a variable in the affine function) should be transmitted to the constraint created by the bridge, but with a sign change.

```
function modify(
    model::ModelLike,
    bridge::SignBridge,
    change::ScalarCoefficientChange,
)
    modify(
        model,
        bridge.constraint,
        ScalarCoefficientChange(change.variable, -change.new_coefficient),
    )
    return
end
```

Bridge deletion

When a bridge is deleted, the constraints it added should be deleted too.

```
function delete(model::ModelLike, bridge::SignBridge)
    delete(model, bridge.constraint)
    return
end
```

37.5 Manipulating expressions

This guide highlights a syntactically appealing way to build expressions at the MOI level, but also to look at their contents. It may be especially useful when writing models or bridge code.

Creating functions

This section details the ways to create functions with MathOptInterface.

Creating scalar affine functions

The simplest scalar function is simply a variable:

```
julia> x = MOI.add_variable(model) # Create the variable x
MathOptInterface.VariableIndex(1)
```

This type of function is extremely simple; to express more complex functions, other types must be used. For instance, a `ScalarAffineFunction` is a sum of linear terms (a factor times a variable) and a constant. Such an object can be built using the standard constructor:

```
julia> f = MOI.ScalarAffineFunction([MOI.ScalarAffineTerm(1, x)], 2) # x + 2
MathOptInterface.ScalarAffineFunction{Int64}(MathOptInterface.ScalarAffineTerm{Int64}[MathOptInterface.ScalarAffineTer
↪  MathOptInterface.VariableIndex(1))], 2)
```

However, you can also use operators to build the same scalar function:

```
julia> f = x + 2
MathOptInterface.ScalarAffineFunction{Int64}(MathOptInterface.ScalarAffineTerm{Int64}[MathOptInterface.ScalarAffineTer
↪  MathOptInterface.VariableIndex(1))], 2)
```

Creating scalar quadratic functions

Scalar quadratic functions are stored in `ScalarQuadraticFunction` objects, in a way that is highly similar to scalar affine functions. You can obtain a quadratic function as a product of affine functions:

```
julia> 1 * x * x
MathOptInterface.ScalarQuadraticFunction{Int64}(MathOptInterface.ScalarQuadraticTerm{Int64}[MathOptInterface.ScalarQua
↪  MathOptInterface.VariableIndex(1), MathOptInterface.VariableIndex(1))],
↪  MathOptInterface.ScalarAffineTerm{Int64}[], 0)

julia> f * f  # (x + 2)²
MathOptInterface.ScalarQuadraticFunction{Int64}(MathOptInterface.ScalarQuadraticTerm{Int64}[MathOptInterface.ScalarQua
↪  MathOptInterface.VariableIndex(1), MathOptInterface.VariableIndex(1))],
↪  MathOptInterface.ScalarAffineTerm{Int64}[MathOptInterface.ScalarAffineTerm{Int64}(2,
↪  MathOptInterface.VariableIndex(1)), MathOptInterface.ScalarAffineTerm{Int64}(2,
↪  MathOptInterface.VariableIndex(1))], 4)

julia> f^2  # (x + 2)² too
MathOptInterface.ScalarQuadraticFunction{Int64}(MathOptInterface.ScalarQuadraticTerm{Int64}[MathOptInterface.ScalarQua
↪  MathOptInterface.VariableIndex(1), MathOptInterface.VariableIndex(1))],
↪  MathOptInterface.ScalarAffineTerm{Int64}[MathOptInterface.ScalarAffineTerm{Int64}(2,
↪  MathOptInterface.VariableIndex(1)), MathOptInterface.ScalarAffineTerm{Int64}(2,
↪  MathOptInterface.VariableIndex(1))], 4)
```

Creating vector functions

A vector function is a function with several values, irrespective of the number of input variables. Similarly to scalar functions, there are three main types of vector functions: `VectorOfVariables`, `VectorAffineFunction`, and `VectorQuadraticFunction`.

The easiest way to create a vector function is to stack several scalar functions using `Utilities.vectorize`. It takes a vector as input, and the generated vector function (of the most appropriate type) has each dimension corresponding to a dimension of the vector.

```
julia> g = MOI.Utilities.vectorize([f, 2 * f])
MathOptInterface.VectorAffineFunction{Int64}(MathOptInterface.VectorAffineTerm{Int64}[MathOptInterface.VectorAffineTer
↪  MathOptInterface.ScalarAffineTerm{Int64}(1, MathOptInterface.VariableIndex(1))),
↪  MathOptInterface.VectorAffineTerm{Int64}(2, MathOptInterface.ScalarAffineTerm{Int64}(2,
↪  MathOptInterface.VariableIndex(1)))], [2, 4])
```

> **Warning**
>
> `Utilities.vectorize` only takes a vector of similar scalar functions: you cannot mix `VariableIndex` and `ScalarAffineFunction`, for instance. In practice, it means that `Utilities.vectorize([x, f])` does not work; you should rather use `Utilities.vectorize([1 * x, f])` instead to only have `ScalarAffineFunction` objects.

Canonicalizing functions

In more advanced use cases, you might need to ensure that a function is "canonical". Functions are stored as an array of terms, but there is no check that these terms are redundant: a `ScalarAffineFunction` object might have two terms with the same variable, like `x + x + 1`. These terms could be merged without changing the semantics of the function: `2x + 1`.

Working with these objects might be cumbersome. Canonicalization helps maintain redundancy to zero.

`Utilities.is_canonical` checks whether a function is already in its canonical form:

```
julia> MOI.Utilities.is_canonical(f + f) # (x + 2) + (x + 2) is stored as x + x + 4
false
```

`Utilities.canonical` returns the equivalent canonical version of the function:

```
julia> MOI.Utilities.canonical(f + f) # Returns 2x + 4
MathOptInterface.ScalarAffineFunction{Int64}(MathOptInterface.ScalarAffineTerm{Int64}[MathOptInterface.ScalarAffineTer
↪  MathOptInterface.VariableIndex(1))], 4)
```

Exploring functions

At some point, you might need to dig into a function, for instance to map it into solver constructs.

Vector functions

`Utilities.scalarize` returns a vector of scalar functions from a vector function:

```
julia> MOI.Utilities.scalarize(g) # Returns a vector [f, 2 * f].
2-element Vector{MathOptInterface.ScalarAffineFunction{Int64}}:
```

```
↪ MathOptInterface.ScalarAffineFunction{Int64}(MathOptInterface.ScalarAffineTerm{Int64}[MathOptInterface.ScalarAffin
↪ MathOptInterface.VariableIndex(1))], 2)

↪ MathOptInterface.ScalarAffineFunction{Int64}(MathOptInterface.ScalarAffineTerm{Int64}[MathOptInterface.ScalarAffin
↪ MathOptInterface.VariableIndex(1))], 4)
```

Note

`Utilities.eachscalar` returns an iterator on the dimensions, which serves the same purpose as `Utilities.scalarize`.

`output_dimension` returns the number of dimensions of the output of a function:

```
julia> MOI.output_dimension(g)
2
```

37.6 Latency

MathOptInterface suffers the "time-to-first-solve" problem of start-up latency.

This hurts both the user- and developer-experience of MathOptInterface. In the first case, because simple models have a multi-second delay before solving, and in the latter, because our tests take so long to run!

This page contains some advice on profiling and fixing latency-related problems in the MathOptInterface.jl repository.

Background

Before reading this part of the documentation, you should familiarize yourself with the reasons for latency in Julia and how to fix them.

- Read the blogposts on julialang.org on precompilation and SnoopCompile
- Read the SnoopCompile documentation.
- Watch Tim Holy's talk at JuliaCon 2021
- Watch the package development workshop at JuliaCon 2021

Causes

There are three main causes of latency in MathOptInterface:

1. A large number of types
2. Lack of method ownership
3. Type-instability in the bridge layer

A large number of types

Julia is very good at specializing method calls based on the input type. Each specialization has a compilation cost, but the benefit of faster run-time performance.

The best-case scenario is for a method to be called a large number of times with a single set of argument types. The worst-case scenario is for a method to be called a single time for a large set of argument types.

Because of MathOptInterface's function-in-set formulation, we fall into the worst-case situation.

This is a fundamental limitation of Julia, so there isn't much we can do about it. However, if we can precompile MathOptInterface, much of the cost can be shifted from start-up latency to the time it takes to precompile a package on installation.

However, there are two things which make MathOptInterface hard to precompile...

Lack of method ownership

Lack of method ownership happens when a call is made using a mix of structs and methods from different modules. Because of this, no single module "owns" the method that is being dispatched, and so it cannot be precompiled.

> **Tip**
>
> This is a slightly simplified explanation. Read the precompilation tutorial for a more in-depth discussion on back-edges.

Unfortunately, the design of MOI means that this is a frequent occurrence! We have a bunch of types in `MOI.Utilities` that wrap types defined in external packages (i.e., the `Optimizers`), which implement methods of functions defined in `MOI` (e.g., `add_variable`, `add_constraint`).

Here's a simple example of method-ownership in practice:

```
module MyMOI
struct Wrapper{T}
    inner::T
end
optimize!(x::Wrapper) = optimize!(x.inner)
end  # MyMOI

module MyOptimizer
using ..MyMOI
struct Optimizer end
MyMOI.optimize!(x::Optimizer) = 1
end  # MyOptimizer

using SnoopCompile
model = MyMOI.Wrapper(MyOptimizer.Optimizer())

julia> tinf = @snoopi_deep MyMOI.optimize!(model)
InferenceTimingNode: 0.008256/0.008543 on InferenceFrameInfo for Core.Compiler.Timings.ROOT() with 1
↪  direct children
```

The result is that there was one method that required type inference. If we visualize `tinf`:

```
using ProfileView
ProfileView.view(flamegraph(tinf))
```

we see a flamegraph with a large red-bar indicating that the method `MyMOI.optimize(MyMOI.Wrapper{MyOptimizer.Optimizer}` cannot be precompiled.

To fix this, we need to designate a module to "own" that method (i.e., create a back-edge). The easiest way to do this is for `MyOptimizer` to call `MyMOI.optimize(MyMOI.Wrapper{MyOptimizer.Optimizer})` during `using MyOptimizer`. Let's see that in practice:

```
module MyMOI
struct Wrapper{T}
    inner::T
end
optimize(x::Wrapper) = optimize(x.inner)
end  # MyMOI

module MyOptimizer
using ..MyMOI
struct Optimizer end
MyMOI.optimize(x::Optimizer) = 1
# The syntax of this let-while loop is very particular:
#  * `let ... end` keeps everything local to avoid polluting the MyOptimizer
#    namespace
#  * `while true ... break end` runs the code once, and forces Julia to compile
#    the inner loop, rather than interpret it.
let
    while true
        model = MyMOI.Wrapper(Optimizer())
        MyMOI.optimize(model)
        break
    end
end
end  # MyOptimizer

using SnoopCompile
model = MyMOI.Wrapper(MyOptimizer.Optimizer())

julia> tinf = @snoopi_deep MyMOI.optimize(model)
InferenceTimingNode: 0.006822/0.006822 on InferenceFrameInfo for Core.Compiler.Timings.ROOT() with 0
↪  direct children
```

There are now `0` direct children that required type inference because the method was already stored in `MyOptimizer`!

Unfortunately, this trick only works if the call-chain is fully inferrable. If there are breaks (due to type instability), then the benefit of doing this is reduced. And unfortunately for us, the design of MathOptInterface has a lot of type instabilities...

Type instability in the bridge layer

Most of MathOptInterface is pretty good at ensuring type-stability. However, a key component is not type stable, and that is the bridging layer.

In particular, the bridging layer defines `Bridges.LazyBridgeOptimizer`, which has fields like:

```
struct LazyBridgeOptimizer
    constraint_bridge_types::Vector{Any}
    constraint_node::Dict{Tuple{Type,Type},ConstraintNode}
    constraint_types::Vector{Tuple{Type,Type}}
end
```

This is because the `LazyBridgeOptimizer` needs to be able to deal with any function-in-set type passed to it, and we also allow users to pass additional bridges that they defined in external packages.

So to recap, MathOptInterface suffers package latency because:

1. there are a large number of types and functions...
2. and these are split between multiple modules, including external packages...
3. and there are type-instabilities like those in the bridging layer.

Resolutions

There are no magic solutions to reduce latency. Issue #1313 tracks progress on reducing latency in MathOptInterface.

A useful script is the following (replace GLPK as needed):

```
using MathOptInterface, GLPK
const MOI = MathOptInterface

function example_diet(optimizer, bridge)
    category_data = [
        1800.0 2200.0;
          91.0    Inf;
           0.0   65.0;
           0.0 1779.0
    ]
    cost = [2.49, 2.89, 1.50, 1.89, 2.09, 1.99, 2.49, 0.89, 1.59]
    food_data = [
        410 24 26 730;
        420 32 10 1190;
        560 20 32 1800;
        380  4 19 270;
        320 12 10 930;
        320 15 12 820;
        320 31 12 1230;
        100  8 2.5 125;
        330  8 10 180
    ]
    bridge_model = if bridge
        MOI.instantiate(optimizer; with_bridge_type=Float64)
    else
        MOI.instantiate(optimizer)
    end
    model = MOI.Utilities.CachingOptimizer(
        MOI.Utilities.UniversalFallback(MOI.Utilities.Model{Float64}()),
        MOI.Utilities.AUTOMATIC,
    )
    MOI.Utilities.reset_optimizer(model, bridge_model)
    MOI.set(model, MOI.Silent(), true)
    nutrition = MOI.add_variables(model, size(category_data, 1))
    for (i, v) in enumerate(nutrition)
        MOI.add_constraint(model, v, MOI.GreaterThan(category_data[i, 1]))
        MOI.add_constraint(model, v, MOI.LessThan(category_data[i, 2]))
    end
```

```
    buy = MOI.add_variables(model, size(food_data, 1))
    MOI.add_constraint.(model, buy, MOI.GreaterThan(0.0))
    MOI.set(model, MOI.ObjectiveSense(), MOI.MIN_SENSE)
    f = MOI.ScalarAffineFunction(MOI.ScalarAffineTerm.(cost, buy), 0.0)
    MOI.set(model, MOI.ObjectiveFunction{typeof(f)}(), f)
    for (j, n) in enumerate(nutrition)
        f = MOI.ScalarAffineFunction(
            MOI.ScalarAffineTerm.(food_data[:, j], buy),
            0.0,
        )
        push!(f.terms, MOI.ScalarAffineTerm(-1.0, n))
        MOI.add_constraint(model, f, MOI.EqualTo(0.0))
    end
    MOI.optimize!(model)
    term_status = MOI.get(model, MOI.TerminationStatus())
    @assert term_status == MOI.OPTIMAL
    MOI.add_constraint(
        model,
        MOI.ScalarAffineFunction(
            MOI.ScalarAffineTerm.(1.0, [buy[end-1], buy[end]]),
            0.0,
        ),
        MOI.LessThan(6.0),
    )
    MOI.optimize!(model)
    @assert MOI.get(model, MOI.TerminationStatus()) == MOI.INFEASIBLE
    return
end

if length(ARGS) > 0
    bridge = get(ARGS, 2, "") != "--no-bridge"
    println("Running: $(ARGS[1]) $(get(ARGS, 2, ""))")
    @time example_diet(GLPK.Optimizer, bridge)
    @time example_diet(GLPK.Optimizer, bridge)
    exit(0)
end
```

You can create a flame-graph via

```
using SnoopComile
tinf = @snoopi_deep example_diet(GLPK.Optimizer, true)
using ProfileView
ProfileView.view(flamegraph(tinf))
```

Here's how things looked in mid-August 2021:

There are a few opportunities for improvement (non-red flames, particularly on the right). But the main problem is a large red (non-precompilable due to method ownership) flame.

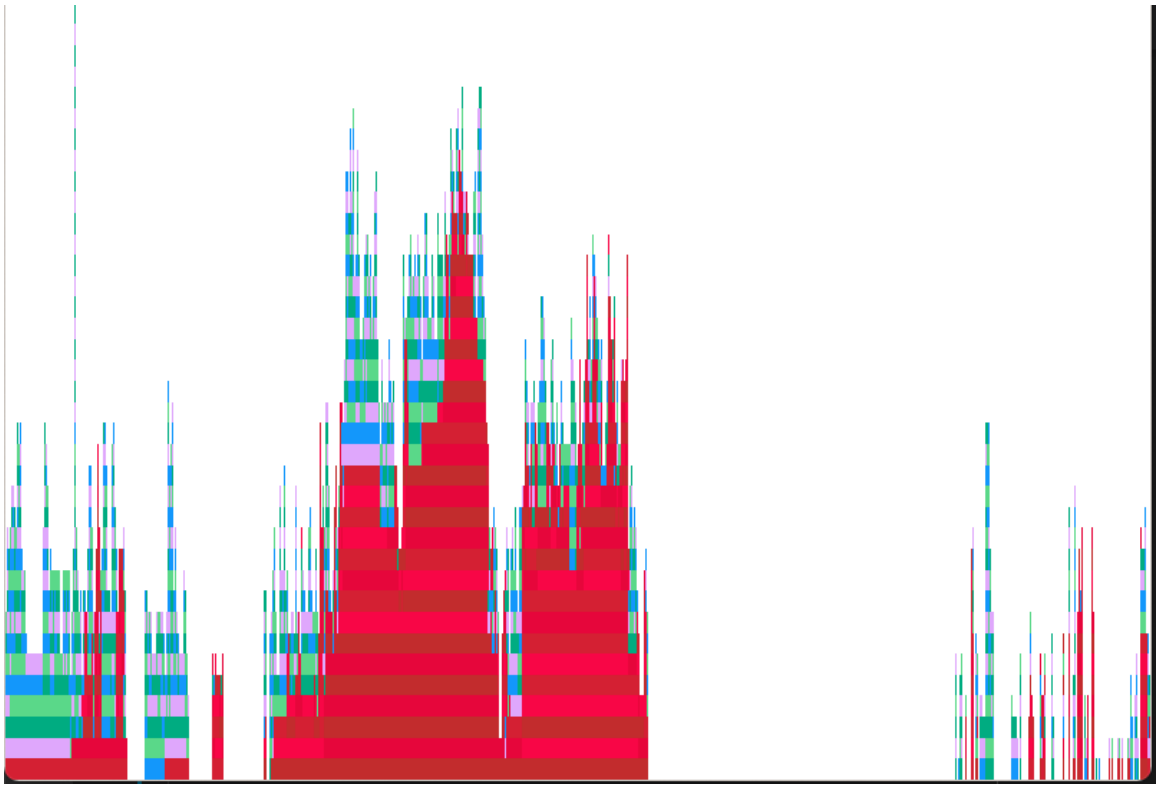


Figure 37.1: flamegraph

Chapter 38

Manual

38.1 Standard form problem

MathOptInterface represents optimization problems in the standard form:

$$\min_{x \in \mathbb{R}^n} \quad f_0(x) \tag{38.1}$$

$$\text{s.t.} \quad f_i(x) \in \mathcal{S}_i \qquad i = 1 \dots m \tag{38.2}$$

where:

- the functions $f_0, f_1, \dots, f_m$ are specified by `AbstractFunction` objects
- the sets $\mathcal{S}_1, \dots, \mathcal{S}_m$ are specified by `AbstractSet` objects

> **Tip**
>
> For more information on this standard form, read our paper.

MOI defines some commonly used functions and sets, but the interface is extensible to other sets recognized by the solver.

Functions

The function types implemented in MathOptInterface.jl are:

- `VariableIndex`: x_j, i.e., projection onto a single coordinate defined by a variable index j.
- `VectorOfVariables`: projection onto multiple coordinates (i.e., extracting a subvector).
- `ScalarAffineFunction`: $a^T x + b$, where a is a vector and b scalar.
- `VectorAffineFunction`: $Ax + b$, where A is a matrix and b is a vector.
- `ScalarQuadraticFunction`: $\frac{1}{2} x^T Q x + a^T x + b$, where Q is a symmetric matrix, a is a vector, and b is a constant.
- `VectorQuadraticFunction`: a vector of scalar-valued quadratic functions.

Extensions for nonlinear programming are present but not yet well documented.

One-dimensional sets

The one-dimensional set types implemented in MathOptInterface.jl are:

- `LessThan(upper)`: $\{x \in \mathbb{R} : x \leq \text{upper}\}$
- `GreaterThan(lower)`: $\{x \in \mathbb{R} : x \geq \text{lower}\}$
- `EqualTo(value)`: $\{x \in \mathbb{R} : x = \text{value}\}$
- `Interval(lower, upper)`: $\{x \in \mathbb{R} : x \in [\text{lower}, \text{upper}]\}$
- `Integer()`: $\mathbb{Z}$
- `ZeroOne()`: $\{0, 1\}$
- `Semicontinuous(lower,upper)`: $\{0\} \cup [\text{lower}, \text{upper}]$
- `Semiinteger(lower,upper)`: $\{0\} \cup \{\text{lower}, \text{lower} + 1, \ldots, \text{upper} - 1, \text{upper}\}$

Vector cones

The vector-valued set types implemented in MathOptInterface.jl are:

- `Reals(dimension)`: $\mathbb{R}^{\text{dimension}}$
- `Zeros(dimension)`: $0^{\text{dimension}}$
- `Nonnegatives(dimension)`: $\{x \in \mathbb{R}^{\text{dimension}} : x \geq 0\}$
- `Nonpositives(dimension)`: $\{x \in \mathbb{R}^{\text{dimension}} : x \leq 0\}$
- `SecondOrderCone(dimension)`: $\{(t, x) \in \mathbb{R}^{\text{dimension}} : t \geq \|x\|_2\}$
- `RotatedSecondOrderCone(dimension)`: $\{(t, u, x) \in \mathbb{R}^{\text{dimension}} : 2tu \geq \|x\|_2^2, t, u \geq 0\}$
- `ExponentialCone()`: $\{(x, y, z) \in \mathbb{R}^3 : y \exp(x/y) \leq z, y > 0\}$
- `DualExponentialCone()`: $\{(u, v, w) \in \mathbb{R}^3 : -u \exp(v/u) \leq \exp(1)w, u < 0\}$
- `GeometricMeanCone(dimension)`: $\{(t, x) \in \mathbb{R}^{n+1} : x \geq 0, t \leq \sqrt[n]{x_1 x_2 \cdots x_n}\}$ where n is dimension-1
- `PowerCone(exponent)`: $\{(x, y, z) \in \mathbb{R}^3 : x^{\text{exponent}} y^{1-\text{exponent}} \geq |z|, x, y \geq 0\}$
- `DualPowerCone(exponent)`: $\{(u, v, w) \in \mathbb{R}^3 : \frac{u}{\text{exponent}}^{\text{exponent}} \frac{v}{1-\text{exponent}}^{1-\text{exponent}} \geq |w|, u, v \geq 0\}$
- `NormOneCone(dimension)`: $\{(t, x) \in \mathbb{R}^{\text{dimension}} : t \geq \|x\|_1\}$ where $\|x\|_1 = \sum_i |x_i|$
- `NormInfinityCone(dimension)`: $\{(t, x) \in \mathbb{R}^{\text{dimension}} : t \geq \|x\|_\infty\}$ where $\|x\|_\infty = \max_i |x_i|$.
- `RelativeEntropyCone(dimension)`: $\{(u, v, w) \in \mathbb{R}^{\text{dimension}} : u \geq \sum_i w_i \log(\frac{w_i}{v_i}), v_i \geq 0, w_i \geq 0\}$

Matrix cones

The matrix-valued set types implemented in MathOptInterface.jl are:

- `RootDetConeTriangle(dimension)`: $\{(t, X) \in \mathbb{R}^{1+\text{dimension}(1+\text{dimension})/2} : t \leq \det(X)^{1/\text{dimension}}, X$ is the upper
- `RootDetConeSquare(dimension)`: $\{(t, X) \in \mathbb{R}^{1+\text{dimension}^2} : t \leq \det(X)^{1/\text{dimension}}, X$ is a PSD matrix$\}$
- `PositiveSemidefiniteConeTriangle(dimension)`: $\{X \in \mathbb{R}^{\text{dimension}(\text{dimension}+1)/2} : X$ is the upper triangle of a PSI
- `PositiveSemidefiniteConeSquare(dimension)`: $\{X \in \mathbb{R}^{\text{dimension}^2} : X$ is a PSD matrix$\}$
- `LogDetConeTriangle(dimension)`: $\{(t, u, X) \in \mathbb{R}^{2+\text{dimension}(1+\text{dimension})/2} : t \leq u \log(\det(X/u)), X$ is the upper t $0\}$
- `LogDetConeSquare(dimension)`: $\{(t, u, X) \in \mathbb{R}^{2+\text{dimension}^2} : t \leq u \log(\det(X/u)), X$ is a PSD matrix, $u > 0\}$
- `NormSpectralCone(row_dim, column_dim)`: $\{(t, X) \in \mathbb{R}^{1+\text{row\_dim}\times\text{column\_dim}} : t \geq \sigma_1(X), X$ is a matrix with row_c
- `NormNuclearCone(row_dim, column_dim)`: $\{(t, X) \in \mathbb{R}^{1+\text{row\_dim}\times\text{column\_dim}} : t \geq \sum_i \sigma_i(X), X$ is a matrix with row

Some of these cones can take two forms: `XXXConeTriangle` and `XXXConeSquare`.

In `XXXConeTriangle` sets, the matrix is assumed to be symmetric, and the elements are provided by a vector, in which the entries of the upper-right triangular part of the matrix are given column by column (or equivalently, the entries of the lower-left triangular part are given row by row).

In `XXXConeSquare` sets, the entries of the matrix are given column by column (or equivalently, row by row), and the matrix is constrained to be symmetric. As an example, given a 2-by-2 matrix of variables `X` and a one-dimensional variable `t`, we can specify a root-det constraint as `[t, X11, X12, X22] ∈ RootDetConeTriangle` or `[t, X11, X12, X21, X22] ∈ RootDetConeSquare`.

We provide both forms to enable flexibility for solvers who may natively support one or the other. Transformations between `XXXConeTriangle` and `XXXConeSquare` are handled by bridges, which removes the chance of conversion mistakes by users or solver developers.

Multi-dimensional sets with combinatorial structure

- `SOS1(weights)`: A special ordered set of Type I.
- `SOS2(weights)`: A special ordered set of Type II.
- `Indicator(set)`: A set to specify indicator constraints.
- `Complements(dimension)`: A set for mixed complementarity constraints.

38.2 Models

The most significant part of MOI is the definition of the **model API** that is used to specify an instance of an optimization problem (e.g., by adding variables and constraints). Objects that implement the model API should inherit from the `ModelLike` abstract type.

Notably missing from the model API is the method to solve an optimization problem. `ModelLike` objects may store an instance (e.g., in memory or backed by a file format) without being linked to a particular solver. In addition to the model API, MOI defines `AbstractOptimizer` and provides methods to solve the model and interact with solutions. See the Solutions section for more details.

Info

Throughout the rest of the manual, `model` is used as a generic `ModelLike`, and `optimizer` is used as a generic `AbstractOptimizer`.

Tip

MOI does not export functions, but for brevity we often omit qualifying names with the MOI module. Best practice is to have

```
using MathOptInterface
const MOI = MathOptInterface
```

and prefix all MOI methods with `MOI.` in user code. If a name is also available in base Julia, we always explicitly use the module prefix, for example, with `MOI.get`.

Attributes

Attributes are properties of the model that can be queried and modified. These include constants such as the number of variables in a model `NumberOfVariables`), and properties of variables and constraints such as the name of a variable (`VariableName`).

There are four types of attributes:

- Model attributes (subtypes of `AbstractModelAttribute`) refer to properties of a model.
- Optimizer attributes (subtypes of `AbstractOptimizerAttribute`) refer to properties of an optimizer.
- Constraint attributes (subtypes of `AbstractConstraintAttribute`) refer to properties of an individual constraint.
- Variable attributes (subtypes of `AbstractVariableAttribute`) refer to properties of an individual variable.

Some attributes are values that can be queried by the user but not modified, while other attributes can be modified by the user.

All interactions with attributes occur through the `get` and `set` functions.

Consult the docstsrings of each attribute for information on what it represents.

ModelLike API

The following attributes are available:

- `ListOfConstraintAttributesSet`
- `ListOfConstraintIndices`
- `ListOfConstraintTypesPresent`
- `ListOfModelAttributesSet`
- `ListOfVariableAttributesSet`
- `ListOfVariableIndices`

- NumberOfConstraints
- NumberOfVariables
- Name
- ObjectiveFunction
- ObjectiveFunctionType
- ObjectiveSense

AbstractOptimizer API

The following attributes are available:

- DualStatus
- PrimalStatus
- RawStatusString
- ResultCount
- TerminationStatus
- BarrierIterations
- DualObjectiveValue
- NodeCount
- NumberOfThreads
- ObjectiveBound
- ObjectiveValue
- RelativeGap
- RawOptimizerAttribute
- RawSolver
- Silent
- SimplexIterations
- SolverName
- SolverVersion
- SolveTimeSec
- TimeLimitSec

38.3 Variables

Add a variable

Use `add_variable` to add a single variable.

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

`add_variable` returns a `VariableIndex` type, which should be used to refer to the added variable in other calls.

Check if a `VariableIndex` is valid using `is_valid`.

```
julia> MOI.is_valid(model, x)
true
```

Use `add_variables` to add a number of variables.

```
julia> y = MOI.add_variables(model, 2)
2-element Vector{MathOptInterface.VariableIndex}:
 MathOptInterface.VariableIndex(2)
 MathOptInterface.VariableIndex(3)
```

> **Warning**
>
> The integer does not necessarily corresond to the column inside an optimizer!

Delete a variable

Delete a variable using `delete`.

```
julia> MOI.delete(model, x)

julia> MOI.is_valid(model, x)
false
```

> **Warning**
>
> Not all `ModelLike` models support deleting variables. A `DeleteNotAllowed` error is thrown if this is not supported.

Variable attributes

The following attributes are available for variables:

- `VariableName`
- `VariablePrimalStart`
- `VariablePrimal`

Get and set these attributes using `get` and `set`.

```
julia> MOI.set(model, MOI.VariableName(), x, "var_x")

julia> MOI.get(model, MOI.VariableName(), x)
"var_x"
```

38.4 Constraints

Add a constraint

Use `add_constraint` to add a single constraint.

```
julia> c = MOI.add_constraint(model, MOI.VectorOfVariables(x), MOI.Nonnegatives(2))
MathOptInterface.ConstraintIndex{MathOptInterface.VectorOfVariables,
↪  MathOptInterface.Nonnegatives}(1)
```

`add_constraint` returns a `ConstraintIndex` type, which should be used to refer to the added constraint in other calls.

Check if a `ConstraintIndex` is valid using `is_valid`.

```
julia> MOI.is_valid(model, c)
true
```

Use `add_constraints` to add a number of constraints of the same type.

```
julia> c = MOI.add_constraints(
           model,
           [x[1], x[2]],
           [MOI.GreaterThan(0.0), MOI.GreaterThan(1.0)]
       )
2-element Vector{MathOptInterface.ConstraintIndex{MathOptInterface.VariableIndex,
↪  MathOptInterface.GreaterThan{Float64}}}:
 MathOptInterface.ConstraintIndex{MathOptInterface.VariableIndex,
↪  MathOptInterface.GreaterThan{Float64}}(1)
 MathOptInterface.ConstraintIndex{MathOptInterface.VariableIndex,
↪  MathOptInterface.GreaterThan{Float64}}(2)
```

This time, a vector of `ConstraintIndex` are returned.

Use `supports_constraint` to check if the model supports adding a constraint type.

```
julia> MOI.supports_constraint(
           model,
           MOI.VariableIndex,
           MOI.GreaterThan{Float64},
       )
true
```

Delete a constraint

Use `delete` to delete a constraint.

```
julia> MOI.delete(model, c)

julia> MOI.is_valid(model, c)
false
```

Constraint attributes

The following attributes are available for constraints:

- `ConstraintName`
- `ConstraintPrimalStart`
- `ConstraintDualStart`
- `ConstraintPrimal`
- `ConstraintDual`
- `ConstraintBasisStatus`
- `ConstraintFunction`
- `CanonicalConstraintFunction`
- `ConstraintSet`

Get and set these attributes using `get` and `set`.

```
julia> MOI.set(model, MOI.ConstraintName(), c, "con_c")

julia> MOI.get(model, MOI.ConstraintName(), c)
"con_c"
```

Constraints by function-set pairs

Below is a list of common constraint types and how they are represented as function-set pairs in MOI. In the notation below, x is a vector of decision variables, x_i is a scalar decision variable, α, β are scalar constants, a, b are constant vectors, A is a constant matrix and $\mathbb{R}_+$ (resp. $\mathbb{R}_-$) is the set of nonnegative (resp. nonpositive) real numbers.

Linear constraints

Mathematical Constraint	MOI Function	MOI Set
$a^T x \leq \beta$	`ScalarAffineFunction`	`LessThan`
$a^T x \geq \alpha$	`ScalarAffineFunction`	`GreaterThan`
$a^T x = \beta$	`ScalarAffineFunction`	`EqualTo`
$\alpha \leq a^T x \leq \beta$	`ScalarAffineFunction`	`Interval`
$x_i \leq \beta$	`VariableIndex`	`LessThan`
$x_i \geq \alpha$	`VariableIndex`	`GreaterThan`
$x_i = \beta$	`VariableIndex`	`EqualTo`
$\alpha \leq x_i \leq \beta$	`VariableIndex`	`Interval`
$Ax + b \in \mathbb{R}^n_+$	`VectorAffineFunction`	`Nonnegatives`
$Ax + b \in \mathbb{R}^n_-$	`VectorAffineFunction`	`Nonpositives`
$Ax + b = 0$	`VectorAffineFunction`	`Zeros`

By convention, solvers are not expected to support nonzero constant terms in the `ScalarAffineFunction`s the first four rows above, because they are redundant with the parameters of the sets. For example, $2x + 1 \leq 2$ should be encoded as $2x \leq 1$.

Constraints with `VariableIndex` in `LessThan`, `GreaterThan`, `EqualTo`, or `Interval` sets have a natural interpretation as variable bounds. As such, it is typically not natural to impose multiple lower- or upper-bounds on the same variable, and the solver interfaces should throw respectively `LowerBoundAlreadySet` or `UpperBoundAlreadySet`.

Moreover, adding two `VariableIndex` constraints on the same variable with the same set is impossible because they share the same index as it is the index of the variable, see `ConstraintIndex`.

It is natural, however, to impose upper- and lower-bounds separately as two different constraints on a single variable. The difference between imposing bounds by using a single `Interval` constraint and by using separate `LessThan` and `GreaterThan` constraints is that the latter will allow the solver to return separate dual multipliers for the two bounds, while the former will allow the solver to return only a single dual for the interval constraint.

Conic constraints

Mathematical Constraint	MOI Function	MOI Set
$\|Ax+b\|_2 \leq c^T x + d$	`VectorAffineFunction`	`SecondOrderCone`
$y \geq \|x\|_2$	`VectorOfVariables`	`SecondOrderCone`
$2yz \geq \|x\|_2^2, y, z \geq 0$	`VectorOfVariables`	`RotatedSecondOrderCone`
$(a_1^T x + b_1, a_2^T x + b_2, a_3^T x + b_3) \in \mathcal{E}$	`VectorAffineFunction`	`ExponentialCone`
$A(x) \in \mathcal{S}_+$	`VectorAffineFunction`	`PositiveSemidefiniteConeTriangle`
$B(x) \in \mathcal{S}_+$	`VectorAffineFunction`	`PositiveSemidefiniteConeSquare`
$x \in \mathcal{S}_+$	`VectorOfVariables`	`PositiveSemidefiniteConeTriangle`
$x \in \mathcal{S}_+$	`VectorOfVariables`	`PositiveSemidefiniteConeSquare`

where $\mathcal{E}$ is the exponential cone (see `ExponentialCone`), $\mathcal{S}_+$ is the set of positive semidefinite symmetric matrices, A is an affine map that outputs symmetric matrices and B is an affine map that outputs square matrices.

Quadratic constraints

Mathematical Constraint	MOI Function	MOI Set
$x^T Q x + a^T x + b \geq 0$	`ScalarQuadraticFunction`	`GreaterThan`
$x^T Q x + a^T x + b \leq 0$	`ScalarQuadraticFunction`	`LessThan`
$x^T Q x + a^T x + b = 0$	`ScalarQuadraticFunction`	`EqualTo`
Bilinear matrix inequality	`VectorQuadraticFunction`	`PositiveSemidefiniteCone...`

Discrete and logical constraints

Mathematical Constraint	MOI Function	MOI Set
$x_i \in \mathbb{Z}$	`VariableIndex`	`Integer`
$x_i \in \{0, 1\}$	`VariableIndex`	`ZeroOne`
$x_i \in \{0\} \cup [l, u]$	`VariableIndex`	`Semicontinuous`
$x_i \in \{0\} \cup \{l, l+1, \ldots, u-1, u\}$	`VariableIndex`	`Semiinteger`
At most one component of x can be nonzero	`VectorOfVariables`	`SOS1`
At most two components of x can be nonzero, and if so they must be adjacent components	`VectorOfVariables`	`SOS2`
$y = 1 \implies a^T x \in S$	`VectorAffineFunction`	`Indicator`

JuMP mapping

The following bullet points show examples of how JuMP constraints are translated into MOI function-set pairs:

- `@constraint(m, 2x + y <= 10)` becomes `ScalarAffineFunction-in-LessThan`
- `@constraint(m, 2x + y >= 10)` becomes `ScalarAffineFunction-in-GreaterThan`
- `@constraint(m, 2x + y == 10)` becomes `ScalarAffineFunction-in-EqualTo`
- `@constraint(m, 0 <= 2x + y <= 10)` becomes `ScalarAffineFunction-in-Interval`
- `@constraint(m, 2x + y in ArbitrarySet())` becomes `ScalarAffineFunction-in-ArbitrarySet`.

Variable bounds are handled in a similar fashion:

- `@variable(m, x <= 1)` becomes `VariableIndex-in-LessThan`
- `@variable(m, x >= 1)` becomes `VariableIndex-in-GreaterThan`

One notable difference is that a variable with an upper and lower bound is translated into two constraints, rather than an interval. i.e.:

- `@variable(m, 0 <= x <= 1)` becomes `VariableIndex-in-LessThan` and `VariableIndex-in-GreaterThan`.

38.5 Solutions

Solving and retrieving the results

Once an optimizer is loaded with the objective function and all of the constraints, we can ask the solver to solve the model by calling `optimize!`.

```
MOI.optimize!(optimizer)
```

Why did the solver stop?

The optimization procedure may terminate for a number of reasons. The `TerminationStatus` attribute of the optimizer returns a `TerminationStatusCode` object which explains why the solver stopped.

The termination statuses distinguish between proofs of optimality, infeasibility, local convergence, limits, and termination because of something unexpected like invalid problem data or failure to converge.

A typical usage of the `TerminationStatus` attribute is as follows:

```
status = MOI.get(optimizer, TerminationStatus())
if status == MOI.OPTIMAL
    # Ok, we solved the problem!
else
    # Handle other cases.
end
```

After checking the `TerminationStatus`, one should typically check `ResultCount`. This attribute returns the number of results that the solver has available to return. A result is defined as a primal-dual pair, but either the primal or the dual may be missing from the result. While the `OPTIMAL` termination status normally implies that at least one result is available, other statuses do not. For example, in the case of infeasiblity, a solver may return no result or a proof of infeasibility. The `ResultCount` attribute distinguishes between these two cases.

Primal solutions

Use the `PrimalStatus` optimizer attribute to return a `ResultStatusCode` describing the status of the primal solution.

Common returns are described below in the Common status situations section.

Query the primal solution using the `VariablePrimal` and `ConstraintPrimal` attributes.

Query the objective function value using the `ObjectiveValue` attribute.

Dual solutions

> **Warning**
>
> See Duality for a discussion of the MOI conventions for primal-dual pairs and certificates.

Use the `DualStatus` optimizer attribute to return a `ResultStatusCode` describing the status of the dual solution.

Query the dual solution using the `ConstraintDual` attribute.

Query the dual objective function value using the `DualObjectiveValue` attribute.

Common status situations

The sections below describe how to interpret typical or interesting status cases for three common classes of solvers. The example cases are illustrative, not comprehensive. Solver wrappers may provide additional information on how the solver's statuses map to MOI statuses.

> **Info**
>
> * in the tables indicate that multiple different values are possible.

Primal-dual convex solver

Linear programming and conic optimization solvers fall into this category.

What happened?	TerminationStatus	ResultCount	PrimalStatus	DualStatus
Proved optimality	OPTIMAL	1	FEASIBLE_POINT	FEASIBLE_POINT
Proved infeasible	INFEASIBLE	1	NO_SOLUTION	INFEASIBILITY_CERTIFICATE
Optimal within relaxed tolerances	ALMOST_OPTIMAL	1	FEASIBLE_POINT	FEASIBLE_POINT
Optimal within relaxed tolerances	ALMOST_OPTIMAL	1	ALMOST_FEASIBLE_POINT	ALMOST_FEASIBLE_POINT
Detected an unbounded ray of the primal	DUAL_INFEASIBLE	1	INFEASIBILITY_CERTIFICATE	NO_SOLUTION
Stall	SLOW_PROGRESS	1	*	*

Global branch-and-bound solvers

Mixed-integer programming solvers fall into this category.

> **Info**
>
> `CPXMIP_OPTIMAL_INFEAS` is a CPLEX status that indicates that a preprocessed problem was solved to optimality, but the solver was unable to recover a feasible solution to the original problem. Handling this status was one of the motivating drivers behind the design of MOI.

What happened?	TerminationStatus	ResultCount	PrimalStatus	DualStatus
Proved optimality	OPTIMAL	1	FEASIBLE_POINT	NO_SOLUTION
Presolve detected infeasibility or unboundedness	INFEASIBLE_OR_UNBOUNDED	0	NO_SOLUTION	NO_SOLUTION
Proved infeasibility	INFEASIBLE	0	NO_SOLUTION	NO_SOLUTION
Timed out (no solution)	TIME_LIMIT	0	NO_SOLUTION	NO_SOLUTION
Timed out (with a solution)	TIME_LIMIT	1	FEASIBLE_POINT	NO_SOLUTION
CPXMIP_OPTIMAL_INFEAS	ALMOST_OPTIMAL	1	INFEASIBLE_POINT	NO_SOLUTION

Local search solvers

Nonlinear programming solvers fall into this category. It also includes non-global tree search solvers like Juniper.

What happened?	TerminationStatus	ResultCount	PrimalStatus	DualStatus
Converged to a stationary point	LOCALLY_SOLVED	1	FEASIBLE_POINT	FEASIBLE_POINT
Completed a non-global tree search (with a solution)	LOCALLY_SOLVED	1	FEASIBLE_POINT	FEASIBLE_POINT
Converged to an infeasible point	LOCALLY_INFEASIBLE	1	INFEASIBLE_POINT	*
Completed a non-global tree search (no solution found)	LOCALLY_INFEASIBLE	0	NO_SOLUTION	NO_SOLUTION
Iteration limit	ITERATION_LIMIT	1	*	*
Diverging iterates	NORM_LIMIT or OBJECTIVE_LIMIT	1	*	*

38.6 Problem modification

In addition to adding and deleting constraints and variables, MathOptInterface supports modifying, in-place, coefficients in the constraints and the objective function of a model.

These modifications can be grouped into two categories:

- modifications which replace the set of function of a constraint with a new set or function
- modifications which change, in-place, a component of a function

Warning

Solve `ModelLike` objects do not support problem modification.

Modify the set of a constraint

Use `set` and `ConstraintSet` to modify the set of a constraint by replacing it with a new instance of the same type.

```
julia> c = MOI.add_constraint(
           model,
           MOI.ScalarAffineFunction([MOI.ScalarAffineTerm(1.0, x)], 0.0),
           MOI.EqualTo(1.0),
       )
MathOptInterface.ConstraintIndex{MathOptInterface.ScalarAffineFunction{Float64},
↪  MathOptInterface.EqualTo{Float64}}(1)
```

```
julia> MOI.set(model, MOI.ConstraintSet(), c, MOI.EqualTo(2.0));

julia> MOI.get(model, MOI.ConstraintSet(), c) == MOI.EqualTo(2.0)
true
```

However, the following will fail as the new set is of a different type to the original set:

```
julia> MOI.set(model, MOI.ConstraintSet(), c, MOI.GreaterThan(2.0))
ERROR: [...]
```

Special cases: set transforms

If our constraint is an affine inequality, then this corresponds to modifying the right-hand side of a constraint in linear programming.

In some special cases, solvers may support efficiently changing the set of a constraint (for example, from `LessThan` to `GreaterThan`). For these cases, MathOptInterface provides the `transform` method.

The `transform` function returns a new constraint index, and the old constraint index (i.e., `c`) is no longer valid.

```
julia> c = MOI.add_constraint(
           model,
           MOI.ScalarAffineFunction([MOI.ScalarAffineTerm(1.0, x)], 0.0),
           MOI.LessThan(1.0),
       )
MathOptInterface.ConstraintIndex{MathOptInterface.ScalarAffineFunction{Float64},
↪  MathOptInterface.LessThan{Float64}}(1)

julia> new_c = MOI.transform(model, c, MOI.GreaterThan(2.0));

julia> MOI.is_valid(model, c)
false

julia> MOI.is_valid(model, new_c)
true
```

> **Note**
>
> `transform` cannot be called with a set of the same type. Use `set` instead.

Modify the function of a constraint

Use `set` and `ConstraintFunction` to modify the function of a constraint by replacing it with a new instance of the same type.

```
julia> c = MOI.add_constraint(
           model,
           MOI.ScalarAffineFunction([MOI.ScalarAffineTerm(1.0, x)], 0.0),
           MOI.EqualTo(1.0),
       )
MathOptInterface.ConstraintIndex{MathOptInterface.ScalarAffineFunction{Float64},
↪  MathOptInterface.EqualTo{Float64}}(1)

julia> new_f = MOI.ScalarAffineFunction([MOI.ScalarAffineTerm(2.0, x)], 1.0);
```

```
julia> MOI.set(model, MOI.ConstraintFunction(), c, new_f);

julia> MOI.get(model, MOI.ConstraintFunction(), c) ≈ new_f
true
```

However, the following will fail as the new function is of a different type to the original function:

```
julia> MOI.set(model, MOI.ConstraintFunction(), c, x)
ERROR: [...]
```

Modify constant term in a scalar function

Use `modify` and `ScalarConstantChange` to modify the constant term in a `ScalarAffineFunction` or `ScalarQuadraticFunction`.

```
julia> c = MOI.add_constraint(
           model,
           MOI.ScalarAffineFunction([MOI.ScalarAffineTerm(1.0, x)], 0.0),
           MOI.EqualTo(1.0),
       )
MathOptInterface.ConstraintIndex{MathOptInterface.ScalarAffineFunction{Float64},
↪  MathOptInterface.EqualTo{Float64}}(1)

julia> MOI.modify(model, c, MOI.ScalarConstantChange(1.0));

julia> new_f = MOI.ScalarAffineFunction([MOI.ScalarAffineTerm(1.0, x)], 1.0);

julia> MOI.get(model, MOI.ConstraintFunction(), c) ≈ new_f
true
```

> **Tip**
>
> `ScalarConstantChange` can also be used to modify the objective function by passing an instance of `ObjectiveFunction` instead of the constraint index `c` as we saw above.

```
julia> MOI.set(
           model,
           MOI.ObjectiveFunction{MOI.ScalarAffineFunction{Float64}}(),
           new_f,
       );

julia> MOI.modify(
           model,
           MOI.ObjectiveFunction{MOI.ScalarAffineFunction{Float64}}(),
           MOI.ScalarConstantChange(-1.0)
       );

julia> MOI.get(
           model,
           MOI.ObjectiveFunction{MOI.ScalarAffineFunction{Float64}}(),
       ) ≈ MOI.ScalarAffineFunction([MOI.ScalarAffineTerm(1.0, x)], -1.0)
true
```

Modify constant terms in a vector function

Use `modify` and `VectorConstantChange` to modify the constant vector in a `VectorAffineFunction` or `VectorQuadraticFunction`

```
julia> c = MOI.add_constraint(
           model,
           MOI.VectorAffineFunction([
                   MOI.VectorAffineTerm(1, MOI.ScalarAffineTerm(1.0, x)),
                   MOI.VectorAffineTerm(2, MOI.ScalarAffineTerm(2.0, x)),
               ],
               [0.0, 0.0],
           ),
           MOI.Nonnegatives(2),
       )
MathOptInterface.ConstraintIndex{MathOptInterface.VectorAffineFunction{Float64},
↪  MathOptInterface.Nonnegatives}(1)

julia> MOI.modify(model, c, MOI.VectorConstantChange([3.0, 4.0]));

julia> new_f = MOI.VectorAffineFunction(
           [
        MOI.VectorAffineTerm(1, MOI.ScalarAffineTerm(1.0, x)),
        MOI.VectorAffineTerm(2, MOI.ScalarAffineTerm(2.0, x)),
           ],
           [3.0, 4.0],
       );

julia> MOI.get(model, MOI.ConstraintFunction(), c) ≈ new_f
true
```

Modify affine coefficients in a scalar function

Use `modify` and `ScalarCoefficientChange` to modify the affine coefficient of a `ScalarAffineFunction` or `ScalarQuadraticFunction`.

```
julia> c = MOI.add_constraint(
           model,
           MOI.ScalarAffineFunction([MOI.ScalarAffineTerm(1.0, x)], 0.0),
           MOI.EqualTo(1.0),
       )
MathOptInterface.ConstraintIndex{MathOptInterface.ScalarAffineFunction{Float64},
↪  MathOptInterface.EqualTo{Float64}}(1)

julia> MOI.modify(model, c, MOI.ScalarCoefficientChange(x, 2.0));

julia> new_f = MOI.ScalarAffineFunction([MOI.ScalarAffineTerm(2.0, x)], 0.0);

julia> MOI.get(model, MOI.ConstraintFunction(), c) ≈ new_f
true
```

> **Tip**
>
> `ScalarCoefficientChange` can also be used to modify the objective function by passing an instance of `ObjectiveFunction` instead of the constraint index c as we saw above.

Modify affine coefficients in a vector function

Use `modify` and `MultirowChange` to modify a vector of affine coefficients in a `VectorAffineFunction` or a `VectorQuadraticFunction`.

```
julia> c = MOI.add_constraint(
           model,
           MOI.VectorAffineFunction([
                   MOI.VectorAffineTerm(1, MOI.ScalarAffineTerm(1.0, x)),
                   MOI.VectorAffineTerm(2, MOI.ScalarAffineTerm(2.0, x)),
               ],
               [0.0, 0.0],
           ),
           MOI.Nonnegatives(2),
       )
MathOptInterface.ConstraintIndex{MathOptInterface.VectorAffineFunction{Float64},
↪  MathOptInterface.Nonnegatives}(1)

julia> MOI.modify(model, c, MOI.MultirowChange(x, [(1, 3.0), (2, 4.0)]));

julia> new_f = MOI.VectorAffineFunction(
           [
        MOI.VectorAffineTerm(1, MOI.ScalarAffineTerm(3.0, x)),
        MOI.VectorAffineTerm(2, MOI.ScalarAffineTerm(4.0, x)),
           ],
           [0.0, 0.0],
       );

julia> MOI.get(model, MOI.ConstraintFunction(), c) ≈ new_f
true
```

Chapter 39

API Reference

39.1 Standard form

Functions

MathOptInterface.AbstractFunction – Type.

```
AbstractFunction
```

Abstract supertype for function objects.

MathOptInterface.AbstractScalarFunction – Type.

```
AbstractScalarFunction
```

Abstract supertype for scalar-valued function objects.

MathOptInterface.AbstractVectorFunction – Type.

```
AbstractVectorFunction
```

Abstract supertype for vector-valued function objects.

MathOptInterface.VariableIndex – Type.

```
VariableIndex
```

A type-safe wrapper for `Int64` for use in referencing variables in a model. To allow for deletion, indices need not be consecutive.

MathOptInterface.VectorOfVariables – Type.

```
VectorOfVariables(variables)
```

The function that extracts the vector of variables referenced by `variables`, a `Vector{VariableIndex}`. This function is naturally be used for constraints that apply to groups of variables, such as an "all different" constraint, an indicator constraint, or a complementarity constraint.

MathOptInterface.ScalarAffineTerm – Type.

```
struct ScalarAffineTerm{T}
    coefficient::T
    variable::VariableIndex
end
```

Represents cx_i where c is `coefficient` and x_i is the variable identified by `variable`.

`MathOptInterface.ScalarAffineFunction` – Type.

```
ScalarAffineFunction{T}(terms, constant)
```

The scalar-valued affine function $a^T x + b$, where:

- a is a sparse vector specified by a list of `ScalarAffineTerm` structs.
- b is a scalar specified by `constant::T`

Duplicate variable indices in `terms` are accepted, and the corresponding coefficients are summed together.

`MathOptInterface.VectorAffineTerm` – Type.

```
struct VectorAffineTerm{T}
    output_index::Int64
    scalar_term::ScalarAffineTerm{T}
end
```

A `ScalarAffineTerm` plus its index of the output component of a `VectorAffineFunction` or `VectorQuadraticFunction`. `output_index` can also be interpreted as a row index into a sparse matrix, where the `scalar_term` contains the column index and coefficient.

`MathOptInterface.VectorAffineFunction` – Type.

```
VectorAffineFunction{T}(terms, constants)
```

The vector-valued affine function $Ax + b$, where:

- A is a sparse matrix specified by a list of `VectorAffineTerm` objects.
- b is a vector specified by `constants`

Duplicate indices in the A are accepted, and the corresponding coefficients are summed together.

`MathOptInterface.ScalarQuadraticTerm` – Type.

```
struct ScalarQuadraticTerm{T}
    coefficient::T
    variable_1::VariableIndex
    variable_2::VariableIndex
end
```

Represents $cx_i x_j$ where c is `coefficient`, x_i is the variable identified by `variable_1` and x_j is the variable identified by `variable_2`.

`MathOptInterface.ScalarQuadraticFunction` – Type.

```
ScalarQuadraticFunction{T}(quadratic_terms, affine_terms, constant)
```

The scalar-valued quadratic function $\frac{1}{2}x^T Q x + a^T x + b$, where:

- a is a sparse vector specified by a list of `ScalarAffineTerm` structs.
- b is a scalar specified by `constant`.
- Q is a symmetric matrix specified by a list of `ScalarQuadraticTerm` structs.

Duplicate indices in a or Q are accepted, and the corresponding coefficients are summed together. "Mirrored" indices `(q,r)` and `(r,q)` (where `r` and `q` are `VariableIndexes`) are considered duplicates; only one need be specified.

For example, for two scalar variables y, z, the quadratic expression $yz + y^2$ is represented by the terms `ScalarQuadraticTerm.([1.0, 2.0], [y, y], [z, y])`.

MathOptInterface.VectorQuadraticTerm – Type.

```
struct VectorQuadraticTerm{T}
    output_index::Int64
    scalar_term::ScalarQuadraticTerm{T}
end
```

A ScalarQuadraticTerm plus its index of the output component of a `VectorQuadraticFunction`. Each output component corresponds to a distinct sparse matrix Q_i.

MathOptInterface.VectorQuadraticFunction – Type.

```
VectorQuadraticFunction{T}(quadratic_terms, affine_terms, constants)
```

The vector-valued quadratic function with `i`th component ("output index") defined as $\frac{1}{2}x^T Q_i x + a_i^T x + b_i$, where:

- a_i is a sparse vector specified by the `VectorAffineTerms` with `output_index == i`.
- b_i is a scalar specified by `constants[i]`
- Q_i is a symmetric matrix specified by the `VectorQuadraticTerm` with `output_index == i`.

Duplicate indices in a_i or Q_i are accepted, and the corresponding coefficients are summed together. "Mirrored" indices `(q,r)` and `(r,q)` (where `r` and `q` are `VariableIndexes`) are considered duplicates; only one need be specified.

Utilities

MathOptInterface.output_dimension – Function.

```
output_dimension(f::AbstractFunction)
```

Return 1 if `f` has a scalar output and the number of output components if `f` has a vector output.

MathOptInterface.constant – Method.

```
constant(f::Union{ScalarAffineFunction, ScalarQuadraticFunction})
```

Returns the constant term of the scalar function

MathOptInterface.constant – Method.

```
constant(f::Union{VectorAffineFunction, VectorQuadraticFunction})
```

Returns the vector of constant terms of the vector function

MathOptInterface.constant – Method.

```
constant(f::VariableIndex, ::Type{T}) where {T}
```

The constant term of a `VariableIndex` function is the zero value of the specified type `T`.

`MathOptInterface.constant` – Method.

```
constant(f::VectorOfVariables, ::Type{T}) where {T}
```

The constant term of a `VectorOfVariables` function is a vector of zero values of the specified type `T`.

Sets

`MathOptInterface.AbstractSet` – Type.

```
AbstractSet
```

Abstract supertype for set objects used to encode constraints. A set object should not contain any `VariableIndex` or `ConstraintIndex` as the set is passed unmodifed during `copy_to`.

`MathOptInterface.AbstractScalarSet` – Type.

```
AbstractScalarSet
```

Abstract supertype for subsets of $\mathbb{R}$.

`MathOptInterface.AbstractVectorSet` – Type.

```
AbstractVectorSet
```

Abstract supertype for subsets of $\mathbb{R}^n$ for some n.

Utilities

`MathOptInterface.dimension` – Function.

```
dimension(s::AbstractSet)
```

Return the `output_dimension` that an `AbstractFunction` should have to be used with the set `s`.

Examples

```
julia> dimension(Reals(4))
4

julia> dimension(LessThan(3.0))
1

julia> dimension(PositiveSemidefiniteConeTriangle(2))
3
```

`MathOptInterface.dual_set` – Function.

```
dual_set(s::AbstractSet)
```

Return the dual set of `s`, that is the dual cone of the set. This follows the definition of duality discussed in Duality.

See Dual cone for more information.

If the dual cone is not defined it returns an error.

Examples

```
julia> dual_set(Reals(4))
Zeros(4)

julia> dual_set(SecondOrderCone(5))
SecondOrderCone(5)

julia> dual_set(ExponentialCone())
DualExponentialCone()
```

MathOptInterface.dual_set_type – Function.

```
dual_set_type(S::Type{<:AbstractSet})
```

Return the type of dual set of sets of type S, as returned by dual_set. If the dual cone is not defined it returns an error.

Examples

```
julia> dual_set_type(Reals)
Zeros

julia> dual_set_type(SecondOrderCone)
SecondOrderCone

julia> dual_set_type(ExponentialCone)
DualExponentialCone
```

MathOptInterface.constant – Method.

```
constant(s::Union{EqualTo, GreaterThan, LessThan})
```

Returns the constant of the set.

MathOptInterface.supports_dimension_update – Function.

```
supports_dimension_update(S::Type{<:MOI.AbstractVectorSet})
```

Return a Bool indicating whether the elimination of any dimension of n-dimensional sets of type S give an n-1-dimensional set S. By default, this function returns false so it should only be implemented for sets that supports dimension update.

For instance, supports_dimension_update(MOI.Nonnegatives} is true because the elimination of any dimension of the n-dimensional nonnegative orthant gives the n-1-dimensional nonnegative orthant. However supports_dimension_update(MOI.ExponentialCone} is false.

MathOptInterface.update_dimension – Function.

```
update_dimension(s::AbstractVectorSet, new_dim)
```

Returns a set with the dimension modified to new_dim.

Scalar sets

List of recognized scalar sets.

MathOptInterface.GreaterThan – Type.

```
GreaterThan{T <: Real}(lower::T)
```

The set $[lower, \infty) \subseteq \mathbb{R}$.

MathOptInterface.LessThan – Type.

```
LessThan{T <: Real}(upper::T)
```

The set $(-\infty, upper] \subseteq \mathbb{R}$.

MathOptInterface.EqualTo – Type.

```
EqualTo{T <: Number}(value::T)
```

The set containing the single point $x \in \mathbb{R}$ where x is given by `value`.

MathOptInterface.Interval – Type.

```
Interval{T <: Real}(lower::T,upper::T)
```

The interval $[lower, upper] \subseteq \mathbb{R}$. If `lower` or `upper` is `-Inf` or `Inf`, respectively, the set is interpreted as a one-sided interval.

```
Interval(s::GreaterThan{<:AbstractFloat})
```

Construct a (right-unbounded) `Interval` equivalent to the given `GreaterThan` set.

```
Interval(s::LessThan{<:AbstractFloat})
```

Construct a (left-unbounded) `Interval` equivalent to the given `LessThan` set.

```
Interval(s::EqualTo{<:Real})
```

Construct a (degenerate) `Interval` equivalent to the given `EqualTo` set.

MathOptInterface.Integer – Type.

```
Integer()
```

The set of integers $\mathbb{Z}$.

MathOptInterface.ZeroOne – Type.

```
ZeroOne()
```

The set $\{0, 1\}$.

MathOptInterface.Semicontinuous – Type.

```
Semicontinuous{T <: Real}(lower::T,upper::T)
```

The set $\{0\} \cup [lower, upper]$.

MathOptInterface.Semiinteger – Type.

```
Semiinteger{T <: Real}(lower::T,upper::T)
```

The set $\{0\} \cup \{lower, lower + 1, \ldots, upper - 1, upper\}$.

Vector sets

List of recognized vector sets.

`MathOptInterface.Reals` – Type.

```
Reals(dimension)
```

The set $\mathbb{R}^{dimension}$ (containing all points) of dimension `dimension`.

`MathOptInterface.Zeros` – Type.

```
Zeros(dimension)
```

The set $\{0\}^{dimension}$ (containing only the origin) of dimension `dimension`.

`MathOptInterface.Nonnegatives` – Type.

```
Nonnegatives(dimension)
```

The nonnegative orthant $\{x \in \mathbb{R}^{dimension} : x \geq 0\}$ of dimension `dimension`.

`MathOptInterface.Nonpositives` – Type.

```
Nonpositives(dimension)
```

The nonpositive orthant $\{x \in \mathbb{R}^{dimension} : x \leq 0\}$ of dimension `dimension`.

`MathOptInterface.NormInfinityCone` – Type.

```
NormInfinityCone(dimension)
```

The ℓ_∞-norm cone $\{(t, x) \in \mathbb{R}^{dimension} : t \geq \|x\|_\infty = \max_i |x_i|\}$ of dimension `dimension`.

`MathOptInterface.NormOneCone` – Type.

```
NormOneCone(dimension)
```

The ℓ_1-norm cone $\{(t, x) \in \mathbb{R}^{dimension} : t \geq \|x\|_1 = \sum_i |x_i|\}$ of dimension `dimension`.

`MathOptInterface.SecondOrderCone` – Type.

```
SecondOrderCone(dimension)
```

The second-order cone (or Lorenz cone or ℓ_2-norm cone) $\{(t, x) \in \mathbb{R}^{dimension} : t \geq \|x\|_2\}$ of dimension `dimension`.

`MathOptInterface.RotatedSecondOrderCone` – Type.

```
RotatedSecondOrderCone(dimension)
```

The rotated second-order cone $\{(t, u, x) \in \mathbb{R}^{dimension} : 2tu \geq \|x\|_2^2, t, u \geq 0\}$ of dimension `dimension`.

`MathOptInterface.GeometricMeanCone` – Type.

```
GeometricMeanCone(dimension)
```

The geometric mean cone $\{(t, x) \in \mathbb{R}^{n+1} : x \geq 0, t \leq \sqrt[n]{x_1 x_2 \cdots x_n}\}$, where `dimension = n + 1 >= 2`.

Duality note

The dual of the geometric mean cone is $\{(u, v) \in \mathbb{R}^{n+1} : u \leq 0, v \geq 0, -u \leq n \sqrt[n]{\prod_i v_i}\}$, where `dimension = n + 1 >= 2`.

`MathOptInterface.ExponentialCone` – Type.

```
ExponentialCone()
```

The 3-dimensional exponential cone $\{(x, y, z) \in \mathbb{R}^3 : y \exp(x/y) \leq z, y > 0\}$.

`MathOptInterface.DualExponentialCone` – Type.

```
DualExponentialCone()
```

The 3-dimensional dual exponential cone $\{(u, v, w) \in \mathbb{R}^3 : -u \exp(v/u) \leq \exp(1) w, u < 0\}$.

`MathOptInterface.PowerCone` – Type.

```
PowerCone{T <: Real}(exponent::T)
```

The 3-dimensional power cone $\{(x, y, z) \in \mathbb{R}^3 : x^{exponent} y^{1-exponent} \geq |z|, x \geq 0, y \geq 0\}$ with parameter `exponent`.

`MathOptInterface.DualPowerCone` – Type.

```
DualPowerCone{T <: Real}(exponent::T)
```

The 3-dimensional power cone $\{(u, v, w) \in \mathbb{R}^3 : (\frac{u}{exponent})^{exponent} (\frac{v}{1-exponent})^{1-exponent} \geq |w|, u \geq 0, v \geq 0\}$ with parameter `exponent`.

`MathOptInterface.RelativeEntropyCone` – Type.

```
RelativeEntropyCone(dimension)
```

The relative entropy cone $\{(u, v, w) \in \mathbb{R}^{1+2n} : u \geq \sum_{i=1}^{n} w_i \log(\frac{w_i}{v_i}), v_i \geq 0, w_i \geq 0\}$, where `dimension = 2n + 1 >= 1`.

Duality note

The dual of the relative entropy cone is $\{(u, v, w) \in \mathbb{R}^{1+2n} : \forall i, w_i \geq u(\log(\frac{u}{v_i}) - 1), v_i \geq 0, u > 0\}$ of dimension `dimension` $= 2n + 1$.

`MathOptInterface.NormSpectralCone` – Type.

```
NormSpectralCone(row_dim, column_dim)
```

The epigraph of the matrix spectral norm (maximum singular value function) $\{(t, X) \in \mathbb{R}^{1+row_dim \times column_dim} : t \geq \sigma_1(X)\}$, where σ_i is the ith singular value of the matrix X of row dimension `row_dim` and column dimension `column_dim`.

The matrix X is vectorized by stacking the columns, matching the behavior of Julia's vec function.

`MathOptInterface.NormNuclearCone` – Type.

```
NormNuclearCone(row_dim, column_dim)
```

The epigraph of the matrix nuclear norm (sum of singular values function) $\{(t, X) \in \mathbb{R}^{1+row_dim \times column_dim} : t \geq \sum_i \sigma_i(X)\}$, where σ_i is the ith singular value of the matrix X of row dimension `row_dim` and column dimension `column_dim`.

The matrix X is vectorized by stacking the columns, matching the behavior of Julia's vec function.

`MathOptInterface.SOS1` – Type.

```
SOS1{T <: Real}(weights::Vector{T})
```

The set corresponding to the special ordered set (SOS) constraint of type 1. Of the variables in the set, at most one can be nonzero. The `weights` induce an ordering of the variables; as such, they should be unique values. The kth element in the set corresponds to the kth weight in `weights`. See here for a description of SOS constraints and their potential uses.

`MathOptInterface.SOS2` – Type.

```
SOS2{T <: Real}(weights::Vector{T})
```

The set corresponding to the special ordered set (SOS) constraint of type 2. Of the variables in the set, at most two can be nonzero, and if two are nonzero, they must be adjacent in the ordering of the set. The `weights` induce an ordering of the variables; as such, they should be unique values. The kth element in the set corresponds to the kth weight in `weights`. See here for a description of SOS constraints and their potential uses.

`MathOptInterface.Indicator` – Type.

```
Indicator{A<:ActivationCondition,S<:AbstractScalarSet}(set::S)
```

The set corresponding to an indicator constraint.

When `A` is `ACTIVATE_ON_ZERO`, this means: $\{(y, x) \in \{0, 1\} \times \mathbb{R}^n : y = 0 \implies x \in set\}$

When `A` is `ACTIVATE_ON_ONE`, this means: $\{(y, x) \in \{0, 1\} \times \mathbb{R}^n : y = 1 \implies x \in set\}$

Notes

Most solvers expect that the first row of the function is interpretable as a variable index `x_i` (e.g., `1.0 * x + 0.0`). An error will be thrown if this is not the case.

Example

The constraint $\{(y, x) \in \{0, 1\} \times \mathbb{R}^2 : y = 1 \implies x_1 + x_2 \leq 9\}$ is defined as

```
f = MOI.VectorAffineFunction(
    [
        MOI.VectorAffineTerm(1, MOI.ScalarAffineTerm(1.0, y)),
        MOI.VectorAffineTerm(2, MOI.ScalarAffineTerm(1.0, x1)),
        MOI.VectorAffineTerm(2, MOI.ScalarAffineTerm(1.0, x2)),
    ],
    [0.0, 0.0],
)
s = MOI.Indicator{MOI.ACTIVATE_ON_ONE}(MOI.LessThan(9.0))
MOI.add_constraint(model, f, s)
```

`MathOptInterface.Complements` – Type.

```
Complements(dimension::Base.Integer)
```

The set corresponding to a mixed complementarity constraint.

Complementarity constraints should be specified with an `AbstractVectorFunction`-in-`Complements(dimension)` constraint.

The dimension of the vector-valued function `F` must be `dimension`. This defines a complementarity constraint between the scalar function `F[i]` and the variable in `F[i + dimension/2]`. Thus, `F[i + dimension/2]` must be interpretable as a single variable `x_i` (e.g., `1.0 * x + 0.0`), and `dimension` must be even.

The mixed complementarity problem consists of finding `x_i` in the interval `[lb, ub]` (i.e., in the set `Interval(lb, ub)`), such that the following holds:

1. `F_i(x) == 0` if `lb_i < x_i < ub_i`
2. `F_i(x) >= 0` if `lb_i == x_i`
3. `F_i(x) <= 0` if `x_i == ub_i`

Classically, the bounding set for `x_i` is `Interval(0, Inf)`, which recovers: `0 <= F_i(x) ⊥ x_i >= 0`, where the ⊥ operator implies `F_i(x) * x_i = 0`.

Examples

The problem:

```
x -in- Interval(-1, 1)
[-4 * x - 3, x] -in- Complements(2)
```

defines the mixed complementarity problem where the following holds:

1. `-4 * x - 3 == 0` if `-1 < x < 1`
2. `-4 * x - 3 >= 0` if `x == -1`
3. `-4 * x - 3 <= 0` if `x == 1`

There are three solutions:

1. `x = -3/4` with `F(x) = 0`
2. `x = -1` with `F(x) = 1`
3. `x = 1` with `F(x) = -7`

The function `F` can also be defined in terms of single variables. For example, the problem:

```
[x_3, x_4] -in- Nonnegatives(2)
[x_1, x_2, x_3, x_4] -in- Complements(4)
```

defines the complementarity problem where `0 <= x_1 ⊥ x_3 >= 0` and `0 <= x_2 ⊥ x_4 >= 0`.

Matrix sets

Matrix sets are vectorized in order to be subtypes of `AbstractVectorSet`. For sets of symmetric matrices, storing both the `(i, j)` and `(j, i)` elements is redundant so there exists the `AbstractSymmetricMatrixSetTriangle` set to represent only the vectorization of the upper triangular part of the matrix. When the matrix of expressions constrained to be in the set is not symmetric and hence the `(i, j)` and `(j, i)` elements should be constrained to be symmetric, the `AbstractSymmetricMatrixSetSquare` set can be used. The `Bridges.Constraint.SquareBridge` can transform a set from the square form to the `triangular_form` by adding appropriate constraints if the `(i, j)` and `(j, i)` expressions are different.

`MathOptInterface.AbstractSymmetricMatrixSetTriangle` – Type.

```
abstract type AbstractSymmetricMatrixSetTriangle <: AbstractVectorSet end
```

Abstract supertype for subsets of the (vectorized) cone of symmetric matrices, with `side_dimension` rows and columns. The entries of the upper-right triangular part of the matrix are given column by column (or equivalently, the entries of the lower-left triangular part are given row by row). A vectorized cone of `dimension` n corresponds to a square matrix with side dimension $\sqrt{1/4 + 2n} - 1/2$. (Because a $d \times d$ matrix has $d(d+1)/2$ elements in the upper or lower triangle.)

Examples

The matrix

$$\begin{bmatrix} 1 & 2 & 4 \\ 2 & 3 & 5 \\ 4 & 5 & 6 \end{bmatrix}$$

has `side_dimension` 3 and vectorization $(1, 2, 3, 4, 5, 6)$.

Note

Two packed storage formats exist for symmetric matrices, the respective orders of the entries are:

- upper triangular column by column (or lower triangular row by row);
- lower triangular column by column (or upper triangular row by row).

The advantage of the first format is the mapping between the `(i, j)` matrix indices and the k index of the vectorized form. It is simpler and does not depend on the side dimension of the matrix. Indeed,

- the entry of matrix indices `(i, j)` has vectorized index `k = div((j - 1) * j, 2) + i` if $i \leq j$ and `k = div((i - 1) * i, 2) + j` if $j \leq i$;
- and the entry with vectorized index k has matrix indices `i = div(1 + isqrt(8k - 7), 2)` and `j = k - div((i - 1) * i, 2)` or `j = div(1 + isqrt(8k - 7), 2)` and `i = k - div((j - 1) * j, 2)`.

Duality note

The scalar product for the symmetric matrix in its vectorized form is the sum of the pairwise product of the diagonal entries plus twice the sum of the pairwise product of the upper diagonal entries; see [p. 634, 1]. This has important consequence for duality.

Consider for example the following problem (`PositiveSemidefiniteConeTriangle` is a subtype of `AbstractSymmetricMatri`

$$\begin{aligned} \max_{x \in \mathbb{R}} \quad & x \\ \text{s.t.} \quad & (1, -x, 1) \in \text{PositiveSemidefiniteConeTriangle}(2). \end{aligned}$$

The dual is the following problem

$$\begin{aligned} \min_{x \in \mathbb{R}^3} \quad & y_1 + y_3 \\ \text{s.t.} \quad & 2y_2 = 1 \\ & y \in \text{PositiveSemidefiniteConeTriangle}(2). \end{aligned}$$

Why do we use $2y_2$ in the dual constraint instead of y_2 ? The reason is that $2y_2$ is the scalar product between y and the symmetric matrix whose vectorized form is $(0, 1, 0)$. Indeed, with our modified scalar products we have

$$\langle (0,1,0), (y_1, y_2, y_3) \rangle = \text{trace} \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} y_1 & y_2 \\ y_2 & y_3 \end{pmatrix} = 2y_2.$$

References

[1] Boyd, S. and Vandenberghe, L.. Convex optimization. Cambridge university press, 2004.

MathOptInterface.AbstractSymmetricMatrixSetSquare – Type.

```
abstract type AbstractSymmetricMatrixSetSquare <: AbstractVectorSet end
```

Abstract supertype for subsets of the (vectorized) cone of symmetric matrices, with side_dimension rows and columns. The entries of the matrix are given column by column (or equivalently, row by row). The matrix is both constrained to be symmetric and to have its triangular_form belong to the corresponding set. That is, if the functions in entries (i, j) and (j, i) are different, then a constraint will be added to make sure that the entries are equal.

Examples

PositiveSemidefiniteConeSquare is a subtype of AbstractSymmetricMatrixSetSquare and constraining the matrix

$$\begin{bmatrix} 1 & -y \\ -z & 0 \end{bmatrix}$$

to be symmetric positive semidefinite can be achieved by constraining the vector $(1, -z, -y, 0)$ (or $(1, -y, -z, 0)$) to belong to the PositiveSemidefiniteConeSquare(2). It both constrains $y = z$ and $(1, -y, 0)$ (or $(1, -z, 0)$) to be in PositiveSemidefiniteConeTriangle(2), since triangular_form(PositiveSemidefiniteConeSquare) is PositiveSemidefiniteConeTriangle.

MathOptInterface.side_dimension – Function.

```
side_dimension(set::Union{AbstractSymmetricMatrixSetTriangle,
                          AbstractSymmetricMatrixSetSquare})
```

Side dimension of the matrices in set. By convention, it should be stored in the side_dimension field but if it is not the case for a subtype of AbstractSymmetricMatrixSetTriangle, the method should be implemented for this subtype.

MathOptInterface.triangular_form – Function.

```
triangular_form(S::Type{<:AbstractSymmetricMatrixSetSquare})
triangular_form(set::AbstractSymmetricMatrixSetSquare)
```

Return the AbstractSymmetricMatrixSetTriangle corresponding to the vectorization of the upper triangular part of matrices in the AbstractSymmetricMatrixSetSquare set.

List of recognized matrix sets.

MathOptInterface.PositiveSemidefiniteConeTriangle – Type.

```
PositiveSemidefiniteConeTriangle(side_dimension) <: AbstractSymmetricMatrixSetTriangle
```

The (vectorized) cone of symmetric positive semidefinite matrices, with `side_dimension` rows and columns.

See `AbstractSymmetricMatrixSetTriangle` for more details on the vectorized form.

`MathOptInterface.PositiveSemidefiniteConeSquare` – Type.

```
PositiveSemidefiniteConeSquare(side_dimension) <: AbstractSymmetricMatrixSetSquare
```

The cone of symmetric positive semidefinite matrices, with side length `side_dimension`.

See `AbstractSymmetricMatrixSetSquare` for more details on the vectorized form.

The entries of the matrix are given column by column (or equivalently, row by row).

The matrix is both constrained to be symmetric and to be positive semidefinite. That is, if the functions in entries (i, j) and (j, i) are different, then a constraint will be added to make sure that the entries are equal.

Examples

Constraining the matrix

$$\begin{bmatrix} 1 & -y \\ -z & 0 \end{bmatrix}$$

to be symmetric positive semidefinite can be achieved by constraining the vector $(1, -z, -y, 0)$ (or $(1, -y, -z, 0)$) to belong to the `PositiveSemidefiniteConeSquare(2)`.

It both constrains $y = z$ and $(1, -y, 0)$ (or $(1, -z, 0)$) to be in `PositiveSemidefiniteConeTriangle(2)`.

`MathOptInterface.LogDetConeTriangle` – Type.

```
LogDetConeTriangle(side_dimension)
```

The log-determinant cone $\{(t, u, X) \in \mathbb{R}^{2+d(d+1)/2} : t \leq u \log(\det(X/u)), u > 0\}$, where the matrix `X` is represented in the same symmetric packed format as in the `PositiveSemidefiniteConeTriangle`.

The argument `side_dimension` is the side dimension of the matrix `X`, i.e., its number of rows or columns.

`MathOptInterface.LogDetConeSquare` – Type.

```
LogDetConeSquare(side_dimension)
```

The log-determinant cone $\{(t, u, X) \in \mathbb{R}^{2+d^2} : t \leq u \log(\det(X/u)), X \text{ symmetric}, u > 0\}$, where the matrix `X` is represented in the same format as in the `PositiveSemidefiniteConeSquare`.

Similarly to `PositiveSemidefiniteConeSquare`, constraints are added to ensure that `X` is symmetric.

The argument `side_dimension` is the side dimension of the matrix `X`, i.e., its number of rows or columns.

`MathOptInterface.RootDetConeTriangle` – Type.

```
RootDetConeTriangle(side_dimension)
```

The root-determinant cone $\{(t, X) \in \mathbb{R}^{1+d(d+1)/2} : t \leq \det(X)^{1/d}\}$, where the matrix `X` is represented in the same symmetric packed format as in the `PositiveSemidefiniteConeTriangle`.

The argument `side_dimension` is the side dimension of the matrix `X`, i.e., its number of rows or columns.

MathOptInterface.RootDetConeSquare – Type.

```
RootDetConeSquare(side_dimension)
```

The root-determinant cone $\{(t, X) \in \mathbb{R}^{1+d^2} : t \leq \det(X)^{1/d}, X \text{ symmetric}\}$, where the matrix X is represented in the same format as PositiveSemidefiniteConeSquare.

Similarly to PositiveSemidefiniteConeSquare, constraints are added to ensure that X is symmetric.

The argument side_dimension is the side dimension of the matrix X, i.e., its number of rows or columns.

39.2 Models

Attribute interface

MathOptInterface.is_set_by_optimize – Function.

```
is_set_by_optimize(::AnyAttribute)
```

Return a Bool indicating whether the value of the attribute is modified during an optimize! call, that is, the attribute is used to query the result of the optimization.

Important note when defining new attributes

This function returns false by default so it should be implemented for attributes that are modified by optimize!.

MathOptInterface.is_copyable – Function.

```
is_copyable(::AnyAttribute)
```

Return a Bool indicating whether the value of the attribute may be copied during copy_to using set.

Important note when defining new attributes

By default is_copyable(attr) returns !is_set_by_optimize(attr). A specific method should be defined for attributes which are copied indirectly during copy_to. For instance, both is_copyable and is_set_by_optimize return false for the following attributes:

- ListOfOptimizerAttributesSet, ListOfModelAttributesSet, ListOfConstraintAttributesSet and ListOfVariableAttributesSet.
- SolverName and RawSolver: these attributes cannot be set.
- NumberOfVariables and ListOfVariableIndices: these attributes are set indirectly by add_variable and add_variables.
- ObjectiveFunctionType: this attribute is set indirectly when setting the ObjectiveFunction attribute.
- NumberOfConstraints, ListOfConstraintIndices, ListOfConstraintTypesPresent, CanonicalConstraintFunctio ConstraintFunction and ConstraintSet: these attributes are set indirectly by add_constraint and add_constraints.

MathOptInterface.get – Function.

```
get(optimizer::AbstractOptimizer, attr::AbstractOptimizerAttribute)
```

Return an attribute attr of the optimizer optimizer.

```
get(model::ModelLike, attr::AbstractModelAttribute)
```

Return an attribute `attr` of the model `model`.

```
get(model::ModelLike, attr::AbstractVariableAttribute, v::VariableIndex)
```

If the attribute `attr` is set for the variable `v` in the model `model`, return its value, return `nothing` otherwise. If the attribute `attr` is not supported by `model` then an error should be thrown instead of returning `nothing`.

```
get(model::ModelLike, attr::AbstractVariableAttribute, v::Vector{VariableIndex})
```

Return a vector of attributes corresponding to each variable in the collection `v` in the model `model`.

```
get(model::ModelLike, attr::AbstractConstraintAttribute, c::ConstraintIndex)
```

If the attribute `attr` is set for the constraint `c` in the model `model`, return its value, return `nothing` otherwise. If the attribute `attr` is not supported by `model` then an error should be thrown instead of returning `nothing`.

```
get(model::ModelLike, attr::AbstractConstraintAttribute, c::Vector{ConstraintIndex{F,S}})
```

Return a vector of attributes corresponding to each constraint in the collection `c` in the model `model`.

```
get(model::ModelLike, ::Type{VariableIndex}, name::String)
```

If a variable with name `name` exists in the model `model`, return the corresponding index, otherwise return `nothing`. Errors if two variables have the same name.

```
get(model::ModelLike, ::Type{ConstraintIndex{F,S}}, name::String) where {F<:AbstractFunction,S<:
    AbstractSet}
```

If an F-in-S constraint with name `name` exists in the model `model`, return the corresponding index, otherwise return `nothing`. Errors if two constraints have the same name.

```
get(model::ModelLike, ::Type{ConstraintIndex}, name::String)
```

If any constraint with name `name` exists in the model `model`, return the corresponding index, otherwise return `nothing`. This version is available for convenience but may incur a performance penalty because it is not type stable. Errors if two constraints have the same name.

Examples

```
get(model, ObjectiveValue())
get(model, VariablePrimal(), ref)
get(model, VariablePrimal(5), [ref1, ref2])
get(model, OtherAttribute("something specific to cplex"))
get(model, VariableIndex, "var1")
get(model, ConstraintIndex{ScalarAffineFunction{Float64},LessThan{Float64}}, "con1")
get(model, ConstraintIndex, "con1")
```

`MathOptInterface.get!` – Function.

```
get!(output, model::ModelLike, args...)
```

An in-place version of `get`.

The signature matches that of `get` except that the the result is placed in the vector `output`.

`MathOptInterface.set` – Function.

```
set(optimizer::AbstractOptimizer, attr::AbstractOptimizerAttribute, value)
```

Assign `value` to the attribute `attr` of the optimizer `optimizer`.

```
set(model::ModelLike, attr::AbstractModelAttribute, value)
```

Assign `value` to the attribute `attr` of the model `model`.

```
set(model::ModelLike, attr::AbstractVariableAttribute, v::VariableIndex, value)
```

Assign `value` to the attribute `attr` of variable `v` in model `model`.

```
set(model::ModelLike, attr::AbstractVariableAttribute, v::Vector{VariableIndex}, vector_of_values
    )
```

Assign a value respectively to the attribute `attr` of each variable in the collection `v` in model `model`.

```
set(model::ModelLike, attr::AbstractConstraintAttribute, c::ConstraintIndex, value)
```

Assign a value to the attribute `attr` of constraint `c` in model `model`.

```
set(model::ModelLike, attr::AbstractConstraintAttribute, c::Vector{ConstraintIndex{F,S}},
    vector_of_values)
```

Assign a value respectively to the attribute `attr` of each constraint in the collection `c` in model `model`.

An `UnsupportedAttribute` error is thrown if `model` does not support the attribute `attr` (see `supports`) and a `SetAttributeNotAllowed` error is thrown if it supports the attribute `attr` but it cannot be set.

Replace set in a constraint

```
set(model::ModelLike, ::ConstraintSet, c::ConstraintIndex{F,S}, set::S)
```

Change the set of constraint `c` to the new set `set` which should be of the same type as the original set.

Examples

If `c` is a `ConstraintIndex{F,Interval}`

```
set(model, ConstraintSet(), c, Interval(0, 5))
set(model, ConstraintSet(), c, GreaterThan(0.0))  # Error
```

Replace function in a constraint

```
set(model::ModelLike, ::ConstraintFunction, c::ConstraintIndex{F,S}, func::F)
```

Replace the function in constraint `c` with `func`. `F` must match the original function type used to define the constraint.

Note

Setting the constraint function is not allowed if `F` is `VariableIndex`, it throws a `SettingVariableIndexNotAllowed` error. Indeed, it would require changing the index `c` as the index of `VariableIndex` constraints should be the same as the index of the variable.

Examples

If `c` is a `ConstraintIndex{ScalarAffineFunction,S}` and `v1` and `v2` are `VariableIndex` objects,

```
set(model, ConstraintFunction(), c,
    ScalarAffineFunction(ScalarAffineTerm.([1.0, 2.0], [v1, v2]), 5.0))
set(model, ConstraintFunction(), c, v1) # Error
```

MathOptInterface.supports – Function.

```
supports(model::ModelLike, sub::AbstractSubmittable)::Bool
```

Return a Bool indicating whether model supports the submittable sub.

```
supports(model::ModelLike, attr::AbstractOptimizerAttribute)::Bool
```

Return a Bool indicating whether model supports the optimizer attribute attr. That is, it returns false if copy_to(model, src) shows a warning in case attr is in the ListOfOptimizerAttributesSet of src; see copy_to for more details on how unsupported optimizer attributes are handled in copy.

```
supports(model::ModelLike, attr::AbstractModelAttribute)::Bool
```

Return a Bool indicating whether model supports the model attribute attr. That is, it returns false if copy_to(model, src) cannot be performed in case attr is in the ListOfModelAttributesSet of src.

```
supports(model::ModelLike, attr::AbstractVariableAttribute, ::Type{VariableIndex})::Bool
```

Return a Bool indicating whether model supports the variable attribute attr. That is, it returns false if copy_to(model, src) cannot be performed in case attr is in the ListOfVariableAttributesSet of src.

```
supports(model::ModelLike, attr::AbstractConstraintAttribute, ::Type{ConstraintIndex{F,S}})::Bool
     where {F,S}
```

Return a Bool indicating whether model supports the constraint attribute attr applied to an F-in-S constraint. That is, it returns false if copy_to(model, src) cannot be performed in case attr is in the ListOfConstraintAttributesSet of src.

For all five methods, if the attribute is only not supported in specific circumstances, it should still return true.

Note that supports is only defined for attributes for which is_copyable returns true as other attributes do not appear in the list of attributes set obtained by ListOf...AttributesSet.

MathOptInterface.attribute_value_type – Function.

```
attribute_value_type(attr::AnyAttribute)
```

Given an attribute attr, return the type of value expected by get, or returned by set.

Notes

- Only implement this if it make sense to do so. If un-implemented, the default is Any.

Model interface

MathOptInterface.ModelLike – Type.

```
ModelLike
```

Abstract supertype for objects that implement the "Model" interface for defining an optimization problem.

MathOptInterface.is_empty – Function.

```
is_empty(model::ModelLike)
```

Returns false if the model has any model attribute set or has any variables or constraints.

Note that an empty model can have optimizer attributes set.

MathOptInterface.empty! – Function.

```
empty!(model::ModelLike)
```

Empty the model, that is, remove all variables, constraints and model attributes but not optimizer attributes.

MathOptInterface.write_to_file – Function.

```
write_to_file(model::ModelLike, filename::String)
```

Writes the current model data to the given file. Supported file types depend on the model type.

MathOptInterface.read_from_file – Function.

```
read_from_file(model::ModelLike, filename::String)
```

Read the file `filename` into the model `model`. If `model` is non-empty, this may throw an error.

Supported file types depend on the model type.

Note

Once the contents of the file are loaded into the model, users can query the variables via `get(model, ListOfVariableIndices())`. However, some filetypes, such as LP files, do not maintain an explicit ordering of the variables. Therefore, the returned list may be in an arbitrary order. To avoid depending on the order of the indices, users should look up each variable index by name: `get(model, VariableIndex, "name")`.

MathOptInterface.supports_incremental_interface – Function.

```
supports_incremental_interface(model::ModelLike)
```

Return a `Bool` indicating whether `model` supports building incrementally via `add_variable` and `add_constraint`.

The main purpose of this function is to determine whether a model can be loaded into `model` incrementally or whether it should be cached and copied at once instead.

MathOptInterface.copy_to – Function.

```
copy_to(dest::ModelLike, src::ModelLike)::IndexMap
```

Copy the model from `src` into `dest`.

The target `dest` is emptied, and all previous indices to variables and constraints in `dest` are invalidated.

Returns an `IndexMap` object that translates variable and constraint indices from the `src` model to the corresponding indices in the `dest` model.

Notes

- If a constraint that in `src` is not supported by `dest`, then an `UnsupportedConstraint` error is thrown.
- If an `AbstractModelAttribute`, `AbstractVariableAttribute`, or `AbstractConstraintAttribute` is set in `src` but not supported by `dest`, then an `UnsupportedAttribute` error is thrown.

`AbstractOptimizerAttribute`s are not copied to the `dest` model.

IndexMap

Implementations of `copy_to` must return an `IndexMap`. For technical reasons, this type is defined in the Utilities submodule as `MOI.Utilities.IndexMap`. However, since it is an integral part of the MOI API, we provide `MOI.IndexMap` as an alias.

Example

```
# Given empty `ModelLike` objects `src` and `dest`.

x = add_variable(src)

is_valid(src, x)   # true
is_valid(dest, x)  # false (`dest` has no variables)

index_map = copy_to(dest, src)
is_valid(dest, x) # false (unless index_map[x] == x)
is_valid(dest, index_map[x]) # true
```

`MathOptInterface.IndexMap` – Type.

```
IndexMap()
```

The dictionary-like object returned by `copy_to`.

IndexMap

Implementations of `copy_to` must return an `IndexMap`. For technical reasons, the `IndexMap` type is defined in the Utilties submodule as `MOI.Utilities.IndexMap`. However, since it is an integral part of the MOI API, we provide this `MOI.IndexMap` as an alias.

Model attributes

`MathOptInterface.AbstractModelAttribute` – Type.

```
AbstractModelAttribute
```

Abstract supertype for attribute objects that can be used to set or get attributes (properties) of the model.

`MathOptInterface.Name` – Type.

```
Name()
```

A model attribute for the string identifying the model. It has a default value of `""` if not set‘.

`MathOptInterface.ObjectiveFunction` – Type.

```
ObjectiveFunction{F<:AbstractScalarFunction}()
```

A model attribute for the objective function which has a type `F<:AbstractScalarFunction`. `F` should be guaranteed to be equivalent but not necessarily identical to the function type provided by the user. Throws an `InexactError` if the objective function cannot be converted to `F`, e.g. the objective function is quadratic and `F` is `ScalarAffineFunction{Float64}` or it has non-integer coefficient and `F` is `ScalarAffineFunction{Int}`.

`MathOptInterface.ObjectiveFunctionType` – Type.

```
ObjectiveFunctionType()
```

A model attribute for the type F of the objective function set using the `ObjectiveFunction{F}` attribute.

Examples

In the following code, `attr` should be equal to `MOI.VariableIndex`:

```
x = MOI.add_variable(model)
MOI.set(model, MOI.ObjectiveFunction{MOI.VariableIndex}(),
         x)
attr = MOI.get(model, MOI.ObjectiveFunctionType())
```

`MathOptInterface.ObjectiveSense` – Type.

```
ObjectiveSense()
```

A model attribute for the objective sense of the objective function, which must be an `OptimizationSense`: `MIN_SENSE`, `MAX_SENSE`, or `FEASIBILITY_SENSE`. The default is `FEASIBILITY_SENSE`.

`MathOptInterface.NumberOfVariables` – Type.

```
NumberOfVariables()
```

A model attribute for the number of variables in the model.

`MathOptInterface.ListOfVariableIndices` – Type.

```
ListOfVariableIndices()
```

A model attribute for the `Vector{VariableIndex}` of all variable indices present in the model (i.e., of length equal to the value of `NumberOfVariables()`) in the order in which they were added.

`MathOptInterface.ListOfConstraintTypesPresent` – Type.

```
ListOfConstraintTypesPresent()
```

A model attribute for the list of tuples of the form `(F,S)`, where `F` is a function type and `S` is a set type indicating that the attribute `NumberOfConstraints{F,S}()` has value greater than zero.

`MathOptInterface.NumberOfConstraints` – Type.

```
NumberOfConstraints{F,S}()
```

A model attribute for the number of constraints of the type F-in-S present in the model.

`MathOptInterface.ListOfConstraintIndices` – Type.

```
ListOfConstraintIndices{F,S}()
```

A model attribute for the `Vector{ConstraintIndex{F,S}}` of all constraint indices of type F-in-S in the model (i.e., of length equal to the value of `NumberOfConstraints{F,S}()`) in the order in which they were added.

`MathOptInterface.ListOfOptimizerAttributesSet` – Type.

```
ListOfOptimizerAttributesSet()
```

An optimizer attribute for the `Vector{AbstractOptimizerAttribute}` of all optimizer attributes that were set.

MathOptInterface.ListOfModelAttributesSet – Type.

```
ListOfModelAttributesSet()
```

A model attribute for the `Vector{AbstractModelAttribute}` of all model attributes `attr` such that 1) `is_copyable(attr)` returns `true` and 2) the attribute was set to the model.

MathOptInterface.ListOfVariableAttributesSet – Type.

```
ListOfVariableAttributesSet()
```

A model attribute for the `Vector{AbstractVariableAttribute}` of all variable attributes `attr` such that 1) `is_copyable(attr)` returns `true` and 2) the attribute was set to variables.

MathOptInterface.ListOfConstraintAttributesSet – Type.

```
ListOfConstraintAttributesSet{F, S}()
```

A model attribute for the `Vector{AbstractConstraintAttribute}` of all constraint attributes `attr` such that 1) `is_copyable(attr)` returns `true` and

2. the attribute was set to F-in-S constraints.

Note

The attributes `ConstraintFunction` and `ConstraintSet` should not be included in the list even if then have been set with `set`.

Optimizer interface

MathOptInterface.AbstractOptimizer – Type.

```
AbstractOptimizer <: ModelLike
```

Abstract supertype for objects representing an instance of an optimization problem tied to a particular solver. This is typically a solver's in-memory representation. In addition to `ModelLike`, `AbstractOptimizer` objects let you solve the model and query the solution.

MathOptInterface.OptimizerWithAttributes – Type.

```
struct OptimizerWithAttributes
    optimizer_constructor
    params::Vector{Pair{AbstractOptimizerAttribute,<:Any}}
end
```

Object grouping an optimizer constructor and a list of optimizer attributes. Instances are created with `instantiate`.

MathOptInterface.optimize! – Function.

```
optimize!(optimizer::AbstractOptimizer)
```

Optimize the problem contained in `optimizer`.

Before calling `optimize!`, the problem should first be constructed using the incremental interface (see `supports_incremental_interface`) or `copy_to`.

MathOptInterface.instantiate – Function.

```
instantiate(
    optimizer_constructor,
    with_bridge_type::Union{Nothing, Type} = nothing,
)
```

Creates an instance of optimizer by either:

- calling `optimizer_constructor.optimizer_constructor()` and setting the parameters in `optimizer_constructor.p` if `optimizer_constructor` is a `OptimizerWithAttributes`
- calling `optimizer_constructor()` if `optimizer_constructor` is callable.

If `with_bridge_type` is not `nothing`, it enables all the bridges defined in the MathOptInterface.Bridges submodule with coefficient type `with_bridge_type`.

If the optimizer created by `optimizer_constructor` does not support loading the problem incrementally (see `supports_incremental_interface`), then a `Utilities.CachingOptimizer` is added to store a cache of the bridged model.

Optimizer attributes

MathOptInterface.AbstractOptimizerAttribute – Type.

```
AbstractOptimizerAttribute
```

Abstract supertype for attribute objects that can be used to set or get attributes (properties) of the optimizer.

Note

The difference between `AbstractOptimizerAttribute` and `AbstractModelAttribute` lies in the behavior of `is_empty`, `empty!` and `copy_to`. Typically optimizer attributes only affect how the model is solved.

MathOptInterface.SolverName – Type.

```
SolverName()
```

An optimizer attribute for the string identifying the solver/optimizer.

MathOptInterface.SolverVersion – Type.

```
SolverVersion()
```

An optimizer attribute for the string identifying the version of the solver.

Note

For solvers supporting semantic versioning, the `SolverVersion` should be a string of the form "vMAJOR.MINOR.PATCH", so that it can be converted to a Julia `VersionNumber` (e.g., 'VersionNumber("v1.2.3")).

We do not require Semantic Versioning because some solvers use alternate versioning systems. For example, CPLEX uses Calendar Versioning, so `SolverVersion` will return a string like `"202001"`.

MathOptInterface.Silent – Type.

```
Silent()
```

An optimizer attribute for silencing the output of an optimizer. When `set` to `true`, it takes precedence over any other attribute controlling verbosity and requires the solver to produce no output. The default value is `false` which has no effect. In this case the verbosity is controlled by other attributes.

Note

Every optimizer should have verbosity on by default. For instance, if a solver has a solver-specific log level attribute, the MOI implementation should set it to `1` by default. If the user sets `Silent` to `true`, then the log level should be set to `0`, even if the user specifically sets a value of log level. If the value of `Silent` is `false` then the log level set to the solver is the value given by the user for this solver-specific parameter or 1 if none is given.

`MathOptInterface.TimeLimitSec` – Type.

```
TimeLimitSec()
```

An optimizer attribute for setting a time limit for an optimization. When `set` to `nothing`, it deactivates the solver time limit. The default value is `nothing`. The time limit is in seconds.

`MathOptInterface.RawOptimizerAttribute` – Type.

```
RawOptimizerAttribute(name::String)
```

An optimizer attribute for the solver-specific parameter identified by `name`.

`MathOptInterface.NumberOfThreads` – Type.

```
NumberOfThreads()
```

An optimizer attribute for setting the number of threads used for an optimization. When set to `nothing` uses solver default. Values are positive integers. The default value is `nothing`.

`MathOptInterface.RawSolver` – Type.

```
RawSolver()
```

A model attribute for the object that may be used to access a solver-specific API for this optimizer.

List of attributes useful for optimizers

`MathOptInterface.TerminationStatus` – Type.

```
TerminationStatus()
```

A model attribute for the `TerminationStatusCode` explaining why the optimizer stopped.

`MathOptInterface.TerminationStatusCode` – Type.

```
TerminationStatusCode
```

An Enum of possible values for the `TerminationStatus` attribute. This attribute is meant to explain the reason why the optimizer stopped executing in the most recent call to `optimize!`.

If no call has been made to `optimize!`, then the `TerminationStatus` is:

- `OPTIMIZE_NOT_CALLED`: The algorithm has not started.

OK

These are generally OK statuses, i.e., the algorithm ran to completion normally.

- `OPTIMAL`: The algorithm found a globally optimal solution.
- `INFEASIBLE`: The algorithm concluded that no feasible solution exists.
- `DUAL_INFEASIBLE`: The algorithm concluded that no dual bound exists for the problem. If, additionally, a feasible (primal) solution is known to exist, this status typically implies that the problem is unbounded, with some technical exceptions.
- `LOCALLY_SOLVED`: The algorithm converged to a stationary point, local optimal solution, could not find directions for improvement, or otherwise completed its search without global guarantees.
- `LOCALLY_INFEASIBLE`: The algorithm converged to an infeasible point or otherwise completed its search without finding a feasible solution, without guarantees that no feasible solution exists.
- `INFEASIBLE_OR_UNBOUNDED`: The algorithm stopped because it decided that the problem is infeasible or unbounded; this occasionally happens during MIP presolve.

Solved to relaxed tolerances

- `ALMOST_OPTIMAL`: The algorithm found a globally optimal solution to relaxed tolerances.
- `ALMOST_INFEASIBLE`: The algorithm concluded that no feasible solution exists within relaxed tolerances.
- `ALMOST_DUAL_INFEASIBLE`: The algorithm concluded that no dual bound exists for the problem within relaxed tolerances.
- `ALMOST_LOCALLY_SOLVED`: The algorithm converged to a stationary point, local optimal solution, or could not find directions for improvement within relaxed tolerances.

Limits

The optimizer stopped because of some user-defined limit.

- `ITERATION_LIMIT`: An iterative algorithm stopped after conducting the maximum number of iterations.
- `TIME_LIMIT`: The algorithm stopped after a user-specified computation time.
- `NODE_LIMIT`: A branch-and-bound algorithm stopped because it explored a maximum number of nodes in the branch-and-bound tree.
- `SOLUTION_LIMIT`: The algorithm stopped because it found the required number of solutions. This is often used in MIPs to get the solver to return the first feasible solution it encounters.
- `MEMORY_LIMIT`: The algorithm stopped because it ran out of memory.
- `OBJECTIVE_LIMIT`: The algorithm stopped because it found a solution better than a minimum limit set by the user.
- `NORM_LIMIT`: The algorithm stopped because the norm of an iterate became too large.
- `OTHER_LIMIT`: The algorithm stopped due to a limit not covered by one of the above.

Problematic

This group of statuses means that something unexpected or problematic happened.

- `SLOW_PROGRESS`: The algorithm stopped because it was unable to continue making progress towards the solution.

- `NUMERICAL_ERROR`: The algorithm stopped because it encountered unrecoverable numerical error.
- `INVALID_MODEL`: The algorithm stopped because the model is invalid.
- `INVALID_OPTION`: The algorithm stopped because it was provided an invalid option.
- `INTERRUPTED`: The algorithm stopped because of an interrupt signal.
- `OTHER_ERROR`: The algorithm stopped because of an error not covered by one of the statuses defined above.

`MathOptInterface.PrimalStatus` – Type.

```
PrimalStatus(result_index::Int = 1)
```

A model attribute for the `ResultStatusCode` of the primal result `result_index`. If `result_index` is omitted, it defaults to 1.

See `ResultCount` for information on how the results are ordered.

If `result_index` is larger than the value of `ResultCount` then `NO_SOLUTION` is returned.

`MathOptInterface.DualStatus` – Type.

```
DualStatus(result_index::Int = 1)
```

A model attribute for the `ResultStatusCode` of the dual result `result_index`. If `result_index` is omitted, it defaults to 1.

See `ResultCount` for information on how the results are ordered.

If `result_index` is larger than the value of `ResultCount` then `NO_SOLUTION` is returned.

`MathOptInterface.ResultStatusCode` – Type.

```
ResultStatusCode
```

An Enum of possible values for the `PrimalStatus` and `DualStatus` attributes. The values indicate how to interpret the result vector.

- `NO_SOLUTION`: the result vector is empty.
- `FEASIBLE_POINT`: the result vector is a feasible point.
- `NEARLY_FEASIBLE_POINT`: the result vector is feasible if some constraint tolerances are relaxed.
- `INFEASIBLE_POINT`: the result vector is an infeasible point.
- `INFEASIBILITY_CERTIFICATE`: the result vector is an infeasibility certificate. If the `PrimalStatus` is `INFEASIBILITY_CERTIFICATE`, then the primal result vector is a certificate of dual infeasibility. If the `DualStatus` is `INFEASIBILITY_CERTIFICATE`, then the dual result vector is a proof of primal infeasibility.
- `NEARLY_INFEASIBILITY_CERTIFICATE`: the result satisfies a relaxed criterion for a certificate of infeasibility.
- `REDUCTION_CERTIFICATE`: the result vector is an ill-posed certificate; see this article for details. If the `PrimalStatus` is `REDUCTION_CERTIFICATE`, then the primal result vector is a proof that the dual problem is ill-posed. If the `DualStatus` is `REDUCTION_CERTIFICATE`, then the dual result vector is a proof that the primal is ill-posed.
- `NEARLY_REDUCTION_CERTIFICATE`: the result satisfies a relaxed criterion for an ill-posed certificate.
- `UNKNOWN_RESULT_STATUS`: the result vector contains a solution with an unknown interpretation.

- OTHER_RESULT_STATUS: the result vector contains a solution with an interpretation not covered by one of the statuses defined above.

MathOptInterface.RawStatusString – Type.

```
RawStatusString()
```

A model attribute for a solver specific string explaining why the optimizer stopped.

MathOptInterface.ResultCount – Type.

```
ResultCount()
```

A model attribute for the number of results available.

Order of solutions

A number of attributes contain an index, result_index, which is used to refer to one of the available results. Thus, result_index must be an integer between 1 and the number of available results.

As a general rule, the first result (result_index=1) is the most important result (e.g., an optimal solution or an infeasibility certificate). Other results will typically be alternate solutions that the solver found during the search for the first result.

If a (local) optimal solution is available, i.e., TerminationStatus is OPTIMAL or LOCALLY_SOLVED, the first result must correspond to the (locally) optimal solution. Other results may be alternative optimal solutions, or they may be other suboptimal solutions; use ObjectiveValue to distingiush between them.

If a primal or dual infeasibility certificate is available, i.e., TerminationStatus is INFEASIBLE or DUAL_INFEASIBLE and the corresponding PrimalStatus or DualStatus is INFEASIBILITY_CERTIFICATE, then the first result must be a certificate. Other results may be alternate certificates, or infeasible points.

MathOptInterface.ObjectiveValue – Type.

```
ObjectiveValue(result_index::Int = 1)
```

A model attribute for the objective value of the primal solution result_index.

If the solver does not have a primal value for the objective because the result_index is beyond the available solutions (whose number is indicated by the ResultCount attribute), getting this attribute must throw a ResultIndexBoundsError. Otherwise, if the result is unavailable for another reason (for instance, only a dual solution is available), the result is undefined. Users should first check PrimalStatus before accessing the ObjectiveValue attribute.

See ResultCount for information on how the results are ordered.

MathOptInterface.DualObjectiveValue – Type.

```
DualObjectiveValue(result_index::Int = 1)
```

A model attribute for the value of the objective function of the dual problem for the result_indexth dual result.

If the solver does not have a dual value for the objective because the result_index is beyond the available solutions (whose number is indicated by the ResultCount attribute), getting this attribute must throw a ResultIndexBoundsError. Otherwise, if the result is unavailable for another reason (for instance, only a primal solution is available), the result is undefined. Users should first check DualStatus before accessing the DualObjectiveValue attribute.

See ResultCount for information on how the results are ordered.

MathOptInterface.ObjectiveBound – Type.

```
ObjectiveBound()
```

A model attribute for the best known bound on the optimal objective value.

MathOptInterface.RelativeGap – Type.

```
RelativeGap()
```

A model attribute for the final relative optimality gap.

> **Warning**
> The definition of this gap is solver-dependent. However, most solvers implementing this attribute define the relative gap as some variation of $\frac{|b-f|}{|f|}$, where b is the best bound and f is the best feasible objective value.

MathOptInterface.SolveTimeSec – Type.

```
SolveTimeSec()
```

A model attribute for the total elapsed solution time (in seconds) as reported by the optimizer.

MathOptInterface.SimplexIterations – Type.

```
SimplexIterations()
```

A model attribute for the cumulative number of simplex iterations during the optimization process. In particular, for a mixed-integer program (MIP), the total simplex iterations for all nodes.

MathOptInterface.BarrierIterations – Type.

```
BarrierIterations()
```

A model attribute for the cumulative number of barrier iterations while solving a problem.

MathOptInterface.NodeCount – Type.

```
NodeCount()
```

A model attribute for the total number of branch-and-bound nodes explored while solving a mixed-integer program (MIP).

Conflict Status

MathOptInterface.compute_conflict! – Function.

```
compute_conflict!(optimizer::AbstractOptimizer)
```

Computes a minimal subset of constraints such that the model with the other constraint removed is still infeasible.

Some solvers call a set of conflicting constraints an Irreducible Inconsistent Subsystem (IIS).

See also ConflictStatus and ConstraintConflictStatus.

Note

If the model is modified after a call to `compute_conflict!`, the implementor is not obliged to purge the conflict. Any calls to the above attributes may return values for the original conflict without a warning. Similarly, when modifying the model, the conflict can be discarded.

MathOptInterface.ConflictStatus – Type.

```
ConflictStatus()
```

A model attribute for the ConflictStatusCode explaining why the conflict refiner stopped when computing the conflict.

MathOptInterface.ConflictStatusCode – Type.

```
ConflictStatusCode
```

An Enum of possible values for the ConflictStatus attribute. This attribute is meant to explain the reason why the conflict finder stopped executing in the most recent call to compute_conflict!.

Possible values are:

- COMPUTE_CONFLICT_NOT_CALLED: the function compute_conflict! has not yet been called
- NO_CONFLICT_EXISTS: there is no conflict because the problem is feasible
- NO_CONFLICT_FOUND: the solver could not find a conflict
- CONFLICT_FOUND: at least one conflict could be found

MathOptInterface.ConstraintConflictStatus – Type.

```
ConstraintConflictStatus()
```

A constraint attribute indicating whether the constraint participates in the conflict. Its type is ConflictParticipationStatusC

MathOptInterface.ConflictParticipationStatusCode – Type.

```
ConflictParticipationStatusCode
```

An Enum of possible values for the ConstraintConflictStatus attribute. This attribute is meant to indicate whether a given constraint participates or not in the last computed conflict.

Possible values are:

- NOT_IN_CONFLICT: the constraint does not participate in the conflict
- IN_CONFLICT: the constraint participates in the conflict
- MAYBE_IN_CONFLICT: the constraint may participate in the conflict, the solver was not able to prove that the constraint can be excluded from the conflict

39.3 Variables

Functions

MathOptInterface.add_variable – Function.

```
add_variable(model::ModelLike)::VariableIndex
```

Add a scalar variable to the model, returning a variable index.

A AddVariableNotAllowed error is thrown if adding variables cannot be done in the current state of the model model.

MathOptInterface.add_variables – Function.

```
add_variables(model::ModelLike, n::Int)::Vector{VariableIndex}
```

Add n scalar variables to the model, returning a vector of variable indices.

A AddVariableNotAllowed error is thrown if adding variables cannot be done in the current state of the model model.

MathOptInterface.add_constrained_variable – Function.

```
add_constrained_variable(
    model::ModelLike,
    set::AbstractScalarSet
)::Tuple{MOI.VariableIndex,
         MOI.ConstraintIndex{MOI.VariableIndex, typeof(set)}}
```

Add to model a scalar variable constrained to belong to set, returning the index of the variable created and the index of the constraint constraining the variable to belong to set.

By default, this function falls back to creating a free variable with add_variable and then constraining it to belong to set with add_constraint.

MathOptInterface.add_constrained_variables – Function.

```
add_constrained_variables(
    model::ModelLike,
    sets::AbstractVector{<:AbstractScalarSet}
)::Tuple{
    Vector{MOI.VariableIndex},
    Vector{MOI.ConstraintIndex{MOI.VariableIndex,eltype(sets)}},
}
```

Add to model scalar variables constrained to belong to sets, returning the indices of the variables created and the indices of the constraints constraining the variables to belong to each set in sets. That is, if it returns variables and constraints, constraints[i] is the index of the constraint constraining variable[i] to belong to sets[i].

By default, this function falls back to calling add_constrained_variable on each set.

```
add_constrained_variables(
    model::ModelLike,
    set::AbstractVectorSet,
)::Tuple{
    Vector{MOI.VariableIndex},
    MOI.ConstraintIndex{MOI.VectorOfVariables,typeof(set)},
}
```

Add to model a vector of variables constrained to belong to set, returning the indices of the variables created and the index of the constraint constraining the vector of variables to belong to set.

By default, this function falls back to creating free variables with add_variables and then constraining it to belong to set with add_constraint.

MathOptInterface.supports_add_constrained_variable – Function.

```
supports_add_constrained_variable(
    model::ModelLike,
    S::Type{<:AbstractScalarSet}
)::Bool
```

Return a `Bool` indicating whether `model` supports constraining a variable to belong to a set of type `S` either on creation of the variable with `add_constrained_variable` or after the variable is created with `add_constraint`.

By default, this function falls back to `supports_add_constrained_variables(model, Reals) && supports_constraint(mo MOI.VariableIndex, S)` which is the correct definition for most models.

Example

Suppose that a solver supports only two kind of variables: binary variables and continuous variables with a lower bound. If the solver decides not to support `VariableIndex`-in-`Binary` and `VariableIndex`-in-`GreaterThan` constraints, it only has to implement `add_constrained_variable` for these two sets which prevents the user to add both a binary constraint and a lower bound on the same variable. Moreover, if the user adds a `VariableIndex`-in-`GreaterThan` constraint, implementing this interface (i.e., `supports_add_constrained_varia` enables the constraint to be transparently bridged into a supported constraint.

`MathOptInterface.supports_add_constrained_variables` – Function.

```
supports_add_constrained_variables(
    model::ModelLike,
    S::Type{<:AbstractVectorSet}
)::Bool
```

Return a `Bool` indicating whether `model` supports constraining a vector of variables to belong to a set of type `S` either on creation of the vector of variables with `add_constrained_variables` or after the variable is created with `add_constraint`.

By default, if `S` is `Reals` then this function returns `true` and otherwise, it falls back to `supports_add_constrained_variables( Reals) && supports_constraint(model, MOI.VectorOfVariables, S)` which is the correct definition for most models.

Example

In the standard conic form (see Duality), the variables are grouped into several cones and the constraints are affine equality constraints. If `Reals` is not one of the cones supported by the solvers then it needs to implement `supports_add_constrained_variables(::Optimizer, ::Type{Reals}) = false` as free variables are not supported. The solvers should then implement `supports_add_constrained_variables(::Optimizer, ::Type{<:SupportedCones}) = true` where `SupportedCones` is the union of all cone types that are supported; it does not have to implement the method `supports_constraint(::Type{VectorOfVariables}, Type{<:SupportedCones})` as it should return `false` and it's the default. This prevents the user to constrain the same variable in two different cones. When a `VectorOfVariables`-in-`S` is added, the variables of the vector have already been created so they already belong to given cones. If bridges are enabled, the constraint will therefore be bridged by adding slack variables in `S` and equality constraints ensuring that the slack variables are equal to the corresponding variables of the given constraint function.

Note that there may also be sets for which `!supports_add_constrained_variables(model, S)` and `supports_constraint(model, MOI.VectorOfVariables, S)`. For instance, suppose a solver supports positive semidefinite variable constraints and two types of variables: binary variables and nonnegative variables. Then the solver should support adding `VectorOfVariables`-in-`PositiveSemidefiniteConeTriangle` constraints, but it should not support creating variables constrained to belong to the `PositiveSemidefiniteConeTriangle` because the variables in `PositiveSemidefiniteConeTriangle` should first be created as either binary or non-negative.

`MathOptInterface.is_valid` – Method.

```
is_valid(model::ModelLike, index::Index)::Bool
```

Return a `Bool` indicating whether this index refers to a valid object in the model `model`.

MathOptInterface.delete – Method.

```
delete(model::ModelLike, index::Index)
```

Delete the referenced object from the model. Throw DeleteNotAllowed if if index cannot be deleted.

The following modifications also take effect if Index is VariableIndex:

- If index used in the objective function, it is removed from the function, i.e., it is substituted for zero.
- For each func-in-set constraint of the model:
  - If `func isa VariableIndex` and `func == index` then the constraint is deleted.
  - If `func isa VectorOfVariables` and `index in func.variables` then
    - if `length(func.variables) == 1` is one, the constraint is deleted;
    - if `length(func.variables) > 1` and `supports_dimension_update(set)` then then the variable is removed from `func` and `set` is replaced by `update_dimension(set, MOI.dimension(set) - 1)`.
    - Otherwise, a DeleteNotAllowed error is thrown.
  - Otherwise, the variable is removed from `func`, i.e., it is substituted for zero.

MathOptInterface.delete – Method.

```
delete(model::ModelLike, indices::Vector{R<:Index}) where {R}
```

Delete the referenced objects in the vector `indices` from the model. It may be assumed that `R` is a concrete type. The default fallback sequentially deletes the individual items in `indices`, although specialized implementations may be more efficient.

Attributes

MathOptInterface.AbstractVariableAttribute – Type.

```
AbstractVariableAttribute
```

Abstract supertype for attribute objects that can be used to set or get attributes (properties) of variables in the model.

MathOptInterface.VariableName – Type.

```
VariableName()
```

A variable attribute for a string identifying the variable. It is valid for two variables to have the same name; however, variables with duplicate names cannot be looked up using get. It has a default value of `""` if not set'.

MathOptInterface.VariablePrimalStart – Type.

```
VariablePrimalStart()
```

A variable attribute for the initial assignment to some primal variable's value that the optimizer may use to warm-start the solve. May be a number or `nothing` (unset).

MathOptInterface.VariablePrimal – Type.

```
VariablePrimal(result_index::Int = 1)
```

A variable attribute for the assignment to some primal variable's value in result `result_index`. If `result_index` is omitted, it is 1 by default.

If the solver does not have a primal value for the variable because the `result_index` is beyond the available solutions (whose number is indicated by the `ResultCount` attribute), getting this attribute must throw a `ResultIndexBoundsError`. Otherwise, if the result is unavailable for another reason (for instance, only a dual solution is available), the result is undefined. Users should first check `PrimalStatus` before accessing the `VariablePrimal` attribute.

See `ResultCount` for information on how the results are ordered.

`MathOptInterface.VariableBasisStatus` – Type.

```
VariableBasisStatus(result_index::Int = 1)
```

A variable attribute for the `BasisStatusCode` of a variable in result `result_index`, with respect to an available optimal solution basis.

If the solver does not have a basis statue for the variable because the `result_index` is beyond the available solutions (whose number is indicated by the `ResultCount` attribute), getting this attribute must throw a `ResultIndexBoundsError`. Otherwise, if the result is unavailable for another reason (for instance, only a dual solution is available), the result is undefined. Users should first check `PrimalStatus` before accessing the `VariableBasisStatus` attribute.

See `ResultCount` for information on how the results are ordered.

39.4 Constraints

Types

`MathOptInterface.ConstraintIndex` – Type.

```
ConstraintIndex{F, S}
```

A type-safe wrapper for `Int64` for use in referencing F-in-S constraints in a model. The parameter `F` is the type of the function in the constraint, and the parameter `S` is the type of set in the constraint. To allow for deletion, indices need not be consecutive. Indices within a constraint type (i.e. F-in-S) must be unique, but non-unique indices across different constraint types are allowed. If `F` is `VariableIndex` then the index is equal to the index of the variable. That is for an `index::ConstraintIndex{VariableIndex}`, we always have

```
index.value == MOI.get(model, MOI.ConstraintFunction(), index).value
```

Functions

`MathOptInterface.is_valid` – Method.

```
is_valid(model::ModelLike, index::Index)::Bool
```

Return a `Bool` indicating whether this index refers to a valid object in the model `model`.

`MathOptInterface.add_constraint` – Function.

```
add_constraint(model::ModelLike, func::F, set::S)::ConstraintIndex{F,S} where {F,S}
```

Add the constraint $f(x) \in \mathcal{S}$ where f is defined by `func`, and $\mathcal{S}$ is defined by `set`.

```
add_constraint(model::ModelLike, v::VariableIndex, set::S)::ConstraintIndex{VariableIndex,S}
    where {S}
add_constraint(model::ModelLike, vec::Vector{VariableIndex}, set::S)::ConstraintIndex{
    VectorOfVariables,S} where {S}
```

Add the constraint $v \in \mathcal{S}$ where v is the variable (or vector of variables) referenced by v and $\mathcal{S}$ is defined by set.

- An UnsupportedConstraint error is thrown if model does not support F-in-S constraints,
- a AddConstraintNotAllowed error is thrown if it supports F-in-S constraints but it cannot add the constraint(s) in its current state and
- a ScalarFunctionConstantNotZero error may be thrown if func is an AbstractScalarFunction with nonzero constant and set is EqualTo, GreaterThan, LessThan or Interval.
- a LowerBoundAlreadySet error is thrown if F is a VariableIndex and a constraint was already added to this variable that sets a lower bound.
- a UpperBoundAlreadySet error is thrown if F is a VariableIndex and a constraint was already added to this variable that sets an upper bound.

MathOptInterface.add_constraints – Function.

```
add_constraints(model::ModelLike, funcs::Vector{F},
↪  sets::Vector{S})::Vector{ConstraintIndex{F,S}} where {F,S}
```

Add the set of constraints specified by each function-set pair in funcs and sets. F and S should be concrete types. This call is equivalent to add_constraint.(model, funcs, sets) but may be more efficient.

MathOptInterface.transform – Function.

Transform Constraint Set

```
transform(model::ModelLike, c::ConstraintIndex{F,S1}, newset::S2)::ConstraintIndex{F,S2}
```

Replace the set in constraint c with newset. The constraint index c will no longer be valid, and the function returns a new constraint index with the correct type.

Solvers may only support a subset of constraint transforms that they perform efficiently (for example, changing from a LessThan to GreaterThan set). In addition, set modification (where S1 = S2) should be performed via the modify function.

Typically, the user should delete the constraint and add a new one.

Examples

If c is a ConstraintIndex{ScalarAffineFunction{Float64},LessThan{Float64}},

```
c2 = transform(model, c, GreaterThan(0.0))
transform(model, c, LessThan(0.0)) # errors
```

MathOptInterface.supports_constraint – Function.

```
MOI.supports_constraint(
    BT::Type{<:AbstractBridge},
    F::Type{<:MOI.AbstractFunction},
    S::Type{<:MOI.AbstractSet},
)::Bool
```

Return a `Bool` indicating whether the bridges of type `BT` support bridging F-in-S constraints.

```
supports_constraint(
    model::ModelLike,
    ::Type{F},
    ::Type{S},
)::Bool where {F<:AbstractFunction,S<:AbstractSet}
```

Return a `Bool` indicating whether `model` supports F-in-S constraints, that is, `copy_to(model, src)` does not throw `UnsupportedConstraint` when `src` contains F-in-S constraints. If F-in-S constraints are only not supported in specific circumstances, e.g. F-in-S constraints cannot be combined with another type of constraint, it should still return `true`.

Attributes

`MathOptInterface.AbstractConstraintAttribute` – Type.

```
AbstractConstraintAttribute
```

Abstract supertype for attribute objects that can be used to set or get attributes (properties) of constraints in the model.

`MathOptInterface.ConstraintName` – Type.

```
ConstraintName()
```

A constraint attribute for a string identifying the constraint.

It is valid for constraints variables to have the same name; however, constraints with duplicate names cannot be looked up using `get`, regardless of whether they have the same F-in-S type.

`ConstraintName` has a default value of `""` if not set.

Notes

You should not implement `ConstraintName` for `VariableIndex` constraints.

`MathOptInterface.ConstraintPrimalStart` – Type.

```
ConstraintPrimalStart()
```

A constraint attribute for the initial assignment to some constraint's `ConstraintPrimal` that the optimizer may use to warm-start the solve.

May be `nothing` (unset), a number for `AbstractScalarFunction`, or a vector for `AbstractVectorFunction`.

`MathOptInterface.ConstraintDualStart` – Type.

```
ConstraintDualStart()
```

A constraint attribute for the initial assignment to some constraint's `ConstraintDual` that the optimizer may use to warm-start the solve.

May be `nothing` (unset), a number for `AbstractScalarFunction`, or a vector for `AbstractVectorFunction`.

`MathOptInterface.ConstraintPrimal` – Type.

```
ConstraintPrimal(result_index::Int = 1)
```

A constraint attribute for the assignment to some constraint's primal value(s) in result `result_index`.

If the constraint is `f(x) in S`, then in most cases the `ConstraintPrimal` is the value of `f`, evaluated at the correspondng `VariablePrimal` solution.

However, some conic solvers reformulate `b - Ax in S` to `s = b - Ax, s in S`. These solvers may return the value of `s` for `ConstraintPrimal`, rather than `b - Ax`. (Although these are constrained by an equality constraint, due to numerical tolerances they may not be identical.)

If the solver does not have a primal value for the constraint because the `result_index` is beyond the available solutions (whose number is indicated by the `ResultCount` attribute), getting this attribute must throw a `ResultIndexBoundsError`. Otherwise, if the result is unavailable for another reason (for instance, only a dual solution is available), the result is undefined. Users should first check `PrimalStatus` before accessing the `ConstraintPrimal` attribute.

If `result_index` is omitted, it is 1 by default. See `ResultCount` for information on how the results are ordered.

`MathOptInterface.ConstraintDual` – Type.

```
ConstraintDual(result_index::Int = 1)
```

A constraint attribute for the assignment to some constraint's dual value(s) in result `result_index`. If `result_index` is omitted, it is 1 by default.

If the solver does not have a dual value for the variable because the `result_index` is beyond the available solutions (whose number is indicated by the `ResultCount` attribute), getting this attribute must throw a `ResultIndexBoundsError`. Otherwise, if the result is unavailable for another reason (for instance, only a primal solution is available), the result is undefined. Users should first check `DualStatus` before accessing the `ConstraintDual` attribute.

See `ResultCount` for information on how the results are ordered.

`MathOptInterface.ConstraintBasisStatus` – Type.

```
ConstraintBasisStatus(result_index::Int = 1)
```

A constraint attribute for the `BasisStatusCode` of some constraint in result `result_index`, with respect to an available optimal solution basis. If `result_index` is omitted, it is 1 by default.

If the solver does not have a basis statue for the constraint because the `result_index` is beyond the available solutions (whose number is indicated by the `ResultCount` attribute), getting this attribute must throw a `ResultIndexBoundsError`. Otherwise, if the result is unavailable for another reason (for instance, only a dual solution is available), the result is undefined. Users should first check `PrimalStatus` before accessing the `ConstraintBasisStatus` attribute.

See `ResultCount` for information on how the results are ordered.

Notes

For the basis status of a variable, query `VariableBasisStatus`.

`ConstraintBasisStatus` does not apply to `VariableIndex` constraints. You can infer the basis status of a `VariableIndex` constraint by looking at the result of `VariableBasisStatus`.

`MathOptInterface.BasisStatusCode` – Type.

```
BasisStatusCode
```

An Enum of possible values for the ConstraintBasisStatus and VariableBasisStatus attributes, explaining the status of a given element with respect to an optimal solution basis.

Possible values are:

- BASIC: element is in the basis
- NONBASIC: element is not in the basis
- NONBASIC_AT_LOWER: element is not in the basis and is at its lower bound
- NONBASIC_AT_UPPER: element is not in the basis and is at its upper bound
- SUPER_BASIC: element is not in the basis but is also not at one of its bounds

Notes

- NONBASIC_AT_LOWER and NONBASIC_AT_UPPER should be used only for constraints with the Interval set. In this case, they are necessary to distinguish which side of the constraint is active. One-sided constraints (e.g., LessThan and GreaterThan) should use NONBASIC instead of the NONBASIC_AT_* values. This restriction does not apply to VariableBasisStatus, which should return NONBASIC_AT_* regardless of whether the alternative bound exists.
- In linear programs, SUPER_BASIC occurs when a variable with no bounds is not in the basis.

MathOptInterface.ConstraintFunction – Type.

```
ConstraintFunction()
```

A constraint attribute for the AbstractFunction object used to define the constraint. It is guaranteed to be equivalent but not necessarily identical to the function provided by the user.

MathOptInterface.CanonicalConstraintFunction – Type.

```
CanonicalConstraintFunction()
```

A constraint attribute for a canonical representation of the AbstractFunction object used to define the constraint. Getting this attribute is guaranteed to return a function that is equivalent but not necessarily identical to the function provided by the user.

By default, MOI.get(model, MOI.CanonicalConstraintFunction(), ci) fallbacks to MOI.Utilities.canonical(MOI.get MOI.ConstraintFunction(), ci)). However, if model knows that the constraint function is canonical then it can implement a specialized method that directly return the function without calling Utilities.canonical. Therefore, the value returned **cannot** be assumed to be a copy of the function stored in model. Moreover, Utilities.Model checks with Utilities.is_canonical whether the function stored internally is already canonical and if it's the case, then it returns the function stored internally instead of a copy.

MathOptInterface.ConstraintSet – Type.

```
ConstraintSet()
```

A constraint attribute for the AbstractSet object used to define the constraint.

39.5 Modifications

MathOptInterface.modify – Function.

Constraint Function

```
modify(model::ModelLike, ci::ConstraintIndex, change::AbstractFunctionModification)
```

Apply the modification specified by change to the function of constraint ci.

An ModifyConstraintNotAllowed error is thrown if modifying constraints is not supported by the model model.

Examples

```
modify(model, ci, ScalarConstantChange(10.0))
```

Objective Function

```
modify(model::ModelLike, ::ObjectiveFunction, change::AbstractFunctionModification)
```

Apply the modification specified by change to the objective function of model. To change the function completely, call set instead.

An ModifyObjectiveNotAllowed error is thrown if modifying objectives is not supported by the model model.

Examples

```
modify(model, ObjectiveFunction{ScalarAffineFunction{Float64}}(), ScalarConstantChange(10.0))
```

MathOptInterface.AbstractFunctionModification – Type.

```
AbstractFunctionModification
```

An abstract supertype for structs which specify partial modifications to functions, to be used for making small modifications instead of replacing the functions entirely.

MathOptInterface.ScalarConstantChange – Type.

```
ScalarConstantChange{T}(new_constant::T)
```

A struct used to request a change in the constant term of a scalar-valued function. Applicable to ScalarAffineFunction and ScalarQuadraticFunction.

MathOptInterface.VectorConstantChange – Type.

```
VectorConstantChange{T}(new_constant::Vector{T})
```

A struct used to request a change in the constant vector of a vector-valued function. Applicable to VectorAffineFunction and VectorQuadraticFunction.

MathOptInterface.ScalarCoefficientChange – Type.

```
ScalarCoefficientChange{T}(variable::VariableIndex, new_coefficient::T)
```

A struct used to request a change in the linear coefficient of a single variable in a scalar-valued function. Applicable to ScalarAffineFunction and ScalarQuadraticFunction.

MathOptInterface.MultirowChange – Type.

```
MultirowChange{T}(variable::VariableIndex, new_coefficients::Vector{Tuple{Int64, T}})
```

A struct used to request a change in the linear coefficients of a single variable in a vector-valued function. New coefficients are specified by `(output_index, coefficient)` tuples. Applicable to `VectorAffineFunction` and `VectorQuadraticFunction`.

39.6 Nonlinear programming

Types

MathOptInterface.AbstractNLPEvaluator – Type.

```
AbstractNLPEvaluator
```

Abstract supertype for the callback object that is used to query function values, derivatives, and expression graphs. It is used in `NLPBlock`.

MathOptInterface.NLPBoundsPair – Type.

```
NLPBoundsPair(lower,upper)
```

A struct holding a pair of lower and upper bounds. `-Inf` and `Inf` can be used to indicate no lower or upper bound, respectively.

MathOptInterface.NLPBlockData – Type.

```
struct NLPBlockData
    constraint_bounds::Vector{NLPBoundsPair}
    evaluator::AbstractNLPEvaluator
    has_objective::Bool
end
```

A `struct` encoding a set of nonlinear constraints of the form $lb \leq g(x) \leq ub$ and, if `has_objective == true`, a nonlinear objective function $f(x)$. `constraint_bounds` holds the pairs of lb and ub elements. Nonlinear objectives override any objective set by using the `ObjectiveFunction` attribute. The `evaluator` is a callback object that is used to query function values, derivatives, and expression graphs. If `has_objective == false`, then it is an error to query properties of the objective function, and in Hessian-of-the-Lagrangian queries, σ must be set to zero.

> **Note**
>
> Throughout the evaluator, all variables are ordered according to `ListOfVariableIndices`. Hence, MOI copies of nonlinear problems should be done with attention.

Attributes

MathOptInterface.NLPBlock – Type.

```
NLPBlock()
```

Holds the `NLPBlockData` that represents a set of nonlinear constraints, and optionally a nonlinear objective.

MathOptInterface.NLPBlockDual – Type.

```
NLPBlockDual(result_index::Int)
NLPBlockDual()
```

The Lagrange multipliers on the constraints from the `NLPBlock` in result `result_index`. If `result_index` is omitted, it is 1 by default.

`MathOptInterface.NLPBlockDualStart` – Type.

```
NLPBlockDualStart()
```

An initial assignment of the Lagrange multipliers on the constraints from the `NLPBlock` that the solver may use to warm-start the solve.

Functions

`MathOptInterface.initialize` – Function.

```
initialize(d::AbstractNLPEvaluator, requested_features::Vector{Symbol})
```

Must be called before any other methods. The vector `requested_features` lists features requested by the solver. These may include `:Grad` for gradients of the obejctive, f, `:Jac` for explicit Jacobians of constraints, g, `:JacVec` for Jacobian-vector products, `:HessVec` for Hessian-vector and Hessian-of-Lagrangian-vector products, `:Hess` for explicit Hessians and Hessian-of-Lagrangians, and `:ExprGraph` for expression graphs.

`MathOptInterface.features_available` – Function.

```
features_available(d::AbstractNLPEvaluator)
```

Returns the subset of features available for this problem instance, as a vector of symbols in the same format as in `initialize`.

`MathOptInterface.eval_objective` – Function.

```
eval_objective(d::AbstractNLPEvaluator, x)
```

Evaluate the objective $f(x)$, returning a scalar value.

`MathOptInterface.eval_constraint` – Function.

```
eval_constraint(d::AbstractNLPEvaluator, g, x)
```

Evaluate the constraint function $g(x)$, storing the result in the vector `g` which must be of the appropriate size.

`MathOptInterface.eval_objective_gradient` – Function.

```
eval_objective_gradient(d::AbstractNLPEvaluator, df, x)
```

Evaluate $\nabla f(x)$ as a dense vector, storing the result in the vector `df` which must be of the appropriate size.

`MathOptInterface.jacobian_structure` – Function.

```
jacobian_structure(d::AbstractNLPEvaluator)::Vector{Tuple{Int64,Int64}}
```

Returns the sparsity structure of the Jacobian matrix $J_g(x) = \begin{bmatrix} \nabla g_1(x) \\ \nabla g_2(x) \\ \vdots \\ \nabla g_m(x) \end{bmatrix}$ where g_i is the ith component of g. The sparsity structure is assumed to be independent of the point x. Returns a vector of tuples, `(row, column)`, where each indicates the position of a structurally nonzero element. These indices are not required to be sorted and can contain duplicates, in which case the solver should combine the corresponding elements by adding them together.

MathOptInterface.hessian_lagrangian_structure – Function.

```
hessian_lagrangian_structure(d::AbstractNLPEvaluator)::Vector{Tuple{Int64,Int64}}
```

Returns the sparsity structure of the Hessian-of-the-Lagrangian matrix $\nabla^2 f + \sum_{i=1}^m \nabla^2 g_i$ as a vector of tuples, where each indicates the position of a structurally nonzero element. These indices are not required to be sorted and can contain duplicates, in which case the solver should combine the corresponding elements by adding them together. Any mix of lower and upper-triangular indices is valid. Elements `(i,j)` and `(j,i)`, if both present, should be treated as duplicates.

MathOptInterface.eval_constraint_jacobian – Function.

```
eval_constraint_jacobian(d::AbstractNLPEvaluator, J, x)
```

Evaluates the sparse Jacobian matrix $J_g(x) = \begin{bmatrix} \nabla g_1(x) \\ \nabla g_2(x) \\ \vdots \\ \nabla g_m(x) \end{bmatrix}$. The result is stored in the vector `J` in the same order as the indices returned by `jacobian_structure`.

MathOptInterface.eval_constraint_jacobian_product – Function.

```
eval_constraint_jacobian_product(d::AbstractNLPEvaluator, y, x, w)
```

Computes the Jacobian-vector product $J_g(x)w$, storing the result in the vector `y`.

MathOptInterface.eval_constraint_jacobian_transpose_product – Function.

```
eval_constraint_jacobian_transpose_product(d::AbstractNLPEvaluator, y, x, w)
```

Computes the Jacobian-transpose-vector product $J_g(x)^T w$, storing the result in the vector `y`.

MathOptInterface.eval_hessian_lagrangian – Function.

```
eval_hessian_lagrangian(d::AbstractNLPEvaluator, H, x, σ, μ)
```

Given scalar weight `σ` and vector of constraint weights `μ`, computes the sparse Hessian-of-the-Lagrangian matrix $\sigma\nabla^2 f(x) + \sum_{i=1}^m \mu_i \nabla^2 g_i(x)$, storing the result in the vector `H` in the same order as the indices returned by `hessian_lagrangian_structure`.

MathOptInterface.eval_hessian_lagrangian_product – Function.

```
eval_hessian_lagrangian_product(d::AbstractNLPEvaluator, h, x, v, σ, μ)
```

Given scalar weight `σ` and vector of constraint weights `μ`, computes the Hessian-of-the-Lagrangian-vector product $\left(\sigma\nabla^2 f(x) + \sum_{i=1}^m \mu_i \nabla^2 g_i(x)\right) v$, storing the result in the vector `h`.

MathOptInterface.objective_expr – Function.

```
objective_expr(d::AbstractNLPEvaluator)
```

Returns an expression graph for the objective function as a standard Julia `Expr` object. All sums and products are flattened out as simple `Expr(:+,...)` and `Expr(:*,...)` objects. The symbol `x` is used as a placeholder for the vector of decision variables. No other undefined symbols are permitted; coefficients are embedded as explicit values. For example, the expression $x_1 + \sin(x_2 / \exp(x_3))$ would be represented as the Julia object `:(x[1] + sin(x[2]/exp(x[3])))`. Each integer index is wrapped in a `VariableIndex`. See the Julia manual for more information on the structure of `Expr` objects. There are currently no restrictions on recognized functions; typically these will be built-in Julia functions like `^`, `exp`, `log`, `cos`, `tan`, `sqrt`, etc., but modeling interfaces may choose to extend these basic functions.

MathOptInterface.constraint_expr – Function.

```
constraint_expr(d::AbstractNLPEvaluator, i)
```

Returns an expression graph for the ith constraint in the same format as described above, with an additional comparison operator indicating the sense of and bounds on the constraint. The right-hand side of the comparison must be a constant; that is, `:(x[1]^3 <= 1)` is allowed, while `:(1 <= x[1]^3)` is not valid. Double-sided constraints are allowed, in which case both the lower bound and upper bounds should be constants; for example, `:(-1 <= cos(x[1]) + sin(x[2]) <= 1)` is valid.

39.7 Callbacks

MathOptInterface.AbstractCallback – Type.

```
abstract type AbstractCallback <: AbstractModelAttribute end
```

Abstract type for a model attribute representing a callback function. The value set to subtypes of `AbstractCallback` is a function that may be called during `optimize!`. As `optimize!` is in progress, the result attributes (i.e, the attributes `attr` such that `is_set_by_optimize(attr)`) may not be accessible from the callback, hence trying to get result attributes might throw a `OptimizeInProgress` error.

At most one callback of each type can be registered. If an optimizer already has a function for a callback type, and the user registers a new function, then the old one is replaced.

The value of the attribute should be a function taking only one argument, commonly called `callback_data`, that can be used for instance in `LazyConstraintCallback`, `HeuristicCallback` and `UserCutCallback`.

MathOptInterface.AbstractSubmittable – Type.

```
AbstractSubmittable
```

Abstract supertype for objects that can be submitted to the model.

MathOptInterface.submit – Function.

```
submit(optimizer::AbstractOptimizer, sub::AbstractSubmittable,
       values...)::Nothing
```

Submit `values` to the submittable `sub` of the optimizer `optimizer`.

An `UnsupportedSubmittable` error is thrown if `model` does not support the attribute `attr` (see `supports`) and a `SubmitNotAllowed` error is thrown if it supports the submittable `sub` but it cannot be submitted.

Attributes

MathOptInterface.CallbackNodeStatus – Type.

```
CallbackNodeStatus(callback_data)
```

An optimizer attribute describing the (in)feasibility of the primal solution available from CallbackVariablePrimal during a callback identified by callback_data.

Returns a CallbackNodeStatusCode Enum.

MathOptInterface.CallbackNodeStatusCode – Type.

```
CallbackNodeStatusCode
```

An Enum of possible return values from calling get with CallbackNodeStatus.

Possible values are:

- CALLBACK_NODE_STATUS_INTEGER: the primal solution available from CallbackVariablePrimal is integer feasible.
- CALLBACK_NODE_STATUS_FRACTIONAL: the primal solution available from CallbackVariablePrimal is integer infeasible.
- CALLBACK_NODE_STATUS_UNKNOWN: the primal solution available from CallbackVariablePrimal might be integer feasible or infeasible.

MathOptInterface.CallbackVariablePrimal – Type.

```
CallbackVariablePrimal(callback_data)
```

A variable attribute for the assignment to some primal variable's value during the callback identified by callback_data.

Lazy constraints

MathOptInterface.LazyConstraintCallback – Type.

```
LazyConstraintCallback() <: AbstractCallback
```

The callback can be used to reduce the feasible set given the current primal solution by submitting a LazyConstraint. For instance, it may be called at an incumbent of a mixed-integer problem. Note that there is no guarantee that the callback is called at every feasible primal solution.

The current primal solution is accessed through CallbackVariablePrimal. Trying to access other result attributes will throw OptimizeInProgress as discussed in AbstractCallback.

Examples

```
x = MOI.add_variables(optimizer, 8)
MOI.set(optimizer, MOI.LazyConstraintCallback(), callback_data -> begin
    sol = MOI.get(optimizer, MOI.CallbackVariablePrimal(callback_data), x)
    if # should add a lazy constraint
        func = # computes function
        set = # computes set
        MOI.submit(optimizer, MOI.LazyConstraint(callback_data), func, set)
    end
end)
```

MathOptInterface.LazyConstraint – Type.

```
LazyConstraint(callback_data)
```

Lazy constraint func-in-set submitted as `func, set`. The optimal solution returned by VariablePrimal will satisfy all lazy constraints that have been submitted.

This can be submitted only from the LazyConstraintCallback. The field `callback_data` is a solver-specific callback type that is passed as the argument to the feasible solution callback.

Examples

Suppose x and y are VariableIndexs of `optimizer`. To add a `LazyConstraint` for `2x + 3y <= 1`, write

```
func = 2.0x + 3.0y
set = MOI.LessThan(1.0)
MOI.submit(optimizer, MOI.LazyConstraint(callback_data), func, set)
```

inside a LazyConstraintCallback of data `callback_data`.

User cuts

MathOptInterface.UserCutCallback – Type.

```
UserCutCallback() <: AbstractCallback
```

The callback can be used to submit UserCut given the current primal solution. For instance, it may be called at fractional (i.e., non-integer) nodes in the branch and bound tree of a mixed-integer problem. Note that there is not guarantee that the callback is called everytime the solver has an infeasible solution.

The infeasible solution is accessed through CallbackVariablePrimal. Trying to access other result attributes will throw OptimizeInProgress as discussed in AbstractCallback.

Examples

```
x = MOI.add_variables(optimizer, 8)
MOI.set(optimizer, MOI.UserCutCallback(), callback_data -> begin
    sol = MOI.get(optimizer, MOI.CallbackVariablePrimal(callback_data), x)
    if # can find a user cut
        func = # computes function
        set = # computes set
        MOI.submit(optimizer, MOI.UserCut(callback_data), func, set)
    end
end
```

MathOptInterface.UserCut – Type.

```
UserCut(callback_data)
```

Constraint func-to-set suggested to help the solver detect the solution given by CallbackVariablePrimal as infeasible. The cut is submitted as `func, set`. Typically CallbackVariablePrimal will violate integrality constraints, and a cut would be of the form ScalarAffineFunction-in-LessThan or ScalarAffineFunction-in-GreaterThan. Note that, as opposed to LazyConstraint, the provided constraint cannot modify the feasible set, the constraint should be redundant, e.g., it may be a consequence of affine and integrality constraints.

This can be submitted only from the UserCutCallback. The field `callback_data` is a solver-specific callback type that is passed as the argument to the infeasible solution callback.

Note that the solver may silently ignore the provided constraint.

Heuristic solutions

MathOptInterface.HeuristicCallback – Type.

```
HeuristicCallback() <: AbstractCallback
```

The callback can be used to submit HeuristicSolution given the current primal solution. For instance, it may be called at fractional (i.e., non-integer) nodes in the branch and bound tree of a mixed-integer problem. Note that there is not guarantee that the callback is called everytime the solver has an infeasible solution.

The current primal solution is accessed through CallbackVariablePrimal. Trying to access other result attributes will throw OptimizeInProgress as discussed in AbstractCallback.

Examples

```
x = MOI.add_variables(optimizer, 8)
MOI.set(optimizer, MOI.HeuristicCallback(), callback_data -> begin
    sol = MOI.get(optimizer, MOI.CallbackVariablePrimal(callback_data), x)
    if # can find a heuristic solution
        values = # computes heuristic solution
        MOI.submit(optimizer, MOI.HeuristicSolution(callback_data), x,
                   values)
    end
end
```

MathOptInterface.HeuristicSolutionStatus – Type.

```
HeuristicSolutionStatus
```

An Enum of possible return values for submit with HeuristicSolution. This informs whether the heuristic solution was accepted or rejected. Possible values are:

- HEURISTIC_SOLUTION_ACCEPTED: The heuristic solution was accepted.
- HEURISTIC_SOLUTION_REJECTED: The heuristic solution was rejected.
- HEURISTIC_SOLUTION_UNKNOWN: No information available on the acceptance.

MathOptInterface.HeuristicSolution – Type.

```
HeuristicSolution(callback_data)
```

Heuristically obtained feasible solution. The solution is submitted as `variables, values` where `values[i]` gives the value of `variables[i]`, similarly to set. The submit call returns a HeuristicSolutionStatus indicating whether the provided solution was accepted or rejected.

This can be submitted only from the HeuristicCallback. The field `callback_data` is a solver-specific callback type that is passed as the argument to the heuristic callback.

Some solvers require a complete solution, others only partial solutions.

39.8 Errors

When an MOI call fails on a model, precise errors should be thrown when possible instead of simply calling `error` with a message. The docstrings for the respective methods describe the errors that the implementation should thrown in certain situations. This error-reporting system allows code to distinguish between internal errors (that should be shown to the user) and unsupported operations which may have automatic workarounds.

When an invalid index is used in an MOI call, an `InvalidIndex` should be thrown:

`MathOptInterface.InvalidIndex` – Type.

```
struct InvalidIndex{IndexType<:Index} <: Exception
    index::IndexType
end
```

An error indicating that the index `index` is invalid.

When an invalid result index is used to retrieve an attribute, a `ResultIndexBoundsError` should be thrown:

`MathOptInterface.ResultIndexBoundsError` – Type.

```
struct ResultIndexBoundsError{AttrType} <: Exception
    attr::AttrType
    result_count::Int
end
```

An error indicating that the requested attribute `attr` could not be retrieved, because the solver returned too few results compared to what was requested. For instance, the user tries to retrieve `VariablePrimal(2)` when only one solution is available, or when the model is infeasible and has no solution.

See also: `check_result_index_bounds`.

`MathOptInterface.check_result_index_bounds` – Function.

```
check_result_index_bounds(model::ModelLike, attr)
```

This function checks whether enough results are available in the `model` for the requested `attr`, using its `result_index` field. If the model does not have sufficient results to answer the query, it throws a `ResultIndexBoundsError`.

As discussed in JuMP mapping, for scalar constraint with a nonzero function constant, a `ScalarFunctionConstantNotZero` exception may be thrown:

`MathOptInterface.ScalarFunctionConstantNotZero` – Type.

```
struct ScalarFunctionConstantNotZero{T, F, S} <: Exception
    constant::T
end
```

An error indicating that the constant part of the function in the constraint F-in-S is nonzero.

Some `VariableIndex` constraints cannot be combined on the same variable:

`MathOptInterface.LowerBoundAlreadySet` – Type.

```
LowerBoundAlreadySet{S1, S2}
```

Error thrown when setting a `VariableIndex`-in-`S2` when a `VariableIndex`-in-`S1` has already been added and the sets `S1`, `S2` both set a lower bound, i.e. they are `EqualTo`, `GreaterThan`, `Interval`, `Semicontinuous` or `Semiinteger`.

`MathOptInterface.UpperBoundAlreadySet` – Type.

```
UpperBoundAlreadySet{S1, S2}
```

Error thrown when setting a `VariableIndex`-in-`S2` when a `VariableIndex`-in-`S1` has already been added and the sets `S1`, `S2` both set an upper bound, i.e. they are `EqualTo`, `LessThan`, `Interval`, `Semicontinuous` or `Semiinteger`.

As discussed in `AbstractCallback`, trying to `get` attributes inside a callback may throw:

`MathOptInterface.OptimizeInProgress` – Type.

```
struct OptimizeInProgress{AttrType<:AnyAttribute} <: Exception
    attr::AttrType
end
```

Error thrown from optimizer when `MOI.get(optimizer, attr)` is called inside an `AbstractCallback` while it is only defined once `optimize!` has completed. This can only happen when `is_set_by_optimize(attr)` is `true`.

Trying to submit the wrong type of `AbstractSubmittable` inside an `AbstractCallback` (e.g., a `UserCut` inside a `LazyConstraintCallback`) will throw:

`MathOptInterface.InvalidCallbackUsage` – Type.

```
struct InvalidCallbackUsage{C, S} <: Exception
    callback::C
    submittable::S
end
```

An error indicating that `submittable` cannot be submitted inside `callback`.

For example, `UserCut` cannot be submitted inside `LazyConstraintCallback`.

The rest of the errors defined in MOI fall in two categories represented by the following two abstract types:

`MathOptInterface.UnsupportedError` – Type.

```
UnsupportedError <: Exception
```

Abstract type for error thrown when an element is not supported by the model.

`MathOptInterface.NotAllowedError` – Type.

```
NotAllowedError <: Exception
```

Abstract type for error thrown when an operation is supported but cannot be applied in the current state of the model.

The different `UnsupportedError` and `NotAllowedError` are the following errors:

`MathOptInterface.UnsupportedAttribute` – Type.

```
struct UnsupportedAttribute{AttrType} <: UnsupportedError
    attr::AttrType
    message::String
end
```

An error indicating that the attribute `attr` is not supported by the model, i.e. that `supports` returns `false`.

`MathOptInterface.SetAttributeNotAllowed` – Type.

```
struct SetAttributeNotAllowed{AttrType} <: NotAllowedError
    attr::AttrType
    message::String # Human-friendly explanation why the attribute cannot be set
end
```

An error indicating that the attribute `attr` is supported (see `supports`) but cannot be set for some reason (see the error string).

`MathOptInterface.AddVariableNotAllowed` – Type.

```
struct AddVariableNotAllowed <: NotAllowedError
    message::String # Human-friendly explanation why the attribute cannot be set
end
```

An error indicating that variables cannot be added to the model.

`MathOptInterface.UnsupportedConstraint` – Type.

```
struct UnsupportedConstraint{F<:AbstractFunction, S<:AbstractSet} <: UnsupportedError
    message::String # Human-friendly explanation why the attribute cannot be set
end
```

An error indicating that constraints of type F-in-S are not supported by the model, i.e. that `supports_constraint` returns `false`.

`MathOptInterface.AddConstraintNotAllowed` – Type.

```
struct AddConstraintNotAllowed{F<:AbstractFunction, S<:AbstractSet} <: NotAllowedError
    message::String # Human-friendly explanation why the attribute cannot be set
end
```

An error indicating that constraints of type F-in-S are supported (see `supports_constraint`) but cannot be added.

`MathOptInterface.ModifyConstraintNotAllowed` – Type.

```
struct ModifyConstraintNotAllowed{F<:AbstractFunction, S<:AbstractSet,
                                  C<:AbstractFunctionModification} <: NotAllowedError
    constraint_index::ConstraintIndex{F, S}
    change::C
    message::String
end
```

An error indicating that the constraint modification `change` cannot be applied to the constraint of index `ci`.

`MathOptInterface.ModifyObjectiveNotAllowed` – Type.

```
struct ModifyObjectiveNotAllowed{C<:AbstractFunctionModification} <: NotAllowedError
    change::C
    message::String
end
```

An error indicating that the objective modification `change` cannot be applied to the objective.

`MathOptInterface.DeleteNotAllowed` – Type.

```
struct DeleteNotAllowed{IndexType <: Index} <: NotAllowedError
    index::IndexType
    message::String
end
```

An error indicating that the index `index` cannot be deleted.

`MathOptInterface.UnsupportedSubmittable` – Type.

```
struct UnsupportedSubmittable{SubmitType} <: UnsupportedError
    sub::SubmitType
    message::String
end
```

An error indicating that the submittable `sub` is not supported by the model, i.e. that `supports` returns `false`.

`MathOptInterface.SubmitNotAllowed` – Type.

```
struct SubmitNotAllowed{SubmitTyp<:AbstractSubmittable} <: NotAllowedError
    sub::SubmitType
    message::String # Human-friendly explanation why the attribute cannot be set
end
```

An error indicating that the submittable `sub` is supported (see `supports`) but cannot be added for some reason (see the error string).

Note that setting the `ConstraintFunction` of a `VariableIndex` constraint is not allowed:

`MathOptInterface.SettingVariableIndexNotAllowed` – Type.

```
SettingVariableIndexNotAllowed()
```

Error type that should be thrown when the user calls `set` to change the `ConstraintFunction` of a `VariableIndex` constraint.

Chapter 40

Submodules

40.1 Benchmarks

Overview

The `Benchmarks` submodule

To aid the development of efficient solver wrappers, MathOptInterface provides benchmarking functionality. Benchmarking a wrapper follows a two-step process.

First, prior to making changes, run and save the benchmark results on a given benchmark suite as follows:

```
using SolverPackage  # Replace with your choice of solver.

using MathOptInterface
const MOI = MathOptInterface

suite = MOI.Benchmarks.suite() do
    SolverPackage.Optimizer()
end

MOI.Benchmarks.create_baseline(
    suite, "current"; directory = "/tmp", verbose = true
)
```

Use the `exclude` argument to `Benchmarks.suite` to exclude benchmarks that the solver doesn't support.

Second, after making changes to the package, re-run the benchmark suite and compare to the prior saved results:

```
using SolverPackage, MathOptInterface

const MOI = MathOptInterface

suite = MOI.Benchmarks.suite() do
    SolverPackage.Optimizer()
end

MOI.Benchmarks.compare_against_baseline(
    suite, "current"; directory = "/tmp", verbose = true
)
```

This comparison will create a report detailing improvements and regressions.

API Reference

Benchmarks Functions to help benchmark the performance of solver wrappers. See The Benchmarks submodule for more details.

MathOptInterface.Benchmarks.suite – Function.

```
suite(
    new_model::Function;
    exclude::Vector{Regex} = Regex[]
)
```

Create a suite of benchmarks. `new_model` should be a function that takes no arguments, and returns a new instance of the optimizer you wish to benchmark.

Use `exclude` to exclude a subset of benchmarks.

Examples

```
suite() do
    GLPK.Optimizer()
end
suite(exclude = [r"delete"]) do
    Gurobi.Optimizer(OutputFlag=0)
end
```

MathOptInterface.Benchmarks.create_baseline – Function.

```
create_baseline(suite, name::String; directory::String = ""; kwargs...)
```

Run all benchmarks in `suite` and save to files called `name` in `directory`.

Extra `kwargs` are based to `BenchmarkTools.run`.

Examples

```
my_suite = suite(() -> GLPK.Optimizer())
create_baseline(my_suite, "glpk_master"; directory = "/tmp", verbose = true)
```

MathOptInterface.Benchmarks.compare_against_baseline – Function.

```
compare_against_baseline(
    suite, name::String; directory::String = "",
    report_filename::String = "report.txt"
)
```

Run all benchmarks in `suite` and compare against files called `name` in `directory` that were created by a call to `create_baseline`.

A report summarizing the comparison is written to `report_filename` in `directory`.

Extra `kwargs` are based to `BenchmarkTools.run`.

Examples

```
my_suite = suite(() -> GLPK.Optimizer())
compare_against_baseline(
    my_suite, "glpk_master"; directory = "/tmp", verbose = true
)
```

40.2 Bridges

Overview

The Bridges submodule

The `Bridges` module simplifies the process of converting models between equivalent formulations.

> **Tip**
>
> Read our paper for more details on how bridges are implemented.

Why bridges? A constraint can often be written in a number of equivalent formulations. For example, the constraint $l \leq a^\top x \leq u$ (`ScalarAffineFunction`-in-`Interval`) could be re-formulated as two constraints: $a^\top x \geq l$ (`ScalarAffineFunction`-in-`GreaterThan`) and $a^\top x \leq u$ (`ScalarAffineFunction`-in-`LessThan`). An alternative re-formulation is to add a dummy variable y with the constraints $l \leq y \leq u$ (`VariableIndex`-in-`Interval`) and $a^\top x - y = 0$ (`ScalarAffineFunction`-in-`EqualTo`).

To avoid each solver having to code these transformations manually, MathOptInterface provides bridges.

A bridge is a small transformation from one constraint type to another (potentially collection of) constraint type.

Because these bridges are included in MathOptInterface, they can be re-used by any optimizer. Some bridges also implement constraint modifications and constraint primal and dual translations.

Several bridges can be used in combination to transform a single constraint into a form that the solver may understand. Choosing the bridges to use takes the form of finding a shortest path in the hypergraph of bridges. The methodology is detailed in the MOI paper.

The three types of bridges There are three types of bridges in MathOptInterface:

1. Constraint bridges
2. Variable bridges
3. Objective bridges

Constraint bridges Constraint bridges convert constraints formulated by the user into an equivalent form supported by the solver. Constraint bridges are subtypes of `Bridges.Constraint.AbstractBridge`.

The equivalent formulation may add constraints (and possibly also variables) in the underlying model.

In particular, constraint bridges can focus on rewriting the function of a constraint, and do not change the set. Function bridges are subtypes of `Bridges.Constraint.AbstractFunctionConversionBridge`.

Read the list of implemented constraint bridges for more details on the types of transformations that are available. Function bridges are `Bridges.Constraint.ScalarFunctionizeBridge` and `Bridges.Constraint.VectorFunctionizeBri`

Variable bridges Variable bridges convert variables added by the user, either free with `add_variable`/`add_variables`, or constrained with `add_constrained_variable`/`add_constrained_variables`, into an equivalent form supported by the solver. Variable bridges are subtypes of `Bridges.Variable.AbstractBridge`.

The equivalent formulation may add constraints (and possibly also variables) in the underlying model.

Read the list of implemented variable bridges for more details on the types of transformations that are available.

Objective bridges Objective bridges convert the `ObjectiveFunction` set by the user into an equivalent form supported by the solver. Objective bridges are subtypes of `Bridges.Objective.AbstractBridge`.

The equivalent formulation may add constraints (and possibly also variables) in the underlying model.

Read the list of implemented objective bridges for more details on the types of transformations that are available.

Bridges.full_bridge_optimizer

> **Tip**
>
> Unless you have an advanced use-case, this is probably the only function you need to care about.

To enable the full power of MathOptInterface's bridges, wrap an `optimizer` in a `Bridges.full_bridge_optimizer`.

```
julia> inner_optimizer = MOI.Utilities.Model{Float64}()
MOIU.Model{Float64}

julia> optimizer = MOI.Bridges.full_bridge_optimizer(inner_optimizer, Float64)
MOIB.LazyBridgeOptimizer{MOIU.Model{Float64}}
with 0 variable bridges
with 0 constraint bridges
with 0 objective bridges
with inner model MOIU.Model{Float64}
```

That's all you have to do! Use `optimizer` as normal, and bridging will happen lazily behind the scenes. By lazily, we mean that bridging will only happen if the constraint is not supported by the `inner_optimizer`.

> **Info**
>
> Most bridges are added by default in `Bridges.full_bridge_optimizer`. However, for technical reasons, some bridges are not added by default. Three examples include `Bridges.Constraint.SOCtoPSDBridge`, `Bridges.Constraint.SOCtoNonConvexQuadBridge` and `Bridges.Constraint.RSOCtoNonConvexQuadBridge`. See the docs of those bridges for more information.

Add a single bridge If you don't want to use `Bridges.full_bridge_optimizer`, you can wrap an optimizer in a single bridge.

However, this will force the constraint to be bridged, even if the `inner_optimizer` supports it.

```
julia> inner_optimizer = MOI.Utilities.Model{Float64}()
MOIU.Model{Float64}

julia> optimizer = MOI.Bridges.Constraint.SplitInterval{Float64}(inner_optimizer)
MOIB.Constraint.SingleBridgeOptimizer{MOIB.Constraint.SplitIntervalBridge{Float64, F, S, LS, US}
↪  where {F<:MOI.AbstractFunction, S<:MOI.AbstractSet, LS<:MOI.AbstractSet, US<:MOI.AbstractSet},
↪  MOIU.Model{Float64}}
with 0 constraint bridges
with inner model MOIU.Model{Float64}

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

julia> MOI.add_constraint(optimizer, x, MOI.Interval(0.0, 1.0))
```

```
MathOptInterface.ConstraintIndex{MathOptInterface.VariableIndex,
↪  MathOptInterface.Interval{Float64}}(1)

julia> MOI.get(optimizer, MOI.ListOfConstraintTypesPresent())
1-element Vector{Tuple{Type, Type}}:
 (MathOptInterface.VariableIndex, MathOptInterface.Interval{Float64})

julia> MOI.get(inner_optimizer, MOI.ListOfConstraintTypesPresent())
2-element Vector{Tuple{Type, Type}}:
 (MathOptInterface.VariableIndex, MathOptInterface.GreaterThan{Float64})
 (MathOptInterface.VariableIndex, MathOptInterface.LessThan{Float64})
```

Bridges.LazyBridgeOptimizer If you don't want to use `Bridges.full_bridge_optimizer`, but you need more than a single bridge (or you want the bridging to happen lazily), you can manually construct a `Bridges.LazyBridgeOptimize`

First, wrap an inner optimizer:

```
julia> inner_optimizer = MOI.Utilities.Model{Float64}()
MOIU.Model{Float64}

julia> optimizer = MOI.Bridges.LazyBridgeOptimizer(inner_optimizer)
MOIB.LazyBridgeOptimizer{MOIU.Model{Float64}}
with 0 variable bridges
with 0 constraint bridges
with 0 objective bridges
with inner model MOIU.Model{Float64}
```

Then use `Bridges.add_bridge` to add individual bridges:

```
julia> MOI.Bridges.add_bridge(optimizer, MOI.Bridges.Constraint.SplitIntervalBridge{Float64})

julia> MOI.Bridges.add_bridge(optimizer, MOI.Bridges.Objective.FunctionizeBridge{Float64})
```

Now the constraints will be bridged only if needed:

```
julia> x = MOI.add_variable(optimizer)
MOI.VariableIndex(1)

julia> MOI.add_constraint(optimizer, x, MOI.Interval(0.0, 1.0))
MathOptInterface.ConstraintIndex{MathOptInterface.VariableIndex,
↪  MathOptInterface.Interval{Float64}}(1)

julia> MOI.get(optimizer, MOI.ListOfConstraintTypesPresent())
1-element Vector{Tuple{Type, Type}}:
 (MathOptInterface.VariableIndex, MathOptInterface.Interval{Float64})

julia> MOI.get(inner_optimizer, MOI.ListOfConstraintTypesPresent())
1-element Vector{Tuple{Type, Type}}:
 (MathOptInterface.VariableIndex, MathOptInterface.Interval{Float64})
```

Implementation

Bridge interface

A bridge should implement the following functions to be usable by a bridge optimizer:

MathOptInterface.Bridges.added_constrained_variable_types – Function.

```
added_constrained_variable_types(
    BT::Type{<:Variable.AbstractBridge},
)::Vector{Tuple{Type}}
```

Return a list of the types of constrained variables that bridges of concrete type BT add. This is used by the LazyBridgeOptimizer.

MathOptInterface.Bridges.added_constraint_types – Function.

```
added_constraint_types(
    BT::Type{<:Constraint.AbstractBridge},
)::Vector{Tuple{Type, Type}}
```

Return a list of the types of constraints that bridges of concrete type BT add. This is used by the LazyBridgeOptimizer.

Additionally, variable bridges should implement:

MathOptInterface.Bridges.Variable.supports_constrained_variable – Function.

```
supports_constrained_variable(
    ::Type{<:AbstractBridge},
    ::Type{<:MOI.AbstractSet},
)::Bool
```

Return a `Bool` indicating whether the bridges of type BT support bridging constrained variables in S.

MathOptInterface.Bridges.Variable.concrete_bridge_type – Function.

```
concrete_bridge_type(
    BT::Type{<:AbstractBridge},
    S::Type{<:MOI.AbstractSet},
)::Type
```

Return the concrete type of the bridge supporting variables in S constraints. This function can only be called if `MOI.supports_constrained_variable(BT, S)` is `true`.

Examples

As a variable in MathOptInterface.GreaterThan is bridged into variables in MathOptInterface.Nonnegatives by the VectorizeBridge:

```
MOI.Bridges.Variable.concrete_bridge_type(
    MOI.Bridges.Variable.VectorizeBridge{Float64},
    MOI.GreaterThan{Float64},
)

# output

MathOptInterface.Bridges.Variable.VectorizeBridge{Float64, MathOptInterface.Nonnegatives}
```

MathOptInterface.Bridges.Variable.bridge_constrained_variable – Function.

```
bridge_constrained_variable(
    BT::Type{<:AbstractBridge},
    model::MOI.ModelLike,
    set::MOI.AbstractSet,
)
```

Bridge the constrained variable in `set` using bridge `BT` to `model` and returns a bridge object of type `BT`. The bridge type `BT` should be a concrete type, that is, all the type parameters of the bridge should be set. Use `concrete_bridge_type` to obtain a concrete type for given set types.

constraint bridges should implement:

`MathOptInterface.supports_constraint` – Method.

```
MOI.supports_constraint(
    BT::Type{<:AbstractBridge},
    F::Type{<:MOI.AbstractFunction},
    S::Type{<:MOI.AbstractSet},
)::Bool
```

Return a `Bool` indicating whether the bridges of type `BT` support bridging `F`-in-`S` constraints.

`MathOptInterface.Bridges.Constraint.concrete_bridge_type` – Function.

```
concrete_bridge_type(
    BT::Type{<:AbstractBridge},
    F::Type{<:MOI.AbstractFunction},
    S::Type{<:MOI.AbstractSet}
)::Type
```

Return the concrete type of the bridge supporting `F`-in-`S` constraints. This function can only be called if `MOI.supports_constraint(BT, F, S)` is `true`.

Examples

As a `MathOptInterface.VariableIndex`-in-`MathOptInterface.Interval` constraint is bridged into a `MathOptInterface.Var` in-`MathOptInterface.GreaterThan` and a `MathOptInterface.VariableIndex`-in-`MathOptInterface.LessThan` by the `SplitIntervalBridge`:

```
MOI.Bridges.Constraint.concrete_bridge_type(
    MOI.Bridges.Constraint.SplitIntervalBridge{Float64},
    MOI.VariableIndex,
    MOI.Interval{Float64},
)

# output

MathOptInterface.Bridges.Constraint.SplitIntervalBridge{Float64, MathOptInterface.VariableIndex,
↪  MathOptInterface.Interval{Float64}, MathOptInterface.GreaterThan{Float64},
↪  MathOptInterface.LessThan{Float64}}
```

`MathOptInterface.Bridges.Constraint.bridge_constraint` – Function.

```
bridge_constraint(
    BT::Type{<:AbstractBridge},
    model::MOI.ModelLike,
    func::AbstractFunction,
    set::MOI.AbstractSet,
)
```

Bridge the constraint `func-in-set` using bridge BT to `model` and returns a bridge object of type BT. The bridge type BT should be a concrete type, that is, all the type parameters of the bridge should be set. Use `concrete_bridge_type` to obtain a concrete type for given function and set types.

and objective bridges should implement:

`MathOptInterface.Bridges.set_objective_function_type` – Function.

```
set_objective_function_type(
    BT::Type{<:Objective.AbstractBridge},
)::Type{<:MOI.AbstractScalarFunction}
```

Return the type of objective function that bridges of concrete type BT set. This is used by the `LazyBridgeOptimizer`.

`MathOptInterface.Bridges.Objective.concrete_bridge_type` – Function.

```
concrete_bridge_type(
    BT::Type{<:MOI.Bridges.Objective.AbstractBridge},
    F::Type{<:MOI.AbstractScalarFunction},
)::Type
```

Return the concrete type of the bridge supporting objective functions of type `F`. This function can only be called if `MOI.supports_objective_function(BT, F)` is `true`.

`MathOptInterface.Bridges.Objective.bridge_objective` – Function.

```
bridge_objective(
    BT::Type{<:MOI.Bridges.Objective.AbstractBridge},
    model::MOI.ModelLike,
    func::MOI.AbstractScalarFunction,
)
```

Bridge the objective function `func` using bridge BT to `model` and returns a bridge object of type BT. The bridge type BT should be a concrete type, that is, all the type parameters of the bridge should be set. Use `concrete_bridge_type` to obtain a concrete type for a given function type.

When querying the `NumberOfVariables`, `NumberOfConstraints` `ListOfVariableIndices`, and `ListOfConstraintIndices`, the variables and constraints created by the bridges in the underlying model are hidden by the bridge optimizer. For this purpose, the bridge should provide access to the variables and constraints it has created by implementing the following methods of `get`:

`MathOptInterface.get` – Method.

```
MOI.get(b::AbstractBridge, ::MOI.NumberOfVariables)
```

The number of variables created by the bridge `b` in the model.

`MathOptInterface.get` – Method.

```
MOI.get(b::AbstractBridge, ::MOI.ListOfVariableIndices)
```

The list of variables created by the bridge `b` in the model.

`MathOptInterface.get` – Method.

```
MOI.get(b::AbstractBridge, ::MOI.NumberOfConstraints{F, S}) where {F, S}
```

The number of constraints of the type F-in-S created by the bridge `b` in the model.

MathOptInterface.get – Method.

```
MOI.get(b::AbstractBridge, ::MOI.ListOfConstraintIndices{F, S}) where {F, S}
```

A Vector{ConstraintIndex{F,S}} with indices of all constraints of type F-inS created by the bride b in the model (i.e., of length equal to the value of NumberOfConstraints{F,S}()).

SetMap bridges

Implementing a constraint bridge relying on linear transformation between two sets is easier thanks to the SetMap interface. The bridge simply needs to be a subtype of [Bridges.Variable.SetMapBridge] for a variable bridge and [Bridges.Constraint.SetMapBridge] for a constraint bridge and the linear transformation is represented with Bridges.map_set, Bridges.map_function, Bridges.inverse_map_set, Bridges.inverse_map_function, Bridges.adjoint_map_function and Bridges.inverse_adjoint_map_function. Note that the implementing last 4 methods is optional in the sense that if they are not implemented, bridging constraint would still work but some features would be missing as described in the docstrings. See [L20, Section 2.1.2] for more details including [L20, Example 2.1.1] that illustrates the idea for Bridges.Variable.SOCtoRSOCBridge, Bridges.Variable.RSOCtoSOCBr Bridges.Constraint.SOCtoRSOCBridge and Bridges.Constraint.RSOCtoSOCBridge.

[L20] Legat, Benoît. Set Programming: Theory and Computation. PhD thesis. 2020.

API Reference

Bridges

MathOptInterface.Bridges.AbstractBridge – Type.

```
AbstractBridge
```

Represents a bridged constraint or variable in a MathOptInterface.Bridges.AbstractBridgeOptimizer. It contains the indices of the variables and constraints that it has created in the model. These can be obtained using MathOptInterface.NumberOfVariables, MathOptInterface.ListOfVariableIndices, MathOptInterface.Nu and MathOptInterface.ListOfConstraintIndices using MathOptInterface.get with the bridge in place of the MathOptInterface.ModelLike. Attributes of the bridged model such as MathOptInterface.ConstraintDual and MathOptInterface.ConstraintPrimal, can be obtained using MathOptInterface.get with the bridge in place of the constraint index. These calls are used by the MathOptInterface.Bridges.AbstractBridgeOptimizer to communicate with the bridge so they should be implemented by the bridge.

MathOptInterface.Bridges.AbstractBridgeOptimizer – Type.

```
AbstractBridgeOptimizer
```

A bridge optimizer applies given constraint bridges to a given optimizer thus extending the types of supported constraints. The attributes of the inner optimizer are automatically transformed to make the bridges transparent, e.g. the variables and constraints created by the bridges are hidden.

By convention, the inner optimizer should be stored in a model field and the dictionary mapping constraint indices to bridges should be stored in a bridges field. If a bridge optimizer deviates from these conventions, it should implement the functions MOI.optimize! and bridge respectively.

MathOptInterface.Bridges.LazyBridgeOptimizer – Type.

```
LazyBridgeOptimizer{OT<:MOI.ModelLike} <: AbstractBridgeOptimizer
```

The LazyBridgeOptimizer combines several bridges, which are added using the add_bridge function.

Whenever a constraint is added, it only attempts to bridge it if it is not supported by the internal model (hence its name Lazy).

When bridging a constraint, it selects the minimal number of bridges needed.

For example, if a constraint F-in-S can be bridged into a constraint F1-in-S1 (supported by the internal model) using bridge 1 or bridged into a constraint F2-in-S2 (unsupported by the internal model) using bridge 2 which can then be bridged into a constraint F3-in-S3 (supported by the internal model) using bridge 3, it will choose bridge 1 as it allows to bridge F-in-'S using only one bridge instead of two if it uses bridge 2 and 3.

MathOptInterface.Bridges.add_bridge – Function.

```
add_bridge(b::LazyBridgeOptimizer, BT::Type{<:AbstractBridge})
```

Enable the use of the bridges of type BT by b.

MathOptInterface.Bridges.remove_bridge – Function.

```
remove_bridge(b::LazyBridgeOptimizer, BT::Type{<:AbstractBridge})
```

Disable the use of the bridges of type BT by b.

MathOptInterface.Bridges.has_bridge – Function.

```
has_bridge(b::LazyBridgeOptimizer, BT::Type{<:AbstractBridge})
```

Return a Bool indicating whether the bridges of type BT are used by b.

MathOptInterface.Bridges.full_bridge_optimizer – Function.

```
full_bridge_optimizer(model::MOI.ModelLike, ::Type{T}) where {T}
```

Returns a LazyBridgeOptimizer bridging model for every bridge defined in this package (see below for the few exceptions) and for the coefficient type T in addition to the bridges in the list returned by MOI.get(model, MOI.Bridges.ListOfNonstandardBridges{T}()).

See also ListOfNonstandardBridges.

Note

The following bridges are not added by full_bridge_optimizer except if they are in the list returned by MOI.get(model, MOI.Bridges.ListOfNonstandardBridges{T}()) (see the doc-strings of the corresponding bridge for the reason they are not added):

- Constraint.SOCtoNonConvexQuadBridge, Constraint.RSOCtoNonConvexQuadBridge and Constraint.SOCtoPSDBridge.
- The subtypes of Constraint.AbstractToIntervalBridge (i.e. Constraint.GreaterToIntervalBridge and Constraint.LessToIntervalBridge) if T is not a subtype of AbstractFloat.

MathOptInterface.Bridges.ListOfNonstandardBridges – Type.

```
ListOfNonstandardBridges{T}() <: MOI.AbstractOptimizerAttribute
```

Any optimizer can be wrapped in a LazyBridgeOptimizer using full_bridge_optimizer. However, by default LazyBridgeOptimizer uses a limited set of bridges that are:

1. implemented in MOI.Bridges

2. generally applicable for all optimizers.

For some optimizers however, it is useful to add additional bridges, such as those that are implemented in external packages (e.g., within the solver package itself) or only apply in certain circumstances (e.g., `Constraint.SOCtoNonConvexQuadBridge`).

Such optimizers should implement the `ListOfNonstandardBridges` attribute to return a vector of bridge types that are added by `full_bridge_optimizer` in addition to the list of default bridges.

Note that optimizers implementing `ListOfNonstandardBridges` may require package-specific functions or sets to be used if the non-standard bridges are not added. Therefore, you are recommended to use `model = MOI.instantiate(Package.Optimizer; with_bridge_type = T)` instead of `model = MOI.instantiate(Package.Optimiz`
See `MathOptInterface.instantiate`.

Examples

An optimizer using a non-default bridge in `MOI.Bridges`

Solvers supporting `MOI.ScalarQuadraticFunction` can support `MOI.SecondOrderCone` and `MOI.RotatedSecondOrderCone` by defining:

```
function MOI.get(::MyQuadraticOptimizer, ::ListOfNonstandardBridges{Float64})
    return Type[
        MOI.Bridges.Constraint.SOCtoNonConvexQuadBridge{Float64},
        MOI.Bridges.Constraint.RSOCtoNonConvexQuadBridge{Float64},
    ]
end
```

An optimizer defining an internal bridge

Suppose an optimizer can exploit specific structure of a constraint, e.g., it can exploit the structure of the matrix `A` in the linear system of equations `A * x = b`.

The optimizer can define the function:

```
struct MatrixAffineFunction{T} <: MOI.AbstractVectorFunction
    A::SomeStructuredMatrixType{T}
    b::Vector{T}
end
```

and then a bridge

```
struct MatrixAffineFunctionBridge{T} <: MOI.Constraint.AbstractBridge
    # ...
end
# ...
```

from `VectorAffineFunction{T}` to the `MatrixAffineFunction`. Finally, it defines:

```
function MOI.get(::Optimizer{T}, ::ListOfNonstandardBridges{T}) where {T}
    return Type[MatrixAffineFunctionBridge{T}]
end
```

`MathOptInterface.Bridges.debug_supports_constraint` – Function.

```
debug_supports_constraint(
    b::LazyBridgeOptimizer,
    F::Type{<:MOI.AbstractFunction},
    S::Type{<:MOI.AbstractSet};
    io::IO = Base.stdout,
)
```

Prints to `io` explanations for the value of `MOI.supports_constraint` with the same arguments.

`MathOptInterface.Bridges.debug_supports` – Function.

```
debug_supports(
    b::LazyBridgeOptimizer,
    ::MOI.ObjectiveFunction{F};
    io::IO = Base.stdout,
) where F
```

Prints to `io` explanations for the value of `MOI.supports` with the same arguments.

`MathOptInterface.Bridges.bridged_variable_function` – Function.

```
bridged_variable_function(
    b::AbstractBridgeOptimizer,
    vi::MOI.VariableIndex,
)
```

Return a `MOI.AbstractScalarFunction` of variables of `b.model` that equals `vi`. That is, if the variable `vi` is bridged, it returns its expression in terms of the variables of `b.model`. Otherwise, it returns `vi`.

`MathOptInterface.Bridges.unbridged_variable_function` – Function.

```
unbridged_variable_function(
    b::AbstractBridgeOptimizer,
    vi::MOI.VariableIndex,
)
```

Return a `MOI.AbstractScalarFunction` of variables of `b` that equals `vi`. That is, if the variable `vi` is an internal variable of `b.model` created by a bridge but not visible to the user, it returns its expression in terms of the variables of bridged variables. Otherwise, it returns `vi`.

`MathOptInterface.Bridges.bridged_function` – Function.

```
bridged_function(b::AbstractBridgeOptimizer, value)::typeof(value)
```

Substitute any bridged `MOI.VariableIndex` in `value` by an equivalent expression in terms of variables of `b.model`.

`MathOptInterface.Bridges.Variable.unbridged_map` – Function.

unbridged_map(bridge::MOI.Bridges.Variable.AbstractBridge, vi::MOI.VariableIndex,)

For a bridged variable in a scalar set, return a tuple of pairs mapping the variables created by the bridge to an affine expression in terms of the bridged variable `vi`.

```
unbridged_map(
    bridge::MOI.Bridges.Variable.AbstractBridge,
    vis::Vector{MOI.VariableIndex},
)
```

For a bridged variable in a vector set, return a tuple of pairs mapping the variables created by the bridge to an affine expression in terms of the bridged variable `vis`. If this method is not implemented, it falls back to calling the following method for every variable of `vis`.

```
unbridged_map(
    bridge::MOI.Bridges.Variable.AbstractBridge,
    vi::MOI.VariableIndex,
    i::MOIB.IndexInVector,
)
```

For a bridged variable in a vector set, return a tuple of pairs mapping the variables created by the bridge to an affine expression in terms of the bridged variable `vi` corresponding to the `i`th variable of the vector.

If there is no way to recover the expression in terms of the bridged variable(s) `vi(s)`, return `nothing`. See `ZerosBridge` for an example of bridge returning `nothing`.

Constraint bridges `MathOptInterface.Bridges.Constraint.AbstractBridge` – Type.

```
AbstractBridge
```

Subtype of `MathOptInterface.Bridges.AbstractBridge` for constraint bridges.

`MathOptInterface.Bridges.Constraint.AbstractFunctionConversionBridge` – Type.

```
abstract type AbstractFunctionConversionBridge{F, S} <: AbstractBridge end
```

Bridge a constraint G-in-S into a constraint F-in-S where F and G are equivalent representations of the same function. By convention, the transformed function is stored in the `constraint` field.

`MathOptInterface.Bridges.Constraint.SingleBridgeOptimizer` – Type.

```
SingleBridgeOptimizer{BT<:AbstractBridge, OT<:MOI.ModelLike} <:
AbstractBridgeOptimizer
```

The `SingleBridgeOptimizer` bridges any constraint supported by the bridge `BT`. This is in contrast with the `MathOptInterface.Bridges.LazyBridgeOptimizer` which only bridges the constraints that are unsupported by the internal model, even if they are supported by one of its bridges.

`MathOptInterface.Bridges.Constraint.add_all_bridges` – Function.

```
add_all_bridges(bridged_model, ::Type{T}) where {T}
```

Add all bridges defined in the `Bridges.Constraint` submodule to `bridged_model`. The coefficient type used is `T`.

SetMap bridges `MathOptInterface.Bridges.Variable.SetMapBridge` – Type.

```
abstract type SetMapBridge{T,S1,S2} <: AbstractBridge end
```

Consider two type of sets `S1`, `S2` and a linear mapping `A` that the image of a set of type `S1` under `A` is a set of type `S2`. A `SetMapBridge{T,S1,S2}` is a bridge that substitutes constrained variables in `S2` into the image through `A` of constrained variables in `S1`.

The linear map `A` is described by `MathOptInterface.Bridges.map_set`, `MathOptInterface.Bridges.map_function`. Implementing a method for these two functions is sufficient to bridge constrained variables. In order for the getters and setters of dual solutions, starting values, etc... to work as well a method for the following functions should be implemented as well: `MathOptInterface.Bridges.inverse_map_set`, `MathOptInterface.Bridges.inverse`

MathOptInterface.Bridges.adjoint_map_function and MathOptInterface.Bridges.inverse_adjoint_map_function. See the docstrings of the function to see which feature would be missing it it was not implemented for a given bridge.

MathOptInterface.Bridges.Constraint.SetMapBridge – Type.

```
abstract type SetMapBridge{T,S2,S1,F,G} <: AbstractBridge end
```

Consider two type of sets S1, S2 and a linear mapping A that the image of a set of type S1 under A is a set of type S2. A SetMapBridge{T,S2,S1,F,G} is a bridge that maps G-in-S2 constraints into F-in-S1 by mapping the function through A.

The linear map A is described by MathOptInterface.Bridges.map_set, MathOptInterface.Bridges.map_function. Implementing a method for these two functions is sufficient to bridge constraints. In order for the getters and setters of dual solutions, starting values, etc... to work as well a method for the following functions should be implemented as well: MathOptInterface.Bridges.inverse_map_set, MathOptInterface.Bridges.inverse_map MathOptInterface.Bridges.adjoint_map_function and MathOptInterface.Bridges.inverse_adjoint_map_function. See the docstrings of the function to see which feature would be missing it it was not implemented for a given bridge.

MathOptInterface.Bridges.map_set – Function.

```
map_set(::Type{BT}, set) where {BT}
```

Return the image of set through the linear map A defined in Variable.SetMapBridge and Constraint.SetMapBridge. This is used for bridging the constraint and setting the MathOptInterface.ConstraintSet.

MathOptInterface.Bridges.inverse_map_set – Function.

```
inverse_map_set(::Type{BT}, set) where {BT}
```

Return the preimage of set through the linear map A defined in Variable.SetMapBridge and Constraint.SetMapBridge. This is used for getting the MathOptInterface.ConstraintSet.

MathOptInterface.Bridges.map_function – Function.

```
map_function(::Type{BT}, func) where {BT}
```

Return the image of func through the linear map A defined in Variable.SetMapBridge and Constraint.SetMapBridge. This is used for getting the MathOptInterface.ConstraintPrimal of variable bridges. For constraint bridges, this is used for bridging the constraint, setting the MathOptInterface.ConstraintFunction and MathOptInterface.ConstraintPrimalStart and modifying the function with MathOptInterface.modify.

```
map_function(::Type{BT}, func, i::IndexInVector) where {BT}
```

Return the scalar function at the ith index of the vector function that would be returned by map_function(BT, func) except that it may compute the ith element. This is used by bridged_function and for getting the MathOptInterface.VariablePrimal and MathOptInterface.VariablePrimalStart of variable bridges.

MathOptInterface.Bridges.inverse_map_function – Function.

```
inverse_map_function(::Type{BT}, func) where {BT}
```

Return the image of func through the inverse of the linear map A defined in Variable.SetMapBridge and Constraint.SetMapBridge. This is used by Variable.unbridged_map and for setting the MathOptInterface.VariablePrima of variable bridges and for getting the MathOptInterface.ConstraintFunction, the MathOptInterface.ConstraintPrimal and the MathOptInterface.ConstraintPrimalStart of constraint bridges.

MathOptInterface.Bridges.adjoint_map_function – Function.

```
adjoint_map_function(::Type{BT}, func) where {BT}
```

Return the image of func through the adjoint of the linear map A defined in Variable.SetMapBridge and Constraint.SetMapBridge. This is used for getting the MathOptInterface.ConstraintDual and MathOptInterface.ConstraintDualStart of constraint bridges.

MathOptInterface.Bridges.inverse_adjoint_map_function – Function.

```
inverse_adjoint_map_function(::Type{BT}, func) where {BT}
```

Return the image of func through the inverse of the adjoint of the linear map A defined in Variable.SetMapBridge and Constraint.SetMapBridge. This is used for getting the MathOptInterface.ConstraintDual of variable bridges and setting the MathOptInterface.ConstraintDualStart of constraint bridges.

Bridges implemented MathOptInterface.Bridges.Constraint.FlipSignBridge – Type.

```
FlipSignBridge{T, S1, S2, F, G}
```

Bridge a G-in-S1 constraint into an F-in-S2 constraint by multiplying the function by -1 and taking the point reflection of the set across the origin. The flipped F-in-S constraint is stored in the constraint field by convention.

MathOptInterface.Bridges.Constraint.AbstractToIntervalBridge – Type.

```
AbstractToIntervalBridge{T, S1, F}
```

Bridge a F-in-Interval constraint into an F-in-Interval{T} constraint where we have either:

- S1 = MOI.GreaterThan{T}
- S1 = MOI.LessThan{T}

The F-in-Interval{T} constraint is stored in the constraint field by convention.

Warning

It is required that T be a AbstractFloat type because otherwise typemin and typemax would either be not implemented (e.g. BigInt) or would not give infinite value (e.g. Int). For this reason, this bridge is only added to MathOptInterface.Bridges.full_bridge_optimizer. when T is a subtype of AbstractFloat.

MathOptInterface.Bridges.Constraint.GreaterToIntervalBridge – Type.

```
GreaterToIntervalBridge{T, F<:MOI.AbstractScalarFunction} <:
    AbstractToIntervalBridge{T, MOI.GreaterThan{T}, F}
```

Transforms a F-in-GreaterThan{T} constraint into an F-in-Interval{T} constraint.

MathOptInterface.Bridges.Constraint.LessToIntervalBridge – Type.

```
LessToIntervalBridge{T, F<:MOI.AbstractScalarFunction} <:
    AbstractToIntervalBridge{T, MOI.LessThan{T}, F}
```

Transforms a F-in-LessThan{T} constraint into an F-in-Interval{T} constraint.

MathOptInterface.Bridges.Constraint.GreaterToLessBridge – Type.

```
GreaterToLessBridge{
    T,
    F<:MOI.AbstractScalarFunction,
    G<:MOI.AbstractScalarFunction
} <: FlipSignBridge{T, MOI.GreaterThan{T}, MOI.LessThan{T}, F, G}
```

Transforms a G-in-GreaterThan{T} constraint into an F-in-LessThan{T} constraint.

MathOptInterface.Bridges.Constraint.LessToGreaterBridge – Type.

```
LessToGreaterBridge{
    T,
    F<:MOI.AbstractScalarFunction,
    G<:MOI.AbstractScalarFunction
} <: FlipSignBridge{T, MOI.LessThan{T}, MOI.GreaterThan{T}, F, G}
```

Transforms a G-in-LessThan{T} constraint into an F-in-GreaterThan{T} constraint.

MathOptInterface.Bridges.Constraint.NonnegToNonposBridge – Type.

```
NonnegToNonposBridge{
    T,
    F<:MOI.AbstractVectorFunction,
    G<:MOI.AbstractVectorFunction
} <: FlipSignBridge{T, MOI.Nonnegatives, MOI.Nonpositives, F, G}
```

Transforms a G-in-Nonnegatives constraint into a F-in-Nonpositives constraint.

MathOptInterface.Bridges.Constraint.NonposToNonnegBridge – Type.

```
NonposToNonnegBridge{
    T,
    F<:MOI.AbstractVectorFunction,
    G<:MOI.AbstractVectorFunction,
} <: FlipSignBridge{T, MOI.Nonpositives, MOI.Nonnegatives, F, G}
```

Transforms a G-in-Nonpositives constraint into a F-in-Nonnegatives constraint.

MathOptInterface.Bridges.Constraint.VectorizeBridge – Type.

```
VectorizeBridge{T,F,S,G}
```

Transforms a constraint G-in-scalar_set_type(S, T) where S <: VectorLinearSet to F-in-S.

Examples

The constraint VariableIndex-in-LessThan{Float64} becomes VectorAffineFunction{Float64}-in-Nonpositives, where T = Float64, F = VectorAffineFunction{Float64}, S = Nonpositives, and G = VariableIndex.

MathOptInterface.Bridges.Constraint.ScalarizeBridge – Type.

```
ScalarizeBridge{T, F, S}
```

Transforms a constraint AbstractVectorFunction-in-vector_set_type(S) where S <: LPCone{T} to F-in-S.

MathOptInterface.Bridges.Constraint.ScalarSlackBridge – Type.

```
ScalarSlackBridge{T, F, S}
```

The `ScalarSlackBridge` converts a constraint `G`-in-`S` where `G` is a function different from `VariableIndex` into the constraints `F`-in-`EqualTo{T}` and `VariableIndex`-in-`S`.

`F` is the result of subtracting a `VariableIndex` from `G`. Typically `G` is the same as `F`, but that is not mandatory.

`MathOptInterface.Bridges.Constraint.VectorSlackBridge` – Type.

```
VectorSlackBridge{T, F, S}
```

The `VectorSlackBridge` converts a constraint `G`-in-`S` where `G` is a function different from `VectorOfVariables` into the constraints `F`-in-`Zeros` and `VectorOfVariables`-in-`S`.

`F` is the result of subtracting a `VectorOfVariables` from `G`. Typically `G` is the same as `F`, but that is not mandatory.

`MathOptInterface.Bridges.Constraint.ScalarFunctionizeBridge` – Type.

```
ScalarFunctionizeBridge{T, S}
```

The `ScalarFunctionizeBridge` converts a constraint `VariableIndex`-in-`S` into the constraint `ScalarAffineFunction{T}`-in-`S`.

`MathOptInterface.Bridges.Constraint.VectorFunctionizeBridge` – Type.

```
VectorFunctionizeBridge{T, S}
```

The `VectorFunctionizeBridge` converts a constraint `VectorOfVariables`-in-`S` into the constraint `VectorAffineFunction{T}`-in-`S`.

`MathOptInterface.Bridges.Constraint.SplitIntervalBridge` – Type.

```
SplitIntervalBridge{T, F, S, LS, US}
```

The `SplitIntervalBridge` splits a `F`-in-`S` constraint into a `F`-in-`LS` and a `F`-in-`US` constraint where we have either:

- `S = MOI.Interval{T}`, `LS = MOI.GreaterThan{T}` and `US = MOI.LessThan{T}`,
- `S = MOI.EqualTo{T}`, `LS = MOI.GreaterThan{T}` and `US = MOI.LessThan{T}`, or
- `S = MOI.Zeros`, `LS = MOI.Nonnegatives` and `US = MOI.Nonpositives`.

For instance, if `F` is `MOI.ScalarAffineFunction` and `S` is `MOI.Interval`, it transforms the constraint $la, x + u$ into the constraints $a, x + l$ and $a, x + u$.

Note

If `T<:AbstractFloat` and `S` is `MOI.Interval{T}` then no lower (resp. upper) bound constraint is created if the lower (resp. upper) bound is `typemin(T)` (resp. `typemax(T)`). Similarly, when `MathOptInterface.ConstraintSet` is set, a lower or upper bound constraint may be deleted or created accordingly.

`MathOptInterface.Bridges.Constraint.SOCtoRSOCBridge` – Type.

```
SOCtoRSOCBridge{T, F, G}
```

We simply do the inverse transformation of RSOCtoSOCBridge. In fact, as the transformation is an involution, we do the same transformation.

MathOptInterface.Bridges.Constraint.RSOCtoSOCBridge – Type.

```
RSOCtoSOCBridge{T, F, G}
```

The RotatedSecondOrderCone is SecondOrderCone representable; see [BN01, p. 104]. Indeed, we have $2tu = (t/\sqrt{2} + u/\sqrt{2})^2 - (t/\sqrt{2} - u/\sqrt{2})^2$ hence

$$2tu \geq \|x\|_2^2$$

is equivalent to

$$(t/\sqrt{2} + u/\sqrt{2})^2 \geq \|x\|_2^2 + (t/\sqrt{2} - u/\sqrt{2})^2.$$

We can therefore use the transformation $(t, u, x) \mapsto (t/\sqrt{2} + u/\sqrt{2}, t/\sqrt{2} - u/\sqrt{2}, x)$. Note that the linear transformation is a symmetric involution (i.e. it is its own transpose and its own inverse). That means in particular that the norm of constraint primal and dual values are preserved by the tranformation.

[BN01] Ben-Tal, Aharon, and Nemirovski, Arkadi. Lectures on modern convex optimization: analysis, algorithms, and engineering applications. Society for Industrial and Applied Mathematics, 2001.

MathOptInterface.Bridges.Constraint.SOCtoNonConvexQuadBridge – Type.

```
SOCtoNonConvexQuadBridge{T}
```

Constraints of the form VectorOfVariables-in-SecondOrderCone can be transformed into a ScalarQuadraticFunction-in-LessThan and a ScalarAffineFunction-in-GreaterThan. Indeed, the definition of the second-order cone

$$t \geq \|x\|_2 \ (1)$$

is equivalent to

$$\sum x_i^2 \leq t^2 (2)$$

with $t \geq 0$. (3)

Warning

This transformation starts from a convex constraint (1) and creates a non-convex constraint (2), because the Q matrix associated with the constraint (2) has one negative eigenvalue. This might be wrongly interpreted by a solver. Some solvers can look at (2) and understand that it is a second order cone, but this is not a general rule. For these reasons this bridge is not automatically added by MOI.Bridges.full_bridge_optimizer. Care is recommended when adding this bridge to a optimizer.

MathOptInterface.Bridges.Constraint.RSOCtoNonConvexQuadBridge – Type.

```
RSOCtoNonConvexQuadBridge{T}
```

Constraints of the form `VectorOfVariables`-in-`SecondOrderCone` can be transformed into a `ScalarQuadraticFunction`-in-`LessThan` and a `ScalarAffineFunction`-in-`GreaterThan`. Indeed, the definition of the second-order cone

$$2tu \geq \|x\|_2^2, t, u \geq 0 (1)$$

is equivalent to

$$\sum x_i^2 \leq 2tu (2)$$

with $t, u \geq 0$. (3)

WARNING This transformation starts from a convex constraint (1) and creates a non-convex constraint (2), because the `Q` matrix associated with the constraint 2 has two negative eigenvalues. This might be wrongly interpreted by a solver. Some solvers can look at (2) and understand that it is a rotated second order cone, but this is not a general rule. For these reasons, this bridge is not automatically added by `MOI.Bridges.full_bridge_optimizer`. Care is recommended when adding this bridge to an optimizer.

`MathOptInterface.Bridges.Constraint.QuadtoSOCBridge` – Type.

```
QuadtoSOCBridge{T}
```

The set of points `x` satisfying the constraint

$$\frac{1}{2}x^T Q x + a^T x + b \leq 0$$

is a convex set if `Q` is positive semidefinite and is the union of two convex cones if `a` and `b` are zero (i.e. homogeneous case) and `Q` has only one negative eigenvalue. Currently, only the non-homogeneous transformation is implemented, see the Note section below for more details.

Non-homogeneous case

If `Q` is positive semidefinite, there exists `U` such that $Q = U^T U$, the inequality can then be rewritten as

$$\|Ux\|_2^2 \leq 2(-a^T x - b)$$

which is equivalent to the membership of `(1, -a^T x - b, Ux)` to the rotated second-order cone.

Homogeneous case

If `Q` has only one negative eigenvalue, the set of `x` such that $x^T Q x \leq 0$ is the union of a convex cone and its opposite. We can choose which one to model by checking the existence of bounds on variables as shown below.

Second-order cone

If `Q` is diagonal and has eigenvalues `(1, 1, -1)`, the inequality $x^2 + x^2 \leq z^2$ combined with $z \geq 0$ defines the Lorenz cone (i.e. the second-order cone) but when combined with $z \leq 0$, it gives the opposite of the

second order cone. Therefore, we need to check if the variable `z` has a lower bound 0 or an upper bound 0 in order to determine which cone is

Rotated second-order cone

The matrix `Q` corresponding to the inequality $x^2 \leq 2yz$ has one eigenvalue `1` with eigenvectors `(1, 0, 0)` and `(0, 1, -1)` and one eigenvalue `-1` corresponding to the eigenvector `(0, 1, 1)`. Hence if we intersect this union of two convex cone with the halfspace $x + y \geq 0$, we get the rotated second-order cone and if we intersect it with the halfspace $x + y \leq 0$ we get the opposite of the rotated second-order cone. Note that `y` and `z` have the same sign since `yz` is nonnegative hence $x + y \geq 0$ is equivalent to $x \geq 0$ and $y \geq 0$.

Note

The check for existence of bound can be implemented (but inefficiently) with the current interface but if bound is removed or transformed (e.g. `≤ 0` transformed into `≥ 0`) then the bridge is no longer valid. For this reason the homogeneous version of the bridge is not implemented yet.

`MathOptInterface.Bridges.Constraint.SOCtoPSDBridge` – Type.

The `SOCtoPSDBridge` transforms the second order cone constraint $\|x\| \leq t$ into the semidefinite cone constraints

$$\begin{pmatrix} t & x^\top \\ x & tI \end{pmatrix} \succeq 0$$

Indeed by the Schur Complement, it is positive definite iff

$$\begin{aligned} tI &\succ 0 \\ t - x^\top (tI)^{-1} x &\succ 0 \end{aligned}$$

which is equivalent to

$$\begin{aligned} t &> 0 \\ t^2 &> x^\top x \end{aligned}$$

Warning

This bridge is not added by default by `MOI.Bridges.full_bridge_optimizer` as bridging second order cone constraints to semidefinite constraints can be achieved by the `SOCtoRSOCBridge` followed by the `RSOCtoPSDBridge` while creating a smaller semidefinite constraint.

`MathOptInterface.Bridges.Constraint.RSOCtoPSDBridge` – Type.

The `RSOCtoPSDBridge` transforms the second order cone constraint $\|x\| \leq 2tu$ with $u \geq 0$ into the semidefinite cone constraints

$$\begin{pmatrix} t & x^\top \\ x & 2uI \end{pmatrix} \succeq 0$$

Indeed by the Schur Complement, it is positive definite iff

$$uI \succ 0$$
$$t - x^\top (2uI)^{-1} x \succ 0$$

which is equivalent to

$$u > 0$$
$$2tu > x^\top x$$

MathOptInterface.Bridges.Constraint.NormInfinityBridge – Type.

```
NormInfinityBridge{T}
```

The NormInfinityCone is representable with LP constraints, since $t \geq \max_i |x_i|$ if and only if $t \geq x_i$ and $t \geq -x_i$ for all i.

MathOptInterface.Bridges.Constraint.NormOneBridge – Type.

```
NormOneBridge{T}
```

The NormOneCone is representable with LP constraints, since $t \geq \sum_i |x_i|$ if and only if there exists a vector y such that $t \geq \sum_i y_i$ and $y_i \geq x_i$, $y_i \geq -x_i$ for all i.

MathOptInterface.Bridges.Constraint.GeoMeantoRelEntrBridge – Type.

```
GeoMeantoRelEntrBridge{T}
```

The `geometric mean` cone is representable with a relative entropy constraint and a nonnegative auxiliary variable.

This is because $u \leq \prod_{i=1}^n w_i^{1/n}$ is equivalent to $y \geq 0$ and $0 \leq u + y \leq \prod_{i=1}^n w_i^{1/n}$, and the latter inequality is equivalent to $1 \leq \prod_{i=1}^n (\frac{w_i}{u+y})^{1/n}$, which is equivalent to $0 \leq \sum_{i=1}^n \log(\frac{w_i}{u+y})^{1/n}$, which is equivalent to $0 \geq \sum_{i=1}^n (u+y) \log(\frac{u+y}{w_i})$.

Thus $(u, w) \in GeometricMeanCone(1+n)$ is representable as $y \geq 0$, $(0, w, (u+y)e) \in RelativeEntropyCone(1+2n)$, where e is a vector of ones.

MathOptInterface.Bridges.Constraint.GeoMeanBridge – Type.

```
GeoMeanBridge{T, F, G, H}
```

The GeometricMeanCone is SecondOrderCone representable; see [1, p. 105].

The reformulation is best described in an example.

Consider the cone of dimension 4:

$$t \leq \sqrt[3]{x_1 x_2 x_3}$$

This can be rewritten as $\exists x_{21} \geq 0$ such that:

$$t \leq x_{21},$$
$$x_{21}^4 \leq x_1 x_2 x_3 x_{21}.$$

Note that we need to create x_{21} and not use t^4 directly as t is allowed to be negative. Now, this is equivalent to:

$$\begin{aligned} t &\leq x_{21}/\sqrt{4}, \\ x_{21}^2 &\leq 2x_{11}x_{12}, \\ x_{11}^2 &\leq 2x_1 x_2, \qquad\qquad x_{12}^2 \leq 2x_3(x_{21}/\sqrt{4}). \end{aligned}$$

[1] Ben-Tal, Aharon, and Arkadi Nemirovski. Lectures on modern convex optimization: analysis, algorithms, and engineering applications. Society for Industrial and Applied Mathematics, 2001.

`MathOptInterface.Bridges.Constraint.RelativeEntropyBridge` – Type.

```
RelativeEntropyBridge{T}
```

The `RelativeEntropyCone` is representable with exponential cone and LP constraints, since $u \geq \sum_{i=1}^n w_i \log(\frac{w_i}{v_i})$ if and only if there exists a vector y such that $u \geq \sum_i y_i$ and $y_i \geq w_i \log(\frac{w_i}{v_i})$ or equivalently $v_i \geq w_i \exp(\frac{-y_i}{w_i})$ or equivalently $(-y_i, w_i, v_i) \in ExponentialCone$, for all i.

`MathOptInterface.Bridges.Constraint.NormSpectralBridge` – Type.

```
NormSpectralBridge{T}
```

The `NormSpectralCone` is representable with a PSD constraint, since $t \geq \sigma_1(X)$ if and only if $[tI X^\top; XtI] \succ 0$.

`MathOptInterface.Bridges.Constraint.NormNuclearBridge` – Type.

```
NormNuclearBridge{T}
```

The `NormNuclearCone` is representable with an SDP constraint and extra variables, since $t \geq \sum_i \sigma_i(X)$ if and only if there exists symmetric matrices U, V such that $[UX^\top; XV] \succ 0$ and $t \geq (tr(U) + tr(V))/2$.

`MathOptInterface.Bridges.Constraint.SquareBridge` – Type.

```
SquareBridge{T, F<:MOI.AbstractVectorFunction,
             G<:MOI.AbstractScalarFunction,
             TT<:MOI.AbstractSymmetricMatrixSetTriangle,
             ST<:MOI.AbstractSymmetricMatrixSetSquare} <: AbstractBridge
```

The `SquareBridge` reformulates the constraint of a square matrix to be in `ST` to a list of equality constraints for pair or off-diagonal entries with different expressions and a `TT` constraint the upper triangular part of the matrix.

For instance, the constraint for the matrix

$$\begin{pmatrix} 1 & 1+x & 2-3x \\ 1+x & 2+x & 3-x \\ 2-3x & 2+x & 2x \end{pmatrix}$$

to be PSD can be broken down to the constraint of the symmetric matrix

$$\begin{pmatrix} 1 & 1+x & 2-3x \\ \cdot & 2+x & 3-x \\ \cdot & \cdot & 2x \end{pmatrix}$$

and the equality constraint between the off-diagonal entries (2, 3) and (3, 2) $2x == 1$. Note that now symmetrization constraint need to be added between the off-diagonal entries (1, 2) and (2, 1) or between (1, 3) and (3, 1) since the expressions are the same.

`MathOptInterface.Bridges.Constraint.RootDetBridge` – Type.

```
RootDetBridge{T,F,G,H}
```

The `RootDetConeTriangle` is representable by a `PositiveSemidefiniteConeTriangle` and an `GeometricMeanCone` constraints; see [1, p. 149].

Indeed, $t \leq \det(X)^{1/n}$ if and only if there exists a lower triangular matrix such that:

$$\begin{pmatrix} X & \\ \top & \text{Diag}() \end{pmatrix} \succeq 0$$
$$t \leq (_{11}{}_{22} \cdots {}_{nn})^{1/n}$$

[1] Ben-Tal, Aharon, and Arkadi Nemirovski. Lectures on modern convex optimization: analysis, algorithms, and engineering applications. Society for Industrial and Applied Mathematics, 2001.

`MathOptInterface.Bridges.Constraint.LogDetBridge` – Type.

```
LogDetBridge{T,F,G,H,I}
```

The `LogDetConeTriangle` is representable by a `PositiveSemidefiniteConeTriangle` and `ExponentialCone` constraints.

Indeed, $\log\det(X) = \log(\delta_1) + \cdots + \log(\delta_n)$ where δ_1, ..., δ_n are the eigenvalues of X.

Adapting the method from [1, p. 149], we see that $t \leq u\log(\det(X/u))$ for $u > 0$ if and only if there exists a lower triangular matrix such that

$$\begin{pmatrix} X & \\ \top & \text{Diag}() \end{pmatrix} \succeq 0$$
$$t \leq u\log(_{11}/u) + u\log(_{22}/u) + \cdots + u\log(_{nn}/u)$$

[1] Ben-Tal, Aharon, and Arkadi Nemirovski. Lectures on modern convex optimization: analysis, algorithms, and engineering applications. Society for Industrial and Applied Mathematics, 2001. "'

`MathOptInterface.Bridges.Constraint.IndicatorActiveOnFalseBridge` – Type.

```
IndicatorActiveOnFalseBridge{T}
```

The `IndicatorActiveOnFalseBridge` replaces an indicator constraint activated on 0 with a variable z_0 with the constraint activated on 1, with a variable z_1. It stores the added `variable` and added constraints:

- $z_1 \in \mathbb{B}$ in `zero_one_cons`

- $z_0 + z_1 == 1$ in 'indisjunction_cons'
- The added ACTIVATE_ON_ONE indicator constraint in indicator_cons_index.

MathOptInterface.Bridges.Constraint.IndicatorSOS1Bridge – Type.

```
IndicatorSOS1Bridge{T,S<:MOI.AbstractScalarSet}
```

The IndicatorSOS1Bridge replaces an indicator constraint of the following form: $z \in \mathbb{B}, z == 1 \implies f(x) \in S$ with a SOS1 constraint: $z \in \mathbb{B}, slack$ free, $f(x) + slack \in S, SOS1(slack, z)$.

MathOptInterface.Bridges.Constraint.SemiToBinaryBridge – Type.

```
SemiToBinaryBridge{T, S <: MOI.AbstractScalarSet}
```

The SemiToBinaryBridge replaces a Semicontinuous constraint: $x \in \text{Semicontinuous}(l, u)$ is replaced by: $z \in \{0, 1\}, x \leq z \cdot u, x \geq z \cdot l$.

The SemiToBinaryBridge replaces a Semiinteger constraint: $x \in Semiinteger(l, u)$ is replaced by: $z \in \{0, 1\}, x \in \mathbb{Z}, x \leq z \cdot u, x \geq z \cdot l$.

MathOptInterface.Bridges.Constraint.ZeroOneBridge – Type.

```
ZeroOneBridge{T}
```

The ZeroOneBridge splits a MOI.VariableIndex-in-MOI.ZeroOne constraint into a MOI.VariableIndex-in-MOI.Integer constraint and a MOI.VariableIndex-in-MOI.Interval(0, 1) constraint.

Variable bridges MathOptInterface.Bridges.Variable.AbstractBridge – Type.

```
AbstractBridge
```

Subtype of MathOptInterface.Bridges.AbstractBridge for variable bridges.

MathOptInterface.Bridges.Variable.SingleBridgeOptimizer – Type.

```
SingleBridgeOptimizer{BT<:AbstractBridge, OT<:MOI.ModelLike} <:
AbstractBridgeOptimizer
```

The SingleBridgeOptimizer bridges any constrained variables supported by the bridge BT. This is in contrast with the MathOptInterface.Bridges.LazyBridgeOptimizer which only bridges the constrained variables that are unsupported by the internal model, even if they are supported by one of its bridges.

> **Note**
> Two bridge optimizers using variable bridges cannot be used together as both of them assume that the underlying model only returns variable indices with nonnegative values.

MathOptInterface.Bridges.Variable.add_all_bridges – Function.

```
add_all_bridges(bridged_model, ::Type{T}) where {T}
```

Add all bridges defined in the Bridges.Variable submodule to bridged_model. The coefficient type used is T.

Bridges implemented `MathOptInterface.Bridges.Variable.FlipSignBridge` – Type.

```
FlipSignBridge{T, S1, S2}
```

Bridge constrained variables in `S1` into constrained variables in `S2` by multiplying the variables by `-1` and taking the point reflection of the set across the origin. The flipped `MOI.VectorOfVariables`-in-`S` constraint is stored in the `flipped_constraint` field by convention.

`MathOptInterface.Bridges.Variable.ZerosBridge` – Type.

```
ZerosBridge{T} <: Bridges.Variable.AbstractBridge
```

Transforms constrained variables in `MathOptInterface.Zeros` to zeros, which ends up creating no variables in the underlying model.

The bridged variables are therefore similar to parameters with zero values. Parameters with non-zero value can be created with constrained variables in `MOI.EqualTo` by combining a `VectorizeBridge` and this bridge. The functions cannot be unbridged, given a function, we cannot determine, if the bridged variables were used.

The dual values cannot be determined by the bridge but they can be determined by the bridged optimizer using `MathOptInterface.Utilities.get_fallback` if a `CachingOptimizer` is used (since `ConstraintFunction` cannot be got as functions cannot be unbridged).

`MathOptInterface.Bridges.Variable.FreeBridge` – Type.

```
FreeBridge{T} <: Bridges.Variable.AbstractBridge
```

Transforms constrained variables in `MOI.Reals` to the difference of constrained variables in `MOI.Nonnegatives`.

`MathOptInterface.Bridges.Variable.NonposToNonnegBridge` – Type.

```
NonposToNonnegBridge{T} <:
    FlipSignBridge{T, MOI.Nonpositives, MOI.Nonnegatives}
```

Transforms constrained variables in `Nonpositives` into constrained variables in `Nonnegatives`.

`MathOptInterface.Bridges.Variable.VectorizeBridge` – Type.

```
VectorizeBridge{T, S}
```

Transforms a constrained variable in `scalar_set_type(S, T)` where `S <: VectorLinearSet` into a constrained vector of one variable in `S`. For instance, `VectorizeBridge{Float64, MOI.Nonnegatives}` transforms a constrained variable in `MOI.GreaterThan{Float64}` into a constrained vector of one variable in `MOI.Nonnegatives`.

`MathOptInterface.Bridges.Variable.SOCtoRSOCBridge` – Type.

```
SOCtoRSOCBridge{T} <:
↪  Bridges.Variable.SetMapBridge{T,MOI.RotatedSecondOrderCone,MOI.SecondOrderCone}
```

Same transformation as `MOI.Bridges.Constraint.SOCtoRSOCBridge`.

`MathOptInterface.Bridges.Variable.RSOCtoSOCBridge` – Type.

```
RSOCtoSOCBridge{T} <:
↪  Bridges.Variable.SetMapBridge{T,MOI.SecondOrderCone,MOI.RotatedSecondOrderCone}
```

Same transformation as MOI.Bridges.Constraint.RSOCtoSOCBridge.

MathOptInterface.Bridges.Variable.RSOCtoPSDBridge – Type.

```
RSOCtoPSDBridge{T} <: Bridges.Variable.AbstractBridge
```

Transforms constrained variables in MathOptInterface.RotatedSecondOrderCone to constrained variables in MathOptInterface.PositiveSemidefiniteConeTriangle.

Objective bridges MathOptInterface.Bridges.Objective.AbstractBridge – Type.

```
AbstractBridge
```

Subtype of MathOptInterface.Bridges.AbstractBridge for objective bridges.

MathOptInterface.Bridges.Objective.SingleBridgeOptimizer – Type.

```
SingleBridgeOptimizer{BT<:AbstractBridge, OT<:MOI.ModelLike} <: AbstractBridgeOptimizer
```

The SingleBridgeOptimizer bridges any objective functions supported by the bridge BT. This is in contrast with the MathOptInterface.Bridges.LazyBridgeOptimizer which only bridges the objective functions that are unsupported by the internal model, even if they are supported by one of its bridges.

MathOptInterface.Bridges.Objective.add_all_bridges – Function.

```
add_all_bridges(bridged_model, ::Type{T}) where {T}
```

Add all bridges defined in the Bridges.Objective submodule to bridged_model. The coefficient type used is T.

Bridges implemented MathOptInterface.Bridges.Objective.SlackBridge – Type.

```
SlackBridge{T, F, G}
```

The SlackBridge converts an objective function of type G into a MOI.VariableIndex objective by creating a slack variable and a F-in-MOI.LessThan constraint for minimization or F-in-MOI.LessThan constraint for maximization where F is MOI.Utilities.promote_operation(-, T, G, MOI.VariableIndex}. Note that when using this bridge, changing the optimization sense is not supported. Set the sense to MOI.FEASIBILITY_SENSE first to delete the bridge in order to change the sense, then re-add the objective.

MathOptInterface.Bridges.Objective.FunctionizeBridge – Type.

```
FunctionizeBridge{T}
```

The FunctionizeBridge converts a VariableIndex objective into a ScalarAffineFunction{T} objective.

40.3 FileFormats

Overview

The FileFormats submodule

The FileFormats module provides functionality for reading and writing MOI models using write_to_file and read_from_file.

Supported file types You must read and write files to a `FileFormats.Model` object. Specifc the file-type by passing a `FileFormats.FileFormat` enum. For example:

The Conic Benchmark Format

```
julia> model = MOI.FileFormats.Model(format = MOI.FileFormats.FORMAT_CBF)
A Conic Benchmark Format (CBF) model
```

The LP file format

```
julia> model = MOI.FileFormats.Model(format = MOI.FileFormats.FORMAT_LP)
A .LP-file model
```

The MathOptFormat file format

```
julia> model = MOI.FileFormats.Model(format = MOI.FileFormats.FORMAT_MOF)
A MathOptFormat Model
```

The MPS file format

```
julia> model = MOI.FileFormats.Model(format = MOI.FileFormats.FORMAT_MPS)
A Mathematical Programming System (MPS) model
```

The NL file format

```
julia> model = MOI.FileFormats.Model(format = MOI.FileFormats.FORMAT_NL)
An AMPL (.nl) model
```

The SDPA file format

```
julia> model = MOI.FileFormats.Model(format = MOI.FileFormats.FORMAT_SDPA)
A SemiDefinite Programming Algorithm Format (SDPA) model
```

Write to file To write a model `src` to a MathOptFormat file, use:

```
julia> src = MOI.Utilities.Model{Float64}()
MOIU.Model{Float64}

julia> MOI.add_variable(src)
MathOptInterface.VariableIndex(1)

julia> dest = MOI.FileFormats.Model(format = MOI.FileFormats.FORMAT_MOF)
A MathOptFormat Model

julia> MOI.copy_to(dest, src)
MathOptInterface.Utilities.IndexMap with 1 entry:
  VariableIndex(1) => VariableIndex(1)

julia> MOI.write_to_file(dest, "file.mof.json")

julia> print(read("file.mof.json", String))
```

```
{
  "name": "MathOptFormat Model",
  "version": {
    "major": 1,
    "minor": 0
  },
  "variables": [
    {
      "name": "x1"
    }
  ],
  "objective": {
    "sense": "feasibility"
  },
  "constraints": []
}
```

Read from file To read a MathOptFormat file, use:

```
julia> dest = MOI.FileFormats.Model(format = MOI.FileFormats.FORMAT_MOF)
A MathOptFormat Model

julia> MOI.read_from_file(dest, "file.mof.json")

julia> MOI.get(dest, MOI.ListOfVariableIndices())
1-element Vector{MathOptInterface.VariableIndex}:
 MathOptInterface.VariableIndex(1)

julia> rm("file.mof.json")  # Clean up after ourselves.
```

Detecing the filetype automatically Instead of the `format` keyword, you can also use the `filename` keyword argument to `FileFormats.Model`. This will attempt to automatically guess the format from the file extension. For example:

```
julia> src = MOI.Utilities.Model{Float64}()
MOIU.Model{Float64}

julia> dest = MOI.FileFormats.Model(filename = "file.cbf.gz")
A Conic Benchmark Format (CBF) model

julia> MOI.copy_to(dest, src)
MathOptInterface.Utilities.IndexMap()

julia> MOI.write_to_file(dest, "file.cbf.gz")

julia> src_2 = MOI.FileFormats.Model(filename = "file.cbf.gz")
A Conic Benchmark Format (CBF) model

julia> src = MOI.Utilities.Model{Float64}()
MOIU.Model{Float64}

julia> dest = MOI.FileFormats.Model(filename = "file.cbf.gz")
A Conic Benchmark Format (CBF) model

julia> MOI.copy_to(dest, src)
```

```
MathOptInterface.Utilities.IndexMap()

julia> MOI.write_to_file(dest, "file.cbf.gz")

julia> src_2 = MOI.FileFormats.Model(filename = "file.cbf.gz")
A Conic Benchmark Format (CBF) model

julia> MOI.read_from_file(src_2, "file.cbf.gz")

julia> rm("file.cbf.gz")  # Clean up after ourselves.
```

Note how the compression format (GZip) is also automatically detected from the filename.

Unsupported constraints In some cases `src` may contain constraints that are not supported by the file format (e.g., the CBF format supports integer variables but not binary). If so, you should copy `src` to a bridged model using `Bridges.full_bridge_optimizer`:

```
src = MOI.Utilities.Model{Float64}()
x = MOI.add_variable(model)
MOI.add_constraint(model, x, MOI.ZeroOne())
dest = MOI.FileFormats.Model(format = MOI.FileFormats.FORMAT_CBF)
bridged = MOI.Bridges.full_bridge_optimizer(dest, Float64)
MOI.copy_to(bridged, src)
MOI.write_to_file(dest, "my_model.cbf")
```

You should also note that even after bridging, it may still not be possible to write the model to file because of unsupported constraints (e.g., PSD variables in the LP file format).

Read and write to `io` In addition to `write_to_file` and `read_from_file`, you can read and write directly from `IO` streams using `Base.write` and `Base.read!`:

```
julia> src = MOI.Utilities.Model{Float64}()
MOIU.Model{Float64}

julia> dest = MOI.FileFormats.Model(format = MOI.FileFormats.FORMAT_MPS)
A Mathematical Programming System (MPS) model

julia> MOI.copy_to(dest, src)
MathOptInterface.Utilities.IndexMap()

julia> io = IOBuffer();

julia> write(io, dest)

julia> seekstart(io);

julia> src_2 = MOI.FileFormats.Model(format = MOI.FileFormats.FORMAT_MPS)
A Mathematical Programming System (MPS) model

julia> read!(io, src_2);
```

Validating MOF files MathOptFormat files are governed by a schema. Use JSONSchema.jl to check if a `.mof.json` file satisfies the schema.

First, construct the schema object as follows:

```
julia> import JSON, JSONSchema

julia> schema = JSONSchema.Schema(JSON.parsefile(MOI.FileFormats.MOF.SCHEMA_PATH))
A JSONSchema
```

Then, check if a model file is valid using `isvalid`:

```
julia> good_model = JSON.parse("""
       {
         "version": {
           "major": 1,
           "minor": 0
         },
         "variables": [{"name": "x"}],
         "objective": {"sense": "feasibility"},
         "constraints": []
       }
       """);

julia> isvalid(good_model, schema)
true
```

If we construct an invalid file, for example by mis-typing `name` as `NaMe`, the validation fails:

```
julia> bad_model = JSON.parse("""
       {
         "version": {
           "major": 1,
           "minor": 0
         },
         "variables": [{"NaMe": "x"}],
         "objective": {"sense": "feasibility"},
         "constraints": []
       }
       """);

julia> isvalid(bad_model, schema)
false
```

Use `JSONSchema.validate` to obtain more insight into why the validation failed:

```
julia> JSONSchema.validate(bad_model, schema)
Validation failed:
path:         [variables][1]
instance:     Dict{String, Any}("NaMe" => "x")
schema key:   required
schema value: Any["name"]
```

API Reference

File Formats Functions to help read and write MOI models to/from various file formats. See The FileFormats submodule for more details.

`MathOptInterface.FileFormats.Model` – Function.

```
Model(
    ;
    format::FileFormat = FORMAT_AUTOMATIC,
    filename::Union{Nothing, String} = nothing,
    kwargs...
)
```

Return model corresponding to the `FileFormat` `format`, or, if `format == FORMAT_AUTOMATIC`, guess the format from `filename`.

The `filename` argument is only needed if `format == FORMAT_AUTOMATIC`.

`kwargs` are passed to the underlying model constructor.

`MathOptInterface.FileFormats.FileFormat` – Type.

```
FileFormat
```

List of accepted export formats.

- `FORMAT_AUTOMATIC`: try to detect the file format based on the file name
- `FORMAT_CBF`: the Conic Benchmark format
- `FORMAT_LP`: the LP file format
- `FORMAT_MOF`: the MathOptFormat file format
- `FORMAT_MPS`: the MPS file format
- `FORMAT_NL`: the AMPL .nl file format
- `FORMAT_SDPA`: the SemiDefinite Programming Algorithm format

40.4 Utilities

Overview

The Utilities submodule

The Utilities submodule provides a variety of functionality for managing `MOI.ModelLike` objects.

Utilities.Model `Utilities.Model` provides an implementation of a `ModelLike` that efficiently supports all functions and sets defined within MOI. However, given the extensibility of MOI, this might not cover all use cases.

Create a model as follows:

```
julia> model = MOI.Utilities.Model{Float64}()
MOIU.Model{Float64}
```

Utilities.UniversalFallback `Utilities.UniversalFallback` is a layer that sits on top of any `ModelLike` and provides non-specialized (slower) fallbacks for constraints and attributes that the underlying `ModelLike` does not support.

For example, `Utilities.Model` doesn't support some variable attributes like `VariablePrimalStart`, so JuMP uses a combination of Universal fallback and `Utilities.Model` as a generic problem cache:

```
julia> model = MOI.Utilities.UniversalFallback(MOI.Utilities.Model{Float64}())
MOIU.UniversalFallback{MOIU.Model{Float64}}
fallback for MOIU.Model{Float64}
```

> **Warning**
>
> Adding a UniversalFallback means that your `model` will now support all constraints, even if the inner-model does not! This can lead to unexpected behavior.

Utilities.@model For advanced use cases that need efficient support for functions and sets defined outside of MOI (but still known at compile time), we provide the `Utilities.@model` macro.

The `@model` macro takes a name (for a new type, which must not exist yet), eight tuples specifying the types of constraints that are supported, and then a `Bool` indicating the type should be a subtype of `MOI.AbstractOptimizer` (if `true`) or `MOI.ModelLike` (if `false`).

The eight tuples are in the following order:

1. Un-typed scalar sets, e.g., `Integer`
2. Typed scalar sets, e.g., `LessThan`
3. Un-typed vector sets, e.g., `Nonnegatives`
4. Typed vector sets, e.g., `PowerCone`
5. Un-typed scalar functions, e.g., `VariableIndex`
6. Typed scalar functions, e.g., `ScalarAffineFunction`
7. Un-typed vector functions, e.g., `VectorOfVariables`
8. Typed vector functions, e.g., `VectorAffineFunction`

The tuples can contain more than one element. Typed-sets should be speficied without their type parameter, i.e., `MOI.LessThan`, not `MOI.LessThan{Float64}`.

Here is an example:

```
julia> MOI.Utilities.@model(
           MyNewModel,
           (MOI.Integer,),                 # Un-typed scalar sets
           (MOI.GreaterThan,),             # Typed scalar sets
           (MOI.Nonnegatives,),            # Un-typed vector sets
           (MOI.PowerCone,),               # Typed vector sets
           (MOI.VariableIndex,),           # Un-typed scalar functions
           (MOI.ScalarAffineFunction,),    # Typed scalar functions
           (MOI.VectorOfVariables,),       # Un-typed vector functions
           (MOI.VectorAffineFunction,),    # Typed vector functions
```

```
            true,                           # <:MOI.AbstractOptimizer?
        )
MathOptInterface.Utilities.GenericOptimizer{T, MathOptInterface.Utilities.ObjectiveContainer{T},
↪  MathOptInterface.Utilities.VariablesContainer{T}, MyNewModelFunctionConstraints{T}} where T

julia> model = MyNewModel{Float64}()
MOIU.GenericOptimizer{Float64, MOIU.ObjectiveContainer{Float64}, MOIU.VariablesContainer{Float64},
↪  MyNewModelFunctionConstraints{Float64}}
```

Warning

MyNewModel supports every VariableIndex-in-Set constraint, as well as VariableIndex, ScalarAffineFunction, and ScalarQuadraticFunction objective functions. Implement MOI.supports as needed to forbid constraint and objective function combinations.

As another example, PATHSolver, which only supports VectorAffineFunction-in-Complements defines its optimizer as:

```
julia> MOI.Utilities.@model(
            PathOptimizer,
            (),  # Scalar sets
            (),  # Typed scalar sets
            (MOI.Complements,),  # Vector sets
            (),  # Typed vector sets
            (),  # Scalar functions
            (),  # Typed scalar functions
            (),  # Vector functions
            (MOI.VectorAffineFunction,),  # Typed vector functions
            true,  # is_optimizer
        )
MathOptInterface.Utilities.GenericOptimizer{T, MathOptInterface.Utilities.ObjectiveContainer{T},
↪  MathOptInterface.Utilities.VariablesContainer{T},
↪  MathOptInterface.Utilities.VectorOfConstraints{MathOptInterface.VectorAffineFunction{T},
↪  MathOptInterface.Complements}} where T
```

However, PathOptimizer does not support some VariableIndex-in-Set constraints, so we must explicitly define:

```
julia> function MOI.supports_constraint(
            ::PathOptimizer,
            ::Type{MOI.VariableIndex},
            ::Type{Union{<:MOI.Semiinteger,MOI.Semicontinuous,MOI.ZeroOne,MOI.Integer}}
        )
            return false
        end
```

Finally, PATH doesn't support an objective function, so we need to add:

```
julia> MOI.supports(::PathOptimizer, ::MOI.ObjectiveFunction) = false
```

Warning

This macro creates a new type, so it must be called from the top-level of a module, e.g., it cannot be called from inside a function.

Utilities.CachingOptimizer A `[Utilities.CachingOptimizer]` is an MOI layer that abstracts the difference between solvers that support incremental modification (e.g., they support adding variables one-by-one), and solvers that require the entire problem in a single API call (e.g., they only accept the `A`, `b` and `c` matrices of a linear program).

It has two parts:

1. A cache, where the model can be built and modified incrementally
2. An optimizer, which is used to solve the problem

```
julia> model = MOI.Utilities.CachingOptimizer(
           MOI.Utilities.Model{Float64}(),
           PathOptimizer{Float64}(),
       )
MOIU.CachingOptimizer{MOIU.GenericOptimizer{Float64, MOIU.ObjectiveContainer{Float64},
↪  MOIU.VariablesContainer{Float64}, MOIU.VectorOfConstraints{MOI.VectorAffineFunction{Float64},
↪  MOI.Complements}}, MOIU.Model{Float64}}
in state EMPTY_OPTIMIZER
in mode AUTOMATIC
with model cache MOIU.Model{Float64}
with optimizer MOIU.GenericOptimizer{Float64, MOIU.ObjectiveContainer{Float64},
↪  MOIU.VariablesContainer{Float64}, MOIU.VectorOfConstraints{MOI.VectorAffineFunction{Float64},
↪  MOI.Complements}}
```

A `Utilities.CachingOptimizer` may be in one of three possible states:

- `NO_OPTIMIZER`: The CachingOptimizer does not have any optimizer.
- `EMPTY_OPTIMIZER`: The CachingOptimizer has an empty optimizer, and it is not synchronized with the cached model. Modifications are forwarded to the cache, but not to the optimizer.
- `ATTACHED_OPTIMIZER`: The CachingOptimizer has an optimizer, and it is synchronized with the cached model. Modifications are forwarded to the optimizer. If the optimizer does not support modifications, and error will be thrown.

Use `Utilities.attach_optimizer` to go from `EMPTY_OPTIMIZER` to `ATTACHED_OPTIMIZER`:

```
julia> MOI.Utilities.attach_optimizer(model)

julia> model
MOIU.CachingOptimizer{MOIU.GenericOptimizer{Float64, MOIU.ObjectiveContainer{Float64},
↪  MOIU.VariablesContainer{Float64}, MOIU.VectorOfConstraints{MOI.VectorAffineFunction{Float64},
↪  MOI.Complements}}, MOIU.Model{Float64}}
in state ATTACHED_OPTIMIZER
in mode AUTOMATIC
with model cache MOIU.Model{Float64}
with optimizer MOIU.GenericOptimizer{Float64, MOIU.ObjectiveContainer{Float64},
↪  MOIU.VariablesContainer{Float64}, MOIU.VectorOfConstraints{MOI.VectorAffineFunction{Float64},
↪  MOI.Complements}}
```

> **Info**
>
> You must be in `ATTACHED_OPTIMIZER` to use `optimize!`.

Use `Utilities.reset_optimizer` to go from `ATTACHED_OPTIMIZER` to `EMPTY_OPTIMIZER`:

```
julia> MOI.Utilities.reset_optimizer(model)

julia> model
MOIU.CachingOptimizer{MOIU.GenericOptimizer{Float64, MOIU.ObjectiveContainer{Float64},
↪  MOIU.VariablesContainer{Float64}, MOIU.VectorOfConstraints{MOI.VectorAffineFunction{Float64},
↪  MOI.Complements}}, MOIU.Model{Float64}}
in state EMPTY_OPTIMIZER
in mode AUTOMATIC
with model cache MOIU.Model{Float64}
with optimizer MOIU.GenericOptimizer{Float64, MOIU.ObjectiveContainer{Float64},
↪  MOIU.VariablesContainer{Float64}, MOIU.VectorOfConstraints{MOI.VectorAffineFunction{Float64},
↪  MOI.Complements}}
```

Info

Calling `MOI.empty!(model)` also resets the state to `EMPTY_OPTIMIZER`. So after emptying a model, the modification will only be applied to the cache.

Use `Utilities.drop_optimizer` to go from any state to `NO_OPTIMIZER`:

```
julia> MOI.Utilities.drop_optimizer(model)

julia> model
MOIU.CachingOptimizer{MOIU.GenericOptimizer{Float64, MOIU.ObjectiveContainer{Float64},
↪  MOIU.VariablesContainer{Float64}, MOIU.VectorOfConstraints{MOI.VectorAffineFunction{Float64},
↪  MOI.Complements}}, MOIU.Model{Float64}}
in state NO_OPTIMIZER
in mode AUTOMATIC
with model cache MOIU.Model{Float64}
with optimizer nothing
```

Pass an empty optimizer to `Utilities.reset_optimizer` to go from `NO_OPTIMIZER` to `EMPTY_OPTIMIZER`:

```
julia> MOI.Utilities.reset_optimizer(model, PathOptimizer{Float64}())

julia> model
MOIU.CachingOptimizer{MOIU.GenericOptimizer{Float64, MOIU.ObjectiveContainer{Float64},
↪  MOIU.VariablesContainer{Float64}, MOIU.VectorOfConstraints{MOI.VectorAffineFunction{Float64},
↪  MOI.Complements}}, MOIU.Model{Float64}}
in state EMPTY_OPTIMIZER
in mode AUTOMATIC
with model cache MOIU.Model{Float64}
with optimizer MOIU.GenericOptimizer{Float64, MOIU.ObjectiveContainer{Float64},
↪  MOIU.VariablesContainer{Float64}, MOIU.VectorOfConstraints{MOI.VectorAffineFunction{Float64},
↪  MOI.Complements}}
```

Deciding when to attach and reset the optimizer is tedious, and you will often write code like this:

```
try
    # modification
catch
```

```
    MOI.Utilities.reset_optimizer(model)
    # Re-try modification
end
```

To make this easier, `Utilities.CachingOptimizer` has two modes of operation:

- `AUTOMATIC`: The `CachingOptimizer` changes its state when necessary. Attempting to add a constraint or perform a modification not supported by the optimizer results in a drop to `EMPTY_OPTIMIZER` mode.
- `MANUAL`: The user must change the state of the `CachingOptimizer`. Attempting to perform an operation in the incorrect state results in an error.

By default, `AUTOMATIC` mode is chosen. However, you can create a `CachingOptimizer` in `MANUAL` mode as follows:

```
julia> model = MOI.Utilities.CachingOptimizer(
           MOI.Utilities.Model{Float64}(),
           MOI.Utilities.MANUAL,
       )
MOIU.CachingOptimizer{MOI.AbstractOptimizer, MOIU.Model{Float64}}
in state NO_OPTIMIZER
in mode MANUAL
with model cache MOIU.Model{Float64}
with optimizer nothing

julia> MOI.Utilities.reset_optimizer(model, PathOptimizer{Float64}())

julia> model
MOIU.CachingOptimizer{MOI.AbstractOptimizer, MOIU.Model{Float64}}
in state EMPTY_OPTIMIZER
in mode MANUAL
with model cache MOIU.Model{Float64}
with optimizer MOIU.GenericOptimizer{Float64, MOIU.ObjectiveContainer{Float64},
↪  MOIU.VariablesContainer{Float64}, MOIU.VectorOfConstraints{MOI.VectorAffineFunction{Float64},
↪  MOI.Complements}}
```

Printing Use `print` to print the formulation of the model.

```
julia> model = MOI.Utilities.Model{Float64}();

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

julia> MOI.set(model, MOI.VariableName(), x, "x_var")

julia> MOI.add_constraint(model, x, MOI.ZeroOne())
MathOptInterface.ConstraintIndex{MathOptInterface.VariableIndex, MathOptInterface.ZeroOne}(1)

julia> MOI.set(model, MOI.ObjectiveFunction{typeof(x)}(), x)

julia> MOI.set(model, MOI.ObjectiveSense(), MOI.MAX_SENSE)

julia> print(model)
Maximize VariableIndex:
```

```
 x_var

Subject to:

VariableIndex-in-ZeroOne
 x_var ∈ {0, 1}
```

Use `Utilities.latex_formulation` to display the model in LaTeX form:

```
julia> MOI.Utilities.latex_formulation(model)
$$ \begin{aligned}
\max\quad & x\_var \\
\text{Subject to}\\
 & \text{VariableIndex-in-ZeroOne} \\
 & x\_var \in \{0, 1\} \\
\end{aligned} $$
```

> **Tip**
>
> In IJulia, calling `print` or ending a cell with `Utilities.latex_formulation` will render the model in LaTeX.

Utilities.MatrixOfConstraints The constraints of `Utilities.Model` are stored as a vector of tuples of function and set in a `Utilities.VectorOfConstraints`. Other representations can be used by parametrizing the type `Utilities.GenericModel` (resp. `Utilities.GenericOptimizer`). For instance, if all non-`VariableIndex` constraints are affine, the coefficients of all the constraints can be stored in a single sparse matrix using `Utilities.MatrixOfConstraints`. The constraints storage can even be customized up to a point where it exactly matches the storage of the solver of interest, in which case `copy_to` can be implemented for the solver by calling `copy_to` to this custom model.

For instance, Clp defines the following model

```
MOI.Utilities.@product_of_scalar_sets(LP, MOI.EqualTo{T}, MOI.LessThan{T}, MOI.GreaterThan{T})
const Model = MOI.Utilities.GenericModel{
    Float64,
    MOI.Utilities.MatrixOfConstraints{
        Float64,
        MOI.Utilities.MutableSparseMatrixCSC{Float64,Cint,MOI.Utilities.ZeroBasedIndexing},
        MOI.Utilities.Hyperrectangle{Float64},
        LP{Float64},
    },
}
```

The `copy_to` operation can now be implemented as follows (assuming that the `Model` definition above is in the `Clp` module so that it can be referred to as `Model`, to be distinguished with `Utilities.Model`):

```
function _copy_to(dest::Optimizer, src::Model)
    @assert MOI.is_empty(dest)
    A = src.constraints.coefficients
    row_bounds = src.constraints.constants
    Clp_loadProblem(
        dest,
        A.n,
```

```
        A.m,
        A.colptr,
        A.rowval,
        A.nzval,
        src.lower_bound,
        src.upper_bound,
        # (...) objective vector (omitted),
        row_bounds.lower,
        row_bounds.upper,
    )
    # Set objective sense and constant (omitted)
    return
end

function MOI.copy_to(dest::Optimizer, src::Model)
    _copy_to(dest, src)
    return MOI.Utilities.identity_index_map(src)
end

function MOI.copy_to(
    dest::Optimizer,
    src::MOI.Utilities.UniversalFallback{Model},
)
    # Copy attributes from `src` to `dest` and error in case any unsupported
    # constraints or attributes are set in `UniversalFallback`.
    return MOI.copy_to(dest, src.model)
end

function MOI.copy_to(
    dest::Optimizer,
    src::MOI.ModelLike,
)
    model = Model()
    index_map = MOI.copy_to(model, src)
    _copy_to(dest, model)
    return index_map
end
```

ModelFilter Utilities provides `Utilities.ModelFilter` as a useful tool to copy a subset of a model. For example, given an infeasible model, we can copy the irreducible infeasible subsystem (for models implementing `ConstraintConflictStatus`) as follows:

```
my_filter(::Any) = true
function my_filter(ci::MOI.ConstraintIndex)
    status = MOI.get(dest, MOI.ConstraintConflictStatus(), ci)
    return status != MOI.NOT_IN_CONFLICT
end
filtered_src = MOI.Utilities.ModelFilter(my_filter, src)
index_map = MOI.copy_to(dest, filtered_src)
```

Fallbacks The value of some attributes can be inferred from the value of other attributes.

For example, the value of `ObjectiveValue` can be computed using `ObjectiveFunction` and `VariablePrimal`.

When a solver gives direct access to an attribute, it is better to return this value. However, if this is not the case, `Utilities.get_fallback` can be used instead. For example:

```
function MOI.get(model::Optimizer, attr::MOI.ObjectiveFunction)
    return MOI.Utilities.get_fallback(model, attr)
end
```

DoubleDicts When writing MOI interfaces, we often need to handle situations in which we map `ConstraintIndex`s to different values. For example, to a string for `ConstraintName`.

One option is to use a dictionary like `Dict{MOI.ConstraintIndex,String}`. However, this incurs a performance cost because the key is not a concrete type.

The DoubleDicts submodule helps this situation by providing two types main types `Utilities.DoubleDicts.DoubleDict` and `Utilities.DoubleDicts.IndexDoubleDict`. These types act like normal dictionaries, but internally they use more efficient dictionaries specialized to the type of the function-set pair.

The most common usage of a `DoubleDict` is in the `index_map` returned by `copy_to`. Performance can be improved, by using a function barrier. That is, instead of code like:

```
index_map = MOI.copy_to(dest, src)
for (F, S) in MOI.get(src, MOI.ListOfConstraintTypesPresent())
    for ci in MOI.get(src, MOI.ListOfConstraintIndices{F,S}())
        dest_ci = index_map[ci]
        # ...
    end
end
```

use instead:

```
function function_barrier(
    dest,
    src,
    index_map::MOI.Utilities.DoubleDicts.IndexDoubleDictInner{F,S},
) where {F,S}
    for ci in MOI.get(src, MOI.ListOfConstraintIndices{F,S}())
        dest_ci = index_map[ci]
        # ...
    end
    return
end

index_map = MOI.copy_to(dest, src)
for (F, S) in MOI.get(src, MOI.ListOfConstraintTypesPresent())
    function_barrier(dest, src, index_map[F, S])
end
```

API Reference

Utilities.Model `MathOptInterface.Utilities.Model` – Type.

An implementation of `ModelLike` that supports all functions and sets defined in MOI. It is parameterized by the coefficient type.

Examples

```
model = Model{Float64}()
x = add_variable(model)
```

Utilities.UniversalFallback MathOptInterface.Utilities.UniversalFallback – Type.

```
UniversalFallback
```

The `UniversalFallback` can be applied on a `MathOptInterface.ModelLike` `model` to create the model `UniversalFallback(model)` supporting any constraint and attribute. This allows to have a specialized implementation in `model` for performance critical constraints and attributes while still supporting other attributes with a small performance penalty. Note that `model` is unaware of constraints and attributes stored by `UniversalFallback` so this is not appropriate if `model` is an optimizer (for this reason, `MathOptInterface.optimize!` has not been implemented). In that case, optimizer bridges should be used instead.

Utilities.@model MathOptInterface.Utilities.@model – Macro.

```
macro model(
    model_name,
    scalar_sets,
    typed_scalar_sets,
    vector_sets,
    typed_vector_sets,
    scalar_functions,
    typed_scalar_functions,
    vector_functions,
    typed_vector_functions,
    is_optimizer = false
)
```

Creates a type `model_name` implementing the MOI model interface and containing `scalar_sets` scalar sets `typed_scalar_sets` typed scalar sets, `vector_sets` vector sets, `typed_vector_sets` typed vector sets, `scalar_functions` scalar functions, `typed_scalar_functions` typed scalar functions, `vector_functions` vector functions and `typed_vector_functions` typed vector functions. To give no set/function, write `()`, to give one set `S`, write `(S,)`.

The function `MathOptInterface.VariableIndex` should not be given in `scalar_functions`. The model supports `MathOptInterface.VariableIndex`-in-`S` constraints where `S` is `MathOptInterface.EqualTo`, `MathOptInterface.Gr` `MathOptInterface.LessThan`, `MathOptInterface.Interval`, `MathOptInterface.Integer`, `MathOptInterface.ZeroOne`, `MathOptInterface.Semicontinuous` or `MathOptInterface.Semiinteger`. The sets supported with the `MathOptInterface.VariableIndex` cannot be controlled from the macro, use the `UniversalFallback` to support more sets.

This macro creates a model specialized for specific types of constraint, by defining specialized structures and methods. To create a model that, in addition to be optimized for specific constraints, also support arbitrary constraints and attributes, use `UniversalFallback`.

If `is_optimizer = true`, the resulting struct is a of `GenericOptimizer`, which is a subtype of `MathOptInterface.AbstractOp` otherwise, it is a `GenericModel`, which is a subtype of `MathOptInterface.ModelLike`.

Examples

The model describing an linear program would be:

```
@model(LPModel,                                                 # Name of model
       (),                                                      # untyped scalar sets
       (MOI.EqualTo, MOI.GreaterThan, MOI.LessThan, MOI.Interval), #   typed scalar sets
       (MOI.Zeros, MOI.Nonnegatives, MOI.Nonpositives),         # untyped vector sets
       (),                                                      #   typed vector sets
       (),                                                      # untyped scalar functions
```

```
      (MOI.ScalarAffineFunction,),                                  #   typed scalar functions
      (MOI.VectorOfVariables,),                                     # untyped vector functions
      (MOI.VectorAffineFunction,),                                  #   typed vector functions
      false
    )
```

Let `MOI` denote `MathOptInterface`, `MOIU` denote `MOI.Utilities`. The macro would create the following types with `struct_of_constraint_code`:

```
struct LPModelScalarConstraints{T, C1, C2, C3, C4} <: MOIU.StructOfConstraints
    moi_equalto::C1
    moi_greaterthan::C2
    moi_lessthan::C3
    moi_interval::C4
end
struct LPModelVectorConstraints{T, C1, C2, C3} <: MOIU.StructOfConstraints
    moi_zeros::C1
    moi_nonnegatives::C2
    moi_nonpositives::C3
end
struct LPModelFunctionConstraints{T} <: MOIU.StructOfConstraints
    moi_scalaraffinefunction::LPModelScalarConstraints{
        T,
        MOIU.VectorOfConstraints{MOI.ScalarAffineFunction{T}, MOI.EqualTo{T}},
        MOIU.VectorOfConstraints{MOI.ScalarAffineFunction{T}, MOI.GreaterThan{T}},
        MOIU.VectorOfConstraints{MOI.ScalarAffineFunction{T}, MOI.LessThan{T}},
        MOIU.VectorOfConstraints{MOI.ScalarAffineFunction{T}, MOI.Interval{T}}
    }
    moi_vectorofvariables::LPModelVectorConstraints{
        T,
        MOIU.VectorOfConstraints{MOI.VectorOfVariables, MOI.Zeros},
        MOIU.VectorOfConstraints{MOI.VectorOfVariables, MOI.Nonnegatives},
        MOIU.VectorOfConstraints{MOI.VectorOfVariables, MOI.Nonpositives}
    }
    moi_vectoraffinefunction::LPModelVectorConstraints{
        T,
        MOIU.VectorOfConstraints{MOI.VectorAffineFunction{T}, MOI.Zeros},
        MOIU.VectorOfConstraints{MOI.VectorAffineFunction{T}, MOI.Nonnegatives},
        MOIU.VectorOfConstraints{MOI.VectorAffineFunction{T}, MOI.Nonpositives}
    }
end
const LPModel{T} =
↪  MOIU.GenericModel{T,MOIU.ObjectiveContainer{T},MOIU.VariablesContainer{T},LPModelFunctionConstraints{T}}
```

The type `LPModel` implements the MathOptInterface API except methods specific to optimizers like `optimize!` or `get` with `VariablePrimal`.

`MathOptInterface.Utilities.GenericModel` – Type.

```
mutable struct GenericModel{T,O,V,C} <: AbstractModelLike{T}
```

Implements a model supporting coefficients of type `T` and:

- An objective function stored in `.objective::O`
- Variables and `VariableIndex` constraints stored in `.variable_bounds::V`

- F-in-S constraints (excluding `VariableIndex` constraints) stored in `.constraints::C`

All interactions should take place via the MOI interface, so the types `O`, `V`, and `C` should implement the API as needed for their functionality.

`MathOptInterface.Utilities.GenericOptimizer` – Type.

```
mutable struct GenericOptimizer{T,O,V,C} <: AbstractOptimizer{T}
```

Implements a model supporting coefficients of type `T` and:

- An objective function stored in `.objective::O`
- Variables and `VariableIndex` constraints stored in `.variable_bounds::V`
- F-in-S constraints (excluding `VariableIndex` constraints) stored in `.constraints::C`

All interactions should take place via the MOI interface, so the types `O`, `V`, and `C` should implement the API as needed for their functionality.

`MathOptInterface.Utilities.@struct_of_constraints_by_function_types` – Macro.

```
Utilities.@struct_of_constraints_by_function_types(name, func_types...)
```

Given a vector of n function types `(F1, F2,..., Fn)` in `func_types`, defines a subtype of `StructOfConstraints` of name `name` and which type parameters `{T, C1, C2, ..., Cn}`. It contains n field where the `i`th field has type `Ci` and stores the constraints of function type `Fi`.

The expression `Fi` can also be a union in which case any constraint for which the function type is in the union is stored in the field with type `Ci`.

`MathOptInterface.Utilities.@struct_of_constraints_by_set_types` – Macro.

```
Utilities.@struct_of_constraints_by_set_types(name, func_types...)
```

Given a vector of n set types `(S1, S2,..., Sn)` in `func_types`, defines a subtype of `StructOfConstraints` of name `name` and which type parameters `{T, C1, C2, ..., Cn}`. It contains n field where the `i`th field has type `Ci` and stores the constraints of set type `Si`. The expression `Si` can also be a union in which case any constraint for which the set type is in the union is stored in the field with type `Ci`. This can be useful if `Ci` is a `MatrixOfConstraints` in order to concatenate the coefficients of constraints of several different set types in the same matrix.

`MathOptInterface.Utilities.struct_of_constraint_code` – Function.

```
struct_of_constraint_code(struct_name, types, field_types = nothing)
```

Given a vector of n `Union{SymbolFun,_UnionSymbolFS{SymbolFun}}` or `Union{SymbolSet,_UnionSymbolFS{SymbolSet}}` in `types`, defines a subtype of `StructOfConstraints` of name `name` and which type parameters `{T, F1, F2, ..., Fn}` if `field_types` is `nothing` and a `{T}` otherwise. It contains n field where the `i`th field has type `Ci` if `field_types` is `nothing` and type `field_types[i]` otherwise. If `types` is vector of `Union{SymbolFun,_UnionSymbolFS{SymbolFun}}` (resp. `Union{SymbolSet,_UnionSymbolFS{SymbolSet}}`) then the constraints of that function (resp. set) type are stored in the corresponding field.

This function is used by the macros `@model`, `@struct_of_constraints_by_function_types` and `@struct_of_constraints_b`

Caching optimizer MathOptInterface.Utilities.CachingOptimizer – Type.

```
CachingOptimizer
```

`CachingOptimizer` is an intermediate layer that stores a cache of the model and links it with an optimizer. It supports incremental model construction and modification even when the optimizer doesn't.

A `CachingOptimizer` may be in one of three possible states (`CachingOptimizerState`):

- `NO_OPTIMIZER`: The CachingOptimizer does not have any optimizer.
- `EMPTY_OPTIMIZER`: The CachingOptimizer has an empty optimizer. The optimizer is not synchronized with the cached model.
- `ATTACHED_OPTIMIZER`: The CachingOptimizer has an optimizer, and it is synchronized with the cached model.

A `CachingOptimizer` has two modes of operation (`CachingOptimizerMode`):

- `MANUAL`: The only methods that change the state of the `CachingOptimizer` are `Utilities.reset_optimizer`, `Utilities.drop_optimizer`, and `Utilities.attach_optimizer`. Attempting to perform an operation in the incorrect state results in an error.
- `AUTOMATIC`: The `CachingOptimizer` changes its state when necessary. For example, `optimize!` will automatically call `attach_optimizer` (an optimizer must have been previously set). Attempting to add a constraint or perform a modification not supported by the optimizer results in a drop to `EMPTY_OPTIMIZER` mode.

MathOptInterface.Utilities.attach_optimizer – Function.

```
attach_optimizer(model::CachingOptimizer)
```

Attaches the optimizer to `model`, copying all model data into it. Can be called only from the `EMPTY_OPTIMIZER` state. If the copy succeeds, the `CachingOptimizer` will be in state `ATTACHED_OPTIMIZER` after the call, otherwise an error is thrown; see `MathOptInterface.copy_to` for more details on which errors can be thrown.

```
MOIU.attach_optimizer(model::Model)
```

Call `MOIU.attach_optimizer` on the backend of `model`.

Cannot be called in direct mode.

source

MathOptInterface.Utilities.reset_optimizer – Function.

```
reset_optimizer(m::CachingOptimizer, optimizer::MOI.AbstractOptimizer)
```

Sets or resets `m` to have the given empty optimizer `optimizer`.

Can be called from any state. An assertion error will be thrown if `optimizer` is not empty.

The `CachingOptimizer` `m` will be in state `EMPTY_OPTIMIZER` after the call.

```
reset_optimizer(m::CachingOptimizer)
```

Detaches and empties the current optimizer. Can be called from `ATTACHED_OPTIMIZER` or `EMPTY_OPTIMIZER` state. The `CachingOptimizer` will be in state `EMPTY_OPTIMIZER` after the call.

```
MOIU.reset_optimizer(model::Model, optimizer::MOI.AbstractOptimizer)
```

Call `MOIU.reset_optimizer` on the backend of `model`.

Cannot be called in direct mode.

source

```
MOIU.reset_optimizer(model::Model)
```

Call `MOIU.reset_optimizer` on the backend of `model`.

Cannot be called in direct mode.

source

`MathOptInterface.Utilities.drop_optimizer` – Function.

```
drop_optimizer(m::CachingOptimizer)
```

Drops the optimizer, if one is present. Can be called from any state. The `CachingOptimizer` will be in state `NO_OPTIMIZER` after the call.

```
MOIU.drop_optimizer(model::Model)
```

Call `MOIU.drop_optimizer` on the backend of `model`.

Cannot be called in direct mode.

source

`MathOptInterface.Utilities.state` – Function.

```
state(m::CachingOptimizer)::CachingOptimizerState
```

Returns the state of the CachingOptimizer `m`. See `Utilities.CachingOptimizer`.

`MathOptInterface.Utilities.mode` – Function.

```
mode(m::CachingOptimizer)::CachingOptimizerMode
```

Returns the operating mode of the CachingOptimizer `m`. See `Utilities.CachingOptimizer`.

Mock optimizer `MathOptInterface.Utilities.MockOptimizer` – Type.

```
MockOptimizer
```

`MockOptimizer` is a fake optimizer especially useful for testing. Its main feature is that it can store the values that should be returned for each attribute.

Printing `MathOptInterface.Utilities.latex_formulation` – Function.

```
latex_formulation(model::MOI.ModelLike; kwargs...)
```

Wrap `model` in a type so that it can be pretty-printed as `text/latex` in a notebook like IJulia, or in Documenter.

To render the model, end the cell with `latex_formulation(model)`, or call `display(latex_formulation(model))` in to force the display of the model from inside a function.

Possible keyword arguments are:

- `simplify_coefficients` : Simplify coefficients if possible by omitting them or removing trailing zeros.
- `default_name` : The name given to variables with an empty name.
- `print_types` : Print the MOI type of each function and set for clarity.

Copy utilities `MathOptInterface.Utilities.default_copy_to` – Function.

```
default_copy_to(dest::MOI.ModelLike, src::MOI.ModelLike)
```

A default implementation of `MOI.copy_to(dest, src)` for models that implement the incremental interface, i.e., `MOI.supports_incremental_interface` returns `true`.

`MathOptInterface.Utilities.IndexMap` – Type.

```
IndexMap()
```

The dictionary-like object returned by `MathOptInterface.copy_to`.

`MathOptInterface.Utilities.identity_index_map` – Function.

```
identity_index_map(model::MOI.ModelLike)
```

Return an `IndexMap` that maps all variable and constraint indices of `model` to themselves.

`MathOptInterface.Utilities.ModelFilter` – Type.

```
ModelFilter(filter::Function, model::MOI.ModelLike)
```

A layer to filter out various components of `model`.

The filter function takes a single argument, which is eacy element from the list returned by the attributes below. It returns `true` if the element should be visible in the filtered model and `false` otherwise.

The components that are filtered are:

- Entire constraint types via:
  - `MOI.ListOfConstraintTypesPresent`
- Individual constraints via:
  - `MOI.ListOfConstraintIndices{F,S}`
- Specific attributes via:
  - `MOI.ListOfModelAttributesSet`
  - `MOI.ListOfConstraintAttributesSet`
  - `MOI.ListOfVariableAttributesSet`

Warning

The list of attributes filtered may change in a future release. You should write functions that are generic and not limited to the five types listed above. Thus, you should probably define a fallback `filter(::Any) = true`.

See below for examples of how this works.

Note

This layer has a limited scope. It is intended by be used in conjunction with `MOI.copy_to`.

Example: copy model excluding integer constraints

Use the do syntax to provide a single function.

```
filtered_src = MOI.Utilities.ModelFilter(src) do item
    return item != (MOI.VariableIndex, MOI.Integer)
end
MOI.copy_to(dest, filtered_src)
```

Example: copy model excluding names

Use type dispatch to simplify the implementation:

```
my_filter(::Any) = true  # Note the generic fallback!
my_filter(::MOI.VariableName) = false
my_filter(::MOI.ConstraintName) = false
filtered_src = MOI.Utilities.ModelFilter(my_filter, src)
MOI.copy_to(dest, filtered_src)
```

Example: copy irreducible infeasible subsystem

```
my_filter(::Any) = true  # Note the generic fallback!
function my_filter(ci::MOI.ConstraintIndex)
    status = MOI.get(dest, MOI.ConstraintConflictStatus(), ci)
    return status != MOI.NOT_IN_CONFLICT
end
filtered_src = MOI.Utilities.ModelFilter(my_filter, src)
MOI.copy_to(dest, filtered_src)
```

MatrixOfConstraints `MathOptInterface.Utilities.MatrixOfConstraints` – Type.

```
mutable struct MatrixOfConstraints{T,AT,BT,ST} <: MOI.ModelLike
    coefficients::AT
    constants::BT
    sets::ST
    caches::Vector{Any}
    are_indices_mapped::Vector{BitSet}
    final_touch::Bool
end
```

Represent `ScalarAffineFunction` and `VectorAffinefunction` constraints in a matrix form where the linear coefficients of the functions are stored in the `coefficients` field, the constants of the functions or sets are stored in the `constants` field. Additional information about the sets are stored in the `sets` field.

This model can only be used as the `constraints` field of a `MOI.Utilities.AbstractModel`.

When the constraints are added, they are stored in the `caches` field. They are only loaded in the `coefficients` and `constants` fields once `MOI.Utilities.final_touch` is called. For this reason, `MatrixOfConstraints` should not be used by an incremental interface. Use `MOI.copy_to` instead.

The constraints can be added in two different ways:

1. With `add_constraint`, in which case a canonicalized copy of the function is stored in `caches`.
2. With `pass_nonvariable_constraints`, in which case the functions and sets are stored themselves in `caches` without mapping the variable indices. The corresponding index in `caches` is added in `are_indices_mapped`. This avoids doing a copy of the function in case the getter of `CanonicalConstraintFunction` does not make a copy for the source model, e.g., this is the case of `VectorOfConstraints`.

We illustrate this with an example. Suppose a model is copied from a `src::MOI.Utilities.Model` to a bridged model with a `MatrixOfConstraints`. For all the types that are not bridged, the constraints will be copied with `pass_nonvariable_constraints`. Hence the functions stored in caches are exactly the same as the ones stored in `src`. This is ok since this is only during the `copy_to` operation during which `src` cannot be modified. On the other hand, for the types that are bridged, the functions added may contain duplicates even if the functions did not contain duplicates in `src` so duplicates are removed with `MOI.Utilities.canonical`.

Interface

The `.coefficients::AT` type must implement:

- `AT()`
- `MOI.empty(::AT)!`
- `MOI.Utilities.add_column`
- `MOI.Utilities.set_number_of_rows`
- `MOI.Utilities.allocate_terms`
- `MOI.Utilities.load_terms`
- `MOI.Utilities.final_touch`

The `.constants::BT` type must implement:

- `BT()`
- `Base.empty!(::BT)`
- `Base.resize(::BT)`
- `MOI.Utilities.load_constants`
- `MOI.Utilities.function_constants`
- `MOI.Utilities.set_from_constants`

The `.sets::ST` type must implement:

- `ST()`
- `MOI.is_empty(::ST)`
- `MOI.empty(::ST)`
- `MOI.dimension(::ST)`
- `MOI.is_valid(::ST, ::MOI.ConstraintIndex)`
- `MOI.get(::ST, ::MOI.ListOfConstraintTypesPresent)`
- `MOI.get(::ST, ::MOI.NumberOfConstraints)`
- `MOI.get(::ST, ::MOI.ListOfConstraintIndices)`
- `MOI.Utilities.set_types`
- `MOI.Utilities.set_index`
- `MOI.Utilities.add_set`
- `MOI.Utilities.rows`
- `MOI.Utilities.final_touch`

.coefficients MathOptInterface.Utilities.add_column – Function.

```
add_column(coefficients)::Nothing
```

Tell `coefficients` to pre-allocate datastructures as needed to store one column.

MathOptInterface.Utilities.allocate_terms – Function.

```
allocate_terms(coefficients, index_map, func)::Nothing
```

Tell `coefficients` that the terms of the function `func` where the variable indices are mapped with `index_map` will be loaded with `load_terms`.

The function `func` must be canonicalized before calling `allocate_terms`. See `is_canonical`.

MathOptInterface.Utilities.set_number_of_rows – Function.

```
set_number_of_rows(coefficients, n)::Nothing
```

Tell `coefficients` to pre-allocate datastructures as needed to store `n` rows.

MathOptInterface.Utilities.load_terms – Function.

```
load_terms(coefficients, index_map, func, offset)::Nothing
```

Loads the terms of `func` to `coefficients`, mapping the variable indices with `index_map`.

The `i`th dimension of `func` is loaded at the `(offset + i)`th row of `coefficients`.

The function must be allocated first with `allocate_terms`.

The function `func` must be canonicalized, see `is_canonical`.

MathOptInterface.Utilities.final_touch – Function.

```
final_touch(coefficients)::Nothing
```

Informs the `coefficients` that all functions have been added with `load_terms`. No more modification is allowed unless `MOI.empty!` is called.

```
final_touch(sets)::Nothing
```

Informs the `sets` that all functions have been added with `add_set`. No more modification is allowed unless `MOI.empty!` is called.

MathOptInterface.Utilities.extract_function – Function.

```
extract_function(coefficients, row::Integer, constant::T) where {T}
```

Return the `MOI.ScalarAffineFunction{T}` function corresponding to row `row` in `coefficients`.

```
extract_function(
    coefficients,
    rows::UnitRange,
    constants::Vector{T},
) where{T}
```

Return the `MOI.VectorAffineFunction{T}` function corresponding to rows `rows` in `coefficients`.

MathOptInterface.Utilities.MutableSparseMatrixCSC – Type.

```
mutable struct MutableSparseMatrixCSC{Tv,Ti<:Integer,I<:AbstractIndexing}
    indexing::I
    m::Int
    n::Int
    colptr::Vector{Ti}
    rowval::Vector{Ti}
    nzval::Vector{Tv}
end
```

Matrix type loading sparse matrices in the Compressed Sparse Column format. The indexing used is `indexing`, see `AbstractIndexing`. The other fields have the same meaning than for `SparseArrays.SparseMatrixCSC` except that the indexing is different unless `indexing` is `OneBasedIndexing`.

The matrix is loaded in 5 steps:

1. `MOI.empty!` is called.
2. `MOI.Utilities.add_column` and `MOI.Utilities.allocate_terms` are called in any order.
3. `MOI.Utilities.set_number_of_rows` is called.
4. `MOI.Utilities.load_terms` is called for each affine function.
5. `MOI.Utilities.final_touch` is called.

`MathOptInterface.Utilities.AbstractIndexing` – Type.

```
abstract type AbstractIndexing end
```

Indexing to be used for storing the row and column indices of `MutableSparseMatrixCSC`. See `ZeroBasedIndexing` and `OneBasedIndexing`.

`MathOptInterface.Utilities.ZeroBasedIndexing` – Type.

```
struct ZeroBasedIndexing <: AbstractIndexing end
```

Zero-based indexing: the `i`th row or column has index `i - 1`. This is useful when the vectors of row and column indices need to be communicated to a library using zero-based indexing such as C libraries.

`MathOptInterface.Utilities.OneBasedIndexing` – Type.

```
struct ZeroBasedIndexing <: AbstractIndexing end
```

One-based indexing: the `i`th row or column has index `i`. This enables an allocation-free conversion of `MutableSparseMatrixCSC` to `SparseArrays.SparseMatrixCSC`.

`.constants` `MathOptInterface.Utilities.load_constants` – Function.

```
load_constants(constants, offset, func_or_set)::Nothing
```

This function loads the constants of `func_or_set` in `constants` at an offset of `offset`. Where `offset` is the sum of the dimensions of the constraints already loaded. The storage should be preallocated with `resize!` before calling this function.

This function should be implemented to be usable as storage of constants for `MatrixOfConstraints`.

The constants are loaded in three steps:

1. `Base.empty!` is called.

2. `Base.resize!` is called with the sum of the dimensions of all constraints.
3. `MOI.Utilities.load_constants` is called for each function for vector constraint or set for scalar constraint.

`MathOptInterface.Utilities.function_constants` – Function.

```
function_constants(constants, rows)
```

This function returns the function constants that were loaded with `load_constants` at the rows `rows`.

This function should be implemented to be usable as storage of constants for `MatrixOfConstraints`.

`MathOptInterface.Utilities.set_from_constants` – Function.

```
set_from_constants(constants, S::Type, rows)::S
```

This function returns an instance of the set S for which the constants where loaded with `load_constants` at the rows `rows`.

This function should be implemented to be usable as storage of constants for `MatrixOfConstraints`.

`MathOptInterface.Utilities.Hyperrectangle` – Type.

```
struct Hyperrectangle{T} <: AbstractVectorBounds
    lower::Vector{T}
    upper::Vector{T}
end
```

A struct for the .constants field in MatrixOfConstraints.

`.sets` `MathOptInterface.Utilities.set_index` – Function.

```
set_index(sets, ::Type{S})::Union{Int,Nothing} where {S<:MOI.AbstractSet}
```

Return an integer corresponding to the index of the set type in the list given by `set_types`.

If S is not part of the list, return `nothing`.

`MathOptInterface.Utilities.set_types` – Function.

```
set_types(sets)::Vector{Type}
```

Return the list of the types of the sets allowed in `sets`.

`MathOptInterface.Utilities.add_set` – Function.

```
add_set(sets, i)::Int64
```

Add a scalar set of type index `i`.

```
add_set(sets, i, dim)::Int64
```

Add a vector set of type index `i` and dimension `dim`.

Both methods return a unique `Int64` of the set that can be used to reference this set.

`MathOptInterface.Utilities.rows` – Function.

```
rows(sets, ci::MOI.ConstraintIndex)::Union{Int,UnitRange{Int}}
```

Return the rows in `1:MOI.dimension(sets)` corresponding to the set of id `ci.value`.

For scalar sets, this returns an `Int`. For vector sets, this returns an `UnitRange{Int}`.

MathOptInterface.Utilities.set_with_dimension – Function.

```
set_with_dimension(::Type{S}, dim) where {S<:MOI.AbstractVectorSet}
```

Returns the instance of S of MathOptInterface.dimension `dim`. This needs to be implemented for sets of type S to be useable with MatrixOfConstraints.

MathOptInterface.Utilities.ProductOfSets – Type.

```
abstract type ProductOfSets{T} end
```

Represents a cartesian product of sets of given types.

MathOptInterface.Utilities.MixOfScalarSets – Type.

```
abstract type MixOfScalarSets{T} <: ProductOfSets{T} end
```

Product of scalar sets in the order the constraints are added, mixing the constraints of different types.

Use @mix_of_scalar_sets to generate a new subtype.

MathOptInterface.Utilities.@mix_of_scalar_sets – Macro.

```
@mix_of_scalar_sets(name, set_types...)
```

Generate a new MixOfScalarSets subtype.

Example

```
@mix_of_scalar_sets(
    MixedIntegerLinearProgramSets,
    MOI.GreaterThan{T},
    MOI.LessThan{T},
    MOI.EqualTo{T},
    MOI.Integer,
)
```

MathOptInterface.Utilities.OrderedProductOfSets – Type.

```
abstract type OrderedProductOfSets{T} <: ProductOfSets{T} end
```

Product of sets in the order the constraints are added, grouping the constraints of the same types contiguously.

Use @product_of_sets to generate new subtypes.

MathOptInterface.Utilities.@product_of_sets – Macro.

```
@product_of_sets(name, set_types...)
```

Generate a new OrderedProductOfSets subtype.

Example

```
@product_of_sets(
    LinearOrthants,
    MOI.Zeros,
    MOI.Nonnegatives,
    MOI.Nonpositives,
    MOI.ZeroOne,
)
```

Fallbacks MathOptInterface.Utilities.get_fallback – Function.

```
get_fallback(model::MOI.ModelLike, ::MOI.ObjectiveValue)
```

Compute the objective function value using the `VariablePrimal` results and the `ObjectiveFunction` value.

```
get_fallback(model::MOI.ModelLike, ::MOI.DualObjectiveValue, T::Type)::T
```

Compute the dual objective value of type `T` using the `ConstraintDual` results and the `ConstraintFunction` and `ConstraintSet` values. Note that the nonlinear part of the model is ignored.

```
get_fallback(model::MOI.ModelLike, ::MOI.ConstraintPrimal,
             constraint_index::MOI.ConstraintIndex)
```

Compute the value of the function of the constraint of index `constraint_index` using the `VariablePrimal` results and the `ConstraintFunction` values.

```
get_fallback(model::MOI.ModelLike, attr::MOI.ConstraintDual,
             ci::MOI.ConstraintIndex{Union{MOI.VariableIndex,
                                           MOI.VectorOfVariables}})
```

Compute the dual of the constraint of index `ci` using the `ConstraintDual` of other constraints and the `ConstraintFunction` values. Throws an error if some constraints are quadratic or if there is one another `MOI.VariableIndex`-in-`S` or `MOI.VectorOfVariables`-in-`S` constraint with one of the variables in the function of the constraint `ci`.

Function utilities The following utilities are available for functions:

MathOptInterface.Utilities.eval_variables – Function.

```
eval_variables(varval::Function, f::AbstractFunction)
```

Returns the value of function `f` if each variable index `vi` is evaluated as `varval(vi)`. Note that `varval` should return a number, see `substitute_variables` for a similar function where `varval` returns a function.

MathOptInterface.Utilities.map_indices – Function.

```
map_indices(index_map::Function, x::X)::X where {X}
```

Substitute any `MOI.VariableIndex` (resp. `MOI.ConstraintIndex`) in `x` by the `MOI.VariableIndex` (resp. `MOI.ConstraintIndex`) of the same type given by `index_map(x)`.

This function is used by implementations of `MOI.copy_to` on constraint functions, attribute values and submittable values hence it needs to be implemented for custom types that are meant to be used as attribute or submittable value.

MathOptInterface.Utilities.substitute_variables – Function.

```
substitute_variables(variable_map::Function, x)
```

Substitute any MOI.VariableIndex in x by variable_map(x). The variable_map function returns either MOI.VariableIndex or MOI.ScalarAffineFunction, see eval_variables for a similar function where variable_map returns a number.

This function is used by bridge optimizers on constraint functions, attribute values and submittable values when at least one variable bridge is used hence it needs to be implemented for custom types that are meant to be used as attribute or submittable value.

WARNING: Don't use substitude_variables(::Function, ...) because Julia will not specialize on this. Use instead substitude_variables(::F, ...) where {F<:Function}.

MathOptInterface.Utilities.filter_variables – Function.

```
filter_variables(keep::Function, f::AbstractFunction)
```

Return a new function f with the variable vi such that !keep(vi) removed.

WARNING: Don't define filter_variables(::Function, ...) because Julia will not specialize on this. Define instead filter_variables(::F, ...) where {F<:Function}.

MathOptInterface.Utilities.remove_variable – Function.

```
remove_variable(f::AbstractFunction, vi::VariableIndex)
```

Return a new function f with the variable vi removed.

```
remove_variable(f::MOI.AbstractFunction, s::MOI.AbstractSet, vi::MOI.VariableIndex)
```

Return a tuple (g, t) representing the constraint f-in-s with the variable vi removed. That is, the terms containing the variable vi in the function f are removed and the dimension of the set s is updated if needed (e.g. when f is a VectorOfVariables with vi being one of the variables).

MathOptInterface.Utilities.all_coefficients – Function.

```
all_coefficients(p::Function, f::MOI.AbstractFunction)
```

Determine whether predicate p returns true for all coefficients of f, returning false as soon as the first coefficient of f for which p returns false is encountered (short-circuiting). Similar to all.

MathOptInterface.Utilities.unsafe_add – Function.

```
unsafe_add(t1::MOI.ScalarAffineTerm, t2::MOI.ScalarAffineTerm)
```

Sums the coefficients of t1 and t2 and returns an output MOI.ScalarAffineTerm. It is unsafe because it uses the variable of t1 as the variable of the output without checking that it is equal to that of t2.

```
unsafe_add(t1::MOI.ScalarQuadraticTerm, t2::MOI.ScalarQuadraticTerm)
```

Sums the coefficients of t1 and t2 and returns an output MOI.ScalarQuadraticTerm. It is unsafe because it uses the variable's of t1 as the variable's of the output without checking that they are the same (up to permutation) to those of t2.

```
unsafe_add(t1::MOI.VectorAffineTerm, t2::MOI.VectorAffineTerm)
```

Sums the coefficients of t1 and t2 and returns an output MOI.VectorAffineTerm. It is unsafe because it uses the output_index and variable of t1 as the output_index and variable of the output term without checking that they are equal to those of t2.

MathOptInterface.Utilities.isapprox_zero – Function.

```
isapprox_zero(f::MOI.AbstractFunction, tol)
```

Return a `Bool` indicating whether the function `f` is approximately zero using `tol` as a tolerance.

Important note

This function assumes that `f` does not contain any duplicate terms, you might want to first call `canonical` if that is not guaranteed. For instance, given

```
f = MOI.ScalarAffineFunction(MOI.ScalarAffineTerm.([1, -1], [x, x]), 0)`.
```

then `isapprox_zero(f)` is `false` but `isapprox_zero(MOIU.canonical(f))` is `true`.

`MathOptInterface.Utilities.modify_function` – Function.

```
modify_function(f::AbstractFunction, change::AbstractFunctionModification)
```

Return a new function `f` modified according to `change`.

`MathOptInterface.Utilities.zero_with_output_dimension` – Function.

```
zero_with_output_dimension(::Type{T}, output_dimension::Integer) where {T}
```

Create an instance of type `T` with the output dimension `output_dimension`.

This is mostly useful in Bridges, when code needs to be agnostic to the type of vector-valued function that is passed in.

The following functions can be used to canonicalize a function:

`MathOptInterface.Utilities.is_canonical` – Function.

```
is_canonical(f::Union{ScalarAffineFunction, VectorAffineFunction})
```

Returns a Bool indicating whether the function is in canonical form. See `canonical`.

```
is_canonical(f::Union{ScalarQuadraticFunction, VectorQuadraticFunction})
```

Returns a Bool indicating whether the function is in canonical form. See `canonical`.

`MathOptInterface.Utilities.canonical` – Function.

```
canonical(
    f::Union{
        ScalarAffineFunction,
        VectorAffineFunction,
        ScalarQuadraticFunction,
        VectorQuadraticFunction,
    },
)
```

Returns the function in a canonical form, i.e.

- A term appear only once.
- The coefficients are nonzero.
- The terms appear in increasing order of variable where there the order of the variables is the order of their value.

- For a `AbstractVectorFunction`, the terms are sorted in ascending order of output index.

The output of `canonical` can be assumed to be a copy of `f`, even for `VectorOfVariables`.

Examples

If x (resp. y, z) is `VariableIndex(1)` (resp. 2, 3). The canonical representation of `ScalarAffineFunction([y, x, z, x, z], [2, 1, 3, -2, -3], 5)` is `ScalarAffineFunction([x, y], [-1, 2], 5)`.

`MathOptInterface.Utilities.canonicalize!` – Function.

```
canonicalize!(f::Union{ScalarAffineFunction, VectorAffineFunction})
```

Convert a function to canonical form in-place, without allocating a copy to hold the result. See `canonical`.

```
canonicalize!(f::Union{ScalarQuadraticFunction, VectorQuadraticFunction})
```

Convert a function to canonical form in-place, without allocating a copy to hold the result. See `canonical`.

The following functions can be used to manipulate functions with basic algebra:

`MathOptInterface.Utilities.scalar_type` – Function.

```
scalar_type(F::Type{<:MOI.AbstractVectorFunction})
```

Type of functions obtained by indexing objects obtained by calling `eachscalar` on functions of type `F`.

`MathOptInterface.Utilities.scalarize` – Function.

```
scalarize(func::MOI.VectorOfVariables, ignore_constants::Bool = false)
```

Returns a vector of scalar functions making up the vector function in the form of a `Vector{MOI.SingleVariable}`.

See also `eachscalar`.

```
scalarize(func::MOI.VectorAffineFunction{T}, ignore_constants::Bool = false)
```

Returns a vector of scalar functions making up the vector function in the form of a `Vector{MOI.ScalarAffineFunction{T}}`.

See also `eachscalar`.

```
scalarize(func::MOI.VectorQuadraticFunction{T}, ignore_constants::Bool = false)
```

Returns a vector of scalar functions making up the vector function in the form of a `Vector{MOI.ScalarQuadraticFunction{T}`

See also `eachscalar`.

`MathOptInterface.Utilities.eachscalar` – Function.

```
eachscalar(f::MOI.AbstractVectorFunction)
```

Returns an iterator for the scalar components of the vector function.

See also `scalarize`.

```
eachscalar(f::MOI.AbstractVector)
```

Returns an iterator for the scalar components of the vector.

`MathOptInterface.Utilities.promote_operation` – Function.

```
promote_operation(
    op::Function,
    ::Type{T},
    ArgsTypes::Type{<:Union{T, MOI.AbstractFunction}}...,
) where {T}
```

Returns the type of the `MOI.AbstractFunction` returned to the call `operate(op, T, args...)` where the types of the arguments `args` are `ArgsTypes`.

`MathOptInterface.Utilities.operate` – Function.

```
operate(
    op::Function,
    ::Type{T},
    args::Union{T,MOI.AbstractFunction}...,
)::MOI.AbstractFunction where {T}
```

Returns an `MOI.AbstractFunction` representing the function resulting from the operation `op(args...)` on functions of coefficient type `T`. No argument can be modified.

`MathOptInterface.Utilities.operate!` – Function.

```
operate!(
    op::Function,
    ::Type{T},
    args::Union{T, MOI.AbstractFunction}...,
)::MOI.AbstractFunction where {T}
```

Returns an `MOI.AbstractFunction` representing the function resulting from the operation `op(args...)` on functions of coefficient type `T`. The first argument can be modified. The return type is the same than the method `operate(op, T, args...)` without `!`.

`MathOptInterface.Utilities.operate_output_index!` – Function.

```
operate_output_index!(
    op::Function,
    ::Type{T},
    output_index::Integer,
    func::MOI.AbstractVectorFunction
    args::Union{T, MOI.AbstractScalarFunction}...
)::MOI.AbstractFunction where {T}
```

Returns an `MOI.AbstractVectorFunction` where the function at `output_index` is the result of the operation `op` applied to the function at `output_index` of `func` and `args`. The functions at output index different to `output_index` are the same as the functions at the same output index in `func`. The first argument can be modified.

`MathOptInterface.Utilities.vectorize` – Function.

```
vectorize(x::AbstractVector{MOI.VariableIndex})
```

Returns the vector of scalar affine functions in the form of a `MOI.VectorAffineFunction{T}`.

```
vectorize(funcs::AbstractVector{MOI.ScalarAffineFunction{T}}) where T
```

Returns the vector of scalar affine functions in the form of a `MOI.VectorAffineFunction{T}`.

```
vectorize(funcs::AbstractVector{MOI.ScalarQuadraticFunction{T}}) where T
```

Returns the vector of scalar quadratic functions in the form of a `MOI.VectorQuadraticFunction{T}`.

Constraint utilities The following utilities are available for moving the function constant to the set for scalar constraints:

`MathOptInterface.Utilities.shift_constant` – Function.

```
shift_constant(set::MOI.AbstractScalarSet, offset)
```

Returns a new scalar set `new_set` such that `func-in-set` is equivalent to `func + offset-in-new_set`.

Only define this function if it makes sense to!

Use `supports_shift_constant` to check if the set supports shifting:

```
if supports_shift_constant(typeof(old_set))
    new_set = shift_constant(old_set, offset)
    f.constant = 0
    add_constraint(model, f, new_set)
else
    add_constraint(model, f, old_set)
end
```

See also `supports_shift_constant`.

Examples

The call `shift_constant(MOI.Interval(-2, 3), 1)` is equal to `MOI.Interval(-1, 4)`.

`MathOptInterface.Utilities.supports_shift_constant` – Function.

```
supports_shift_constant(::Type{S}) where {S<:MOI.AbstractSet}
```

Return `true` if `shift_constant` is defined for set `S`.

See also `shift_constant`.

`MathOptInterface.Utilities.normalize_and_add_constraint` – Function.

```
normalize_and_add_constraint(
    model::MOI.ModelLike,
    func::MOI.AbstractScalarFunction,
    set::MOI.AbstractScalarSet;
    allow_modify_function::Bool = false,
)
```

Adds the scalar constraint obtained by moving the constant term in `func` to the set in `model`. If `allow_modify_function` is `true` then the function `func` can be modified.

`MathOptInterface.Utilities.normalize_constant` – Function.

```
normalize_constant(
    func::MOI.AbstractScalarFunction,
    set::MOI.AbstractScalarSet;
    allow_modify_function::Bool = false,
)
```

Return the `func-in-set` constraint in normalized form. That is, if `func` is `MOI.ScalarQuadraticFunction` or `MOI.ScalarAffineFunction`, the constant is moved to the set. If `allow_modify_function` is `true` then the function `func` can be modified.

The following utility identifies those constraints imposing bounds on a given variable, and returns those bound values:

MathOptInterface.Utilities.get_bounds – Function.

```
get_bounds(model::MOI.ModelLike, ::Type{T}, x::MOI.VariableIndex)
```

Return a tuple `(lb, ub)` of type `Tuple{T, T}`, where `lb` and `ub` are lower and upper bounds, respectively, imposed on `x` in `model`.

The following utilities are useful when working with symmetric matrix cones.

MathOptInterface.Utilities.is_diagonal_vectorized_index – Function.

```
is_diagonal_vectorized_index(index::Base.Integer)
```

Return whether `index` is the index of a diagonal element in a `MOI.AbstractSymmetricMatrixSetTriangle` set.

MathOptInterface.Utilities.side_dimension_for_vectorized_dimension – Function.

```
side_dimension_for_vectorized_dimension(n::Integer)
```

Return the dimension `d` such that `MOI.dimension(MOI.PositiveSemidefiniteConeTriangle(d))` is `n`.

DoubleDicts MathOptInterface.Utilities.DoubleDicts.DoubleDict – Type.

```
DoubleDict{V}
```

An optimized dictionary to map `MOI.ConstraintIndex` to values of type `V`.

Works as a `AbstractDict{MOI.ConstraintIndex,V}` with minimal differences.

If `V` is also a `MOI.ConstraintIndex`, use `IndexDoubleDict`.

Note that `MOI.ConstraintIndex` is not a concrete type, opposed to `MOI.ConstraintIndex{MOI.VariableIndex, MOI.Integers}`, which is a concrete type.

When looping through multiple keys of the same Function-in-Set type, use

```
inner = dict[F, S]
```

to return a type-stable `DoubleDictInner`.

MathOptInterface.Utilities.DoubleDicts.DoubleDictInner – Type.

```
DoubleDictInner{F,S,V}
```

A type stable inner dictionary of `DoubleDict`.

MathOptInterface.Utilities.DoubleDicts.IndexDoubleDict – Type.

```
IndexDoubleDict
```

A specialized version of [`DoubleDict`] in which the values are of type `MOI.ConstraintIndex`

When looping through multiple keys of the same Function-in-Set type, use

```
inner = dict[F, S]
```

to return a type-stable IndexDoubleDictInner.

MathOptInterface.Utilities.DoubleDicts.IndexDoubleDictInner – Type.

```
IndexDoubleDictInner{F,S}
```

A type stable inner dictionary of IndexDoubleDict.

40.5 Test

Overview

The Test submodule

The Test submodule provides tools to help solvers implement unit tests in order to ensure they implement the MathOptInterface API correctly, and to check for solver-correctness.

We use a centralized repository of tests, so that if we find a bug in one solver, instead of adding a test to that particular repository, we add it here so that all solvers can benefit.

How to test a solver The skeleton below can be used for the wrapper test file of a solver named FooBar.

```
# ============================ /test/MOI_wrapper.jl ============================
module TestFooBar

import FooBar
using MathOptInterface
using Test

const MOI = MathOptInterface

const OPTIMIZER = MOI.instantiate(
    MOI.OptimizerWithAttributes(FooBar.Optimizer, MOI.Silent() => true),
)

const BRIDGED = MOI.instantiate(
    MOI.OptimizerWithAttributes(FooBar.Optimizer, MOI.Silent() => true),
    with_bridge_type = Float64,
)

# See the docstring of MOI.Test.Config for other arguments.
const CONFIG = MOI.Test.Config(
    # Modify tolerances as necessary.
    atol = 1e-6,
    rtol = 1e-6,
    # Use MOI.LOCALLY_SOLVED for local solvers.
    optimal_status = MOI.OPTIMAL,
    # Pass attributes or MOI functions to `exclude` to skip tests that
    # rely on this functionality.
    exclude = Any[MOI.VariableName, MOI.delete],
)

"""
    runtests()

This function runs all functions in the this Module starting with `test_`.
```

```
"""
function runtests()
    for name in names(@__MODULE__; all = true)
        if startswith("$(name)", "test_")
            @testset "$(name)" begin
                getfield(@__MODULE__, name)()
            end
        end
    end
end

"""
    test_runtests()

This function runs all the tests in MathOptInterface.Test.

Pass arguments to `exclude` to skip tests for functionality that is not
implemented or that your solver doesn't support.
"""
function test_runtests()
    MOI.Test.runtests(
        BRIDGED,
        CONFIG,
        exclude = [
            "test_attribute_NumberOfThreads",
            "test_quadratic_",
        ]
    )
    return
end

"""
    test_SolverName()

You can also write new tests for solver-specific functionality. Write each new
test as a function with a name beginning with `test_`.
"""
function test_SolverName()
    @test MOI.get(FooBar.Optimizer(), MOI.SolverName()) == "FooBar"
    return
end

end # module TestFooBar

# This line at tne end of the file runs all the tests!
TestFooBar.runtests()
```

Then modify your `runtests.jl` file to include the `MOI_wrapper.jl` file:

```
# ========================== /test/runtests.jl ============================

using Test

@testset "MOI" begin
    include("test/MOI_wrapper.jl")
end
```

> **Info**
>
> The optimizer `BRIDGED` constructed with `instantiate` automatically bridges constraints that are not supported by `OPTIMIZER` using the bridges listed in Bridges. It is recommended for an implementation of MOI to only support constraints that are natively supported by the solver and let bridges transform the constraint to the appropriate form. For this reason it is expected that tests may not pass if `OPTIMIZER` is used instead of `BRIDGED`.

How to debug a failing test When writing a solver, it's likely that you will initially fail many tests! Some failures will be bugs, but other failures you may choose to exclude.

There are two ways to exclude tests:

- Exclude tests whose names contain a string using:

  ```
  MOI.Test.runtests(
      model,
      config;
      exclude = String["test_to_exclude", "test_conic_"],
  )
  ```

  This will exclude tests whose name contains either of the two strings provided.

- Exclude tests which rely on specific functionality using:

  ```
  MOI.Test.Config(exclude = Any[MOI.VariableName, MOI.optimize!])
  ```

  This will exclude tests which use the `MOI.VariableName` attribute, or which call `MOI.optimize!`.

Each test that fails can be independently called as:

```
model = FooBar.Optimizer()
config = MOI.Test.Config()
MOI.empty!(model)
MOI.Test.test_category_name_that_failed(model, config)
```

You can look-up the source code of the test that failed by searching for it in the `src/Test/test_category.jl` file.

> **Tip**
>
> Each test function also has a docstring that explains what the test is for. Use `? MOI.Test.test_category_name_that_faile` from the REPL to read it.

How to add a test To detect bugs in solvers, we add new tests to `MOI.Test`.

As an example, ECOS errored calling `optimize!` twice in a row. (See ECOS.jl PR #72.) We could add a test to ECOS.jl, but that would only stop us from re-introducing the bug to ECOS.jl in the future, but it would not catch other solvers in the ecosystem with the same bug! Instead, if we add a test to `MOI.Test`, then all solvers will also check that they handle a double optimize call!

For this test, we care about correctness, rather than performance. therefore, we don't expect solvers to efficiently decide that they have already solved the problem, only that calling `optimize!` twice doesn't throw an error or give the wrong answer.

Step 1

Install the `MathOptInterface` julia package in dev mode (ref):

```
julia> ]
(@v1.6) pkg> dev MathOptInterface
```

Step 2

From here on, proceed with making the following changes in the `~/.julia/dev/MathOptInterface` folder (or equivalent dev path on your machine).

Step 3

Since the double-optimize error involves solving an optimization problem, add a new test to src/Test/UnitTests/solve.jl.

The test should be something like

```
"""
    test_unit_optimize!_twice(model::MOI.ModelLike, config::Config)

Test that calling `MOI.optimize!` twice does not error.

This problem was first detected in ECOS.jl PR#72:
https://github.com/jump-dev/ECOS.jl/pull/72
"""
function test_unit_optimize!_twice(
    model::MOI.ModelLike,
    config::Config{T},
) where {T}
    # Use the `@requires` macro to check conditions that the test function
    # requires in order to run. Models failing this `@requires` check will
    # silently skip the test.
    @requires MOI.supports_constraint(
        model,
        MOI.VariableIndex,
        MOI.GreaterThan{Float64},
    )
    @requires _supports(config, MOI.optimize!)
    # If needed, you can test that the model is empty at the start of the test.
    # You can assume that this will be the case for tests run via `runtests`.
    # User's calling tests individually need to call `MOI.empty!` themselves.
    @test MOI.is_empty(model)
    # Create a simple model. Try to make this as simple as possible so that the
    # majority of solvers can run the test.
    x = MOI.add_variable(model)
    MOI.add_constraint(model, x, MOI.GreaterThan(one(T)))
    MOI.set(model, MOI.ObjectiveSense(), MOI.MIN_SENSE)
    MOI.set(
        model,
        MOI.ObjectiveFunction{MOI.VariableIndex}(),
        x,
    )
    # The main component of the test: does calling `optimize!` twice error?
    MOI.optimize!(model)
    MOI.optimize!(model)
    # Check we have a solution.
    @test MOI.get(model, MOI.TerminationStatus()) == MOI.OPTIMAL
    # There is a three-argument version of `Base.isapprox` for checking
    # approximate equality based on the tolerances defined in `config`:
```

```
    @test isapprox(MOI.get(model, MOI.VariablePrimal(), x), one(T), config)
    # For code-style, these tests should always `return` `nothing`.
    return
end
```

> **Info**
>
> Make sure the function is agnoistic to the number type T! Don't assume it is a `Float64` capable solver!

We also need to write a test for the test. Place this function immediately below the test you just wrote in the same file:

```
function setup_test(
    ::typeof(test_unit_optimize!_twice),
    model::MOI.Utilities.MockOptimizer,
    ::Config,
)
    MOI.Utilities.set_mock_optimize!(
        model,
        (mock::MOI.Utilities.MockOptimizer) -> MOIU.mock_optimize!(
            mock,
            MOI.OPTIMAL,
            (MOI.FEASIBLE_POINT, [1.0]),
        ),
    )
    return
end
```

Step 6

Commit the changes to git from `~/.julia/dev/MathOptInterface` and submit the PR for review.

> **Tip**
>
> If you need help writing a test, open an issue on GitHub, or ask the Developer Chatroom

API Reference

The Test submodule

Functions to help test implementations of MOI. See The Test submodule for more details.

`MathOptInterface.Test.Config` – Type.

```
Config(
    ::Type{T} = Float64;
    atol::Real = Base.rtoldefault(T),
    rtol::Real = Base.rtoldefault(T),
    optimal_status::MOI.TerminationStatusCode = MOI.OPTIMAL,
    exclude::Vector{Any} = Any[],
) where {T}
```

Return an object that is used to configure various tests.

Configuration arguments

- `atol::Real = Base.rtoldefault(T)`: Control the absolute tolerance used when comparing solutions.
- `rtol::Real = Base.rtoldefault(T)`: Control the relative tolerance used when comparing solutions.
- `optimal_status = MOI.OPTIMAL`: Set to `MOI.LOCALLY_SOLVED` if the solver cannot prove global optimality.
- `exclude = Vector{Any}`: Pass attributes or functions to `exclude` to skip parts of tests that require certain functionality. Common arguments include:
  - `MOI.delete` to skip deletion-related tests
  - `MOI.optimize!` to skip optimize-related tests
  - `MOI.ConstraintDual` to skip dual-related tests
  - `MOI.VariableName` to skip setting variable names
  - `MOI.ConstraintName` to skip setting constraint names

Examples

For a nonlinear solver that finds local optima and does not support finding dual variables or constraint names:

```
Config(
    Float64;
    optimal_status = MOI.LOCALLY_SOLVED,
    exclude = Any[
        MOI.ConstraintDual,
        MOI.VariableName,
        MOI.ConstraintName,
        MOI.delete,
    ],
)
```

`MathOptInterface.Test.runtests` – Function.

```
runtests(
    model::MOI.ModelLike,
    config::Config;
    include::Vector{String} = String[],
    exclude::Vector{String} = String[],
    warn_unsupported::Bool = false,
)
```

Run all tests in `MathOptInterface.Test` on `model`.

Configuration arguments

- `config` is a `Test.Config` object that can be used to modify the behavior of tests.
- If `include` is not empty, only run tests that contain an element from `include` in their name.
- If `exclude` is not empty, skip tests that contain an element from `exclude` in their name.
- `exclude` takes priority over `include`.
- If `warn_unsupported` is `false`, `runtests` will silently skip tests that fail with `UnsupportedConstraint` or `UnsupportedAttribute`. When `warn_unsupported` is `true`, a warning will be printed. For most cases the default behavior (`false`) is what you want, since these tests likely test functionality that is not supported by `model`. However, it can be useful to run `warn_unsupported = true` to check you are not skipping tests due to a missing `supports_constraint` method or equivalent.

See also: setup_test.

Example

```
config = MathOptInterface.Test.Config()
MathOptInterface.Test.runtests(
    model,
    config;
    include = ["test_linear_"],
    exclude = ["VariablePrimalStart"],
    warn_unsupported = true,
)
```

MathOptInterface.Test.setup_test – Function.

```
setup_test(::typeof(f), model::MOI.ModelLike, config::Config)
```

Overload this method to modify `model` before running the test function `f` on `model` with `config`. You can also modify the fields in `config` (e.g., to loosen the default tolerances).

This function should either return `nothing`, or return a function which, when called with zero arguments, undoes the setup to return the model to its previous state. You do not need to undo any modifications to `config`.

This function is most useful when writing new tests of the tests for MOI, but it can also be used to set test-specific tolerances, etc.

See also: runtests

Example

```
function MOI.Test.setup_test(
    ::typeof(MOI.Test.test_linear_VariablePrimalStart_partial),
    mock::MOIU.MockOptimizer,
    ::MOI.Test.Config,
)
    MOIU.set_mock_optimize!(
        mock,
        (mock::MOIU.MockOptimizer) -> MOIU.mock_optimize!(mock, [1.0, 0.0]),
    )
    mock.eval_variable_constraint_dual = false

    function reset_function()
        mock.eval_variable_constraint_dual = true
        return
    end
    return reset_function
end
```

MathOptInterface.Test.@requires – Macro.

```
@requires(x)
```

Check that the condition `x` is `true`. Otherwise, throw an RequirementUnmet error to indicate that the model does not support something required by the test function.

Examples

```
@requires MOI.supports(model, MOI.Silent())
@test MOI.get(model, MOI.Silent())
```

`MathOptInterface.Test.RequirementUnmet` – Type.

```
RequirementUnmet(msg::String) <: Exception
```

An error for throwing in tests to indicate that the model does not support some requirement expected by the test function.

Chapter 41

Release notes

41.1 Release notes

v0.10.4 (October 26, 2021)

For a detailed list of the closed issues and pull requests from this release, see the tag notes.

New features

- Add `SolverVersion` attribute
- Add new tests:
  - `test_solve_conflict_zeroone_ii`
  - `test_nonlinear_objective`
- `Utilities.VariablesContainer` now supports `ConstraintFunction` and `ConstraintSet`
- The documentation is now available as a PDF

Maintenance

- Update to MutableArithmetics 0.3
- Various improvements to the documentation

v0.10.3 (September 18, 2021)

For a detailed list of the closed issues and pull requests from this release, see the tag notes.

- Fix bug which prevented callbacks from working through a CachingOptimizer
- Fix bug in `Test` submodule

v0.10.2 (September 16, 2021)

For a detailed list of the closed issues and pull requests from this release, see the tag notes.

- Updated MathOptFormat to v1.0

- Updated JSONSchema to v1.0
- Added `Utilities.set_with_dimension`
- Added two-argument `optimize!(::AbstractOptimizer, ::ModelLike)`
- The experimental feature `copy_to_and_optimize!` has been removed
- Det bridges now support getting `ConstraintFunction` and `ConstraintSet`
- Various minor bug fixes identified by improved testing

v0.10.1 (September 8, 2021)

For a detailed list of the closed issues and pull requests from this release, see the tag notes.

- Various fixes to `MOI.Test`

v0.10.0 (September 6, 2021)

MOI v0.10 is a significant breaking release. There are a large number of user-visible breaking changes and code refactors, as well as a substantial number of new features.

For a detailed list of the closed issues and pull requests from this release, see the tag notes.

Breaking changes in MOI

- `SingleVariable` has been removed; use `VariableIndex` instead
- `SingleVariableConstraintNameError` has been renamed to `VariableIndexConstraintNameError`
- `SettingSingleVariableFunctionNotAllowed` has been renamed to `SettingVariableIndexFunctionNotAllowed`
- `VariableIndex` constraints should not support `ConstraintName`
- `VariableIndex` constraints should not support `ConstraintBasisStatus`; implement `VariableBasisStatus` instead
- `ListOfConstraints` has been renamed to `ListOfConstraintTypesPresent`
- `ListOfConstraintTypesPresent` should now return `Tuple{Type,Type}` instead of `Tuple{DataType,DataType}`
- `SolveTime` has been renamed to `SolveTimeSec`
- `IndicatorSet` has been renamed to `Indicator`
- `RawParameter` has been renamed to `RawOptimizerAttribute` and now takes `String` instead of `Any` as the only argument
- The `.N` field in result attributes has been renamed to `.result_index`
- The `.variable_index` field in `ScalarAffineTerm` has been renamed to `.variable`
- The `.variable_index_1` field in `ScalarQuadraticTerm` has been renamed to `.variable_1`
- The `.variable_index_2` field in `ScalarQuadraticTerm` has been renamed to `.variable_2`
- The order of `affine_terms` and `quadratic_terms` in `ScalarQuadraticFunction` and `VectorQuadraticFunction` have been reversed. Both functions now accept quadratic, affine, and constant terms in that order.

- The `index_value` function has been removed. Use `.value` instead.
- `isapprox` has been removed for `SOS1` and `SOS2`.
- The `dimension` argument to `Complements(dimension::Int)` should now be the length of the corresponding function, instead of half the length. An `ArgumentError` is thrown if `dimension` is not even.
- `copy_to` no longer takes keyword arguments:
  - `copy_names`: now copy names if they are supported by the destination solver
  - `filter_constraints`: use `Utilities.ModelFilter` instead
  - `warn_attributes`: never warn about optimizer attributes

Breaking changes in `Bridges`

- `Constraint.RSOCBridge` has been renamed to `Constraint.RSOCtoSOCBridge`
- `Constraint.SOCRBridge` has been renamed to `Constraint.SOCtoRSOCBridge`
- Bridges now return vectors that can be modified by the user. Previously, some bridges returned views instead of copies.
- `Bridges.IndexInVector` has been unified into a single type. Previously, there was a different type for each submodule within `Bridges`
- The signature of indicator bridges has been fixed. Use `MOI.Bridges.Constraint.IndicatortoSOS1{Float64}(model)`.

Breaking changes in `FileFormats`

- `FileFormats.MOF.Model` no longer accepts `validate` argument. Use the JSONSchema package to validate the MOF file. See the documentation for more information.

Breaking changes in `Utilities`

- The datastructure of `Utilities.Model` (and models created with `Utilities.@model`) has been significantly refactored in a breaking way. This includes the way that objecive functions and variable-related information is stored.
- `Utilities.supports_default_copy` has been renamed to `supports_incremental_interface`
- `Utilities.automatic_copy_to` has been renamed to `Utilities.default_copy_to`
- The allocate-load API has been removed
- `CachingOptimizers` are now initialized as `EMPTY_OPTIMIZER` instead of `ATTACHED_OPTIMIZER`. If your code relies on the optimizer being attached, call `MOIU.attach_optimizer(model)` after creation.
- The field names of `Utilities.IndexMap` have been renamed to `var_map` and `con_map`. Accessing these fields directly is considered a private detail that may change. Use the public `getindex` and `setindex!` API instead.
- The size argument to `Utilities.CleverDicts.CleverDict(::Integer)` has been removed.
- The size argument to `Utilities.IndexMap(::Integer)` has been removed.
- `Utilities.DoubleDicts` have been significantly refactored. Consult the source code for details.
- `Utilities.test_models_equal` has been moved to `MOI.Test`

Breaking changes in `Test`

- `MOI.Test` has been renamed to `MOI.DeprecatedTest`
- An entirely new `MOI.Test` submodule has been written. See the documentation for details. The new `MOI.Test` submodule may find many bugs in the implementations of existing solvers that were previously untested.

Other changes:

- `attribute_value_type` has been added
- `copy_to_and_optimize!` has been added
- `VariableBasisStatus` has been added
- `print(model)` now prints a human-readable description of the model
- Various improvements to the `FileFormats` submodule
  - `FileFormats.CBF` was refactored and received bugfixes
  - Support for MathOptFormat v0.6 was added in `FileFormats.MOF`
  - `FileFormats.MPS` has had bugfixes and support for more features such as `OBJSENSE` and objective constants.
  - `FileFormats.NL` has been added to support nonlinear files
- Improved type inference throughout to reduce latency

Updating

A helpful script when updating is:

```
for (root, dirs, files) in walkdir(".")
    for file in files
        if !endswith(file, ".jl")
            continue
        end
        path = joinpath(root, file)
        s = read(path, String)
        for pair in [
            ".variable_index" => ".variable",
            "RawParameter" => "RawOptimizerAttribute",
            "ListOfConstraints" => "ListOfConstraintTypesPresent",
            "TestConfig" => "Config",
            "attr.N" => "attr.result_index",
            "SolveTime" => "SolveTimeSec",
            "DataType" => "Type",
            "Utilities.supports_default_copy_to" =>
                "supports_incremental_interface",
            "SingleVariableConstraintNameError" =>
                "VariableIndexConstraintNameError",
            "SettingSingleVariableFunctionNotAllowed" =>
                "SettingVariableIndexFunctionNotAllowed",
            "automatic_copy_to" => "default_copy_to",
```

```
        ]
            s = replace(s, pair)
        end
        write(path, s)
    end
end
```

v0.9.22 (May 22, 2021)

This release contains backports from the ongoing development of the v0.10 release. For a detailed list of the closed issues and pull requests from this release, see the tag notes.

- Improved type inference in `Utilities`, `Bridges` and `FileFormats` submodules to reduce latency.
- Improved performance of `Utilities.is_canonical`.
- Fixed `Utilities.pass_nonvariable_constraints` with bridged variables.
- Fixed performance regression of `Utilities.Model`.
- Fixed ordering of objective setting in parser.

v0.9.21 (April 23, 2021)

For a detailed list of the closed issues and pull requests from this release, see the tag notes.

- Added `supports_shift_constant`.
- Improve performance of bridging quadratic constraints.
- Add precompilation statements.
- Large improvements to the documentation.
- Fix a variety of inference issues, benefiting precompilation and reducing initial latency.
- `RawParameters` are now ignored when resetting a `CachingOptimizer`. Previously, changing the underlying optimizer after `RawParameters` were set would throw an errror.
- `Utilities.AbstractModel` is being refactored. This may break users interacting with private fields of a model generated using `@model`.

v0.9.20 (February 20, 2021)

For a detailed list of the closed issues and pull requests from this release, see the tag notes.

- Improved performance of `Utilities.ScalarFunctionIterator`
- Added support for `compute_conflict` to MOI layers
- Added test with zero off-diagonal quadratic term in objective
- Fixed double deletion of nested bridged `SingleVariable`/`VectorOfVariables` constraints
- Fixed modification of un-set objective
- Fixed function modification with duplicate terms

- Made unit tests abort without failing if the problem class is not supported
- Formatted code with JuliaFormatter
- Clarified `BasisStatusCode`'s docstring

v0.9.19 (December 1, 2020)

For a detailed list of the closed issues and pull requests from this release, see the tag notes.

- Added `CallbackNodeStatus` attribute
- Added bridge from `GreaterThan` or `LessThan` to `Interval`
- Added tests for infeasibility certificates and double optimize
- Fixed support for Julia v1.6
- Re-organized MOI docs and added documentation for adding a test

v0.9.18 (November 3, 2020)

For a detailed list of the closed issues and pull requests from this release, see the tag notes.

- Various improvements for working with complex numbers
- Added `GeoMeantoRelEntrBridge` to bridge a geomean constraint to a relative entropy constraint

v0.9.17 (September 21, 2020)

For a detailed list of the closed issues and pull requests from this release, see the tag notes.

- Fixed `CleverDict` with variable of negative index value
- Implement `supports_add_constrained_variable` for `MockOptimizer`

v0.9.16 (September 17, 2020)

For a detailed list of the closed issues and pull requests from this release, see the tag notes.

- Various fixes:
  - 32-bit support
  - `CleverDict` with abstract value type
  - Checks in test suite

v0.9.15 (September 14, 2020)

For a detailed list of the closed issues and pull requests from this release, see the tag notes.

A summary of changes are as follows:

- Bridges improvements:
  - (R)SOCtoNonConvexQuad bridge
  - ZeroOne bridge
  - Use `supports_add_constrained_variable` in `LazyBridgeOptimizer`
  - Exposed `VariableBridgeCost` and `ConstraintBridgeCost` attributes
  - Prioritize constraining variables on creation according to these costs
  - Refactor bridge debugging
- Large performance improvements across all submodules
- Lots of documentation improvements
- FileFormats improvements:
  - Update MathOptFormat to v0.5
  - Fix supported objectives in `FileFormats`
- Testing improvements:
  - Add name option for `basic_constraint_test`
- Bug fixes and missing methods
  - Add `length` for iterators
  - Fix bug with duplicate terms
  - Fix order of `LinearOfConstraintIndices`

v0.9.14 (May 30, 2020)

- Add a solver-independent interface for accessing the set of conflicting constraints an Irreducible Inconsistent Subsystem (#1056).
- Bump JSONSchema dependency from v0.2 to v0.3 (#1090).
- Documentation improvements:
  - Fix typos (#1054, #1060, #1061, #1064, #1069, #1070).
  - Remove the outdated recommendation for a package implementing MOI for a solver XXX to be called `MathOptInterfaceXXX` (#1087).
- Utilities improvements:
  - Fix `is_canonical` for quadratic functions (#1081, #1089).
  - Implement `add_constrained_variable[s]` for `CachingOptimizer` so that it is added as constrained variables to the underlying optimizer (#1084).

  - Add support for custom objective functions for `UniversalFallback` (#1086).
  - Deterministic ordering of constraints in `UniversalFallback` (#1088).

- Testing improvements:
  - Add `NormOneCone`/`NormInfinityCone` tests (#1045).

- Bridges improvements:
  - Add bridges from `Semiinteger` and `Semicontinuous` (#1059).
  - Implement getting `ConstraintSet` for `Variable.FlipSignBridge` (#1066).
  - Fix setting `ConstraintFunction` for `Constraint.ScalarizeBridge` (#1093).
  - Fix `NormOne`/`NormInf` bridges with nonzero constants (#1045).
  - Fix StackOverflow in debug (#1063).

- FileFormats improvements:

SDPA Implement the extension for integer variables (#1079).
SDPA Ignore comments aftere `m` and `nblocks` and detect `dat-s` extension (#1077).
SDPA No scaling of off-diagonal coefficient (#1076).
SDPA Add missing negation of constant (#1075).

v0.9.13 (March 24, 2020)

- Added tests for `Semicontinuous` and `Semiinteger` variables (#1033).
- Added tests for using `ExprGraphs` from NLP evaluators (#1043).
- Update version compatibilities of depedencies (#1034, #1051, #1052).
- Fixed typos in documentation (#1044).

v0.9.12 (February 28, 2020)

- Fixed writing `NLPBlock` in MathOptFormat (#1037).
- Fixed `MockOptimizer` for result attributes with non-one result index (#1039).
- Updated test template with `instantiate` (#1032).

v0.9.11 (February 21, 2020)

- Add an option for the model created by `Utilities.@model` to be a subtype of `AbstractOptimizer` (#1031).
- Described dual cone in docstrings of `GeoMeanCone` and `RelativeEntropyCone` (#1018, #1028).
- Fixed typos in documentation (#1022, #1024).
- Fixed warning of unsupported attribute (#1027).
- Added more rootdet/logdet conic tests (#1026).
- Implemented `ConstraintDual` for `Constraint.GeoMeanBridge`, `Constraint.RootDetBridge` and `Constraint.LogDetBri` and test duals in tests with `GeoMeanCone` and `RootDetConeTriangle` and `LogDetConeTriangle` cones (#1025, #1026).

v0.9.10 (January 31, 2020)

- Added `OptimizerWithAttributes` grouping an optimizer constructor and a list of optimizer attributes (#1008).
- Added `RelativeEntropyCone` with corresponding bridge into exponential cone constraints (#993).
- Added `NormSpectralCone` and `NormNuclearCone` with corresponding bridges into positive semidefinite constraints (#976).
- Added `supports_constrained_variable(s)` (#1004).
- Added `dual_set_type` (#1002).
- Added tests for vector specialized version of `delete` (#989, #1011).
- Added PSD3 test (#1007).
- Clarified dual solution of `Tests.pow1v` and `Tests.pow1f` (#1013).
- Added support for `EqualTo` and `Zero` in `Bridges.Constraint.SplitIntervalBridge` (#1005).
- Fixed `Utilities.vectorize` for empty vector (#1003).
- Fixed free variables in LP writer (#1006).

v0.9.9 (December 29, 2019)

- Incorporated MathOptFormat.jl as the FileFormats submodule. FileFormats provides readers and writers for a number of standard file formats and MOF, a file format specialized for MOI (#969).
- Improved performance of deletion of vector of variables in `MOI.Utilities.Model` (#983).
- Updated to MutableArithmetics v0.2 (#981).
- Added `MutableArithmetics.promote_operation` allocation tests (#975).
- Fixed inference issue on Julia v1.1 (#982).

v0.9.8 (December 19, 2019)

- Implemented MutableArithmetics API (#924).
- Fixed callbacks with `CachingOptimizer` (#959).
- Fixed `MOI.dimension` for `MOI.Complements` (#948).
- Added fallback for `add_variables` (#972).
- Added `is_diagonal_vectorized_index` utility (#965).
- Improved linear constraints display in manual (#963, #964).
- Bridges improvements:
  - Added `IndicatorSet` to `SOS1` bridge (#877).
  - Added support for starting values for `Variable.VectorizeBridge` (#944).
  - Fixed `MOI.add_constraints` with non-bridged variable constraint on bridged variable (#951).

  - Fixed corner cases and docstring of geomean bridge (#961, #962, #966).
  - Fixed choice between variable or constraint bridges for constrained variables (#973).
  - Improve performance of bridge shortest path (#945, #946, #956).
  - Added docstring for `test_delete_bridge` (#954).
  - Added Variable bridge tests (#952).

v0.9.7 (October 30, 2019)

- Implemented `_result_index_field` for `NLPBlockDual` (#934).
- Fixed copy of model with starting values for vector constraints (#941).
- Bridges improvements:
  - Improved performance of `add_bridge` and added `has_bridge` (#935).
  - Added `AbstractSetMapBridge` for bridges between sets `S1`, `S2` such that there is a linear map `A` such that `A*S1 = S2` (#933).
  - Added support for starting values for `FlipSignBridge`, `VectorizeBridge`, `ScalarizeBridge`, `SlackBridge`, `SplitIntervalBridge`, `RSOCBridge`, `SOCRBridge` `NormInfinityBridge`, `SOCtoPSDBridge` and `RSOCtoPSDBridge` (#933, #936, #937, #938, #939).

v0.9.6 (October 25, 2019)

- Added complementarity constraints (#913).
- Allowed `ModelLike` objects as value of attributes (#928).
- Testing improvements:
  - Added `dual_objective_value` option to `MOI.Test.TestConfig` (#922).
  - Added `InvalidIndex` tests in `basic_constraint_tests` (#921).
  - Added tests for the constant term in indicator constraint (#929).
- Bridges improvements:
  - Added support for starting values for functionize bridges (#923).
  - Added variable indices context to variable bridges (#920).
  - Fixed a typo in printing o `debug_supports` (#927).

v0.9.5 (October 9, 2019)

- Clarified `PrimalStatus`/`DualStatus` to be `NO_SOLUTION` if `result_index` is out of bounds (#912).
- Added tolerance for checks and use `ResultCount + 1` for the `result_index` in `MOI.Test.solve_result_status` (#910, #917).
- Use `0.5` instead of `2.0` for power in PowerCone in `basic_constraint_test` (#916).
- Bridges improvements:
  - Added debug utilities for unsupported variable/constraint/objective (#861).
  - Fixed deletion of variables in bridged `VectorOfVariables` constraints (#909).
  - Fixed `result_index` with objective bridges (#911).

v0.9.4 (October 2, 2019)

- Added solver-independent MIP callbacks (#782).
- Implements submit for `Utilities.CachingOptimizer` and `Bridges.AbstractBridgeOptimizer` (#906).
- Added tests for result count of solution attributes (#901, #904).
- Added `NumberOfThreads` attribute (#892).
- Added `Utilities.get_bounds` to get the bounds on a variable (#890).
- Added a note on duplicate coefficients in documentation (#581).
- Added result index in `ConstraintBasisStatus` (#898).
- Added extension dictionary to `Utilities.Model` (#884, #895).
- Fixed deletion of constrained variables for CachingOptimizer (#905).
- Implemented `Utilities.shift_constraint` for `Test.UnknownScalarSet` (#896).
- Bridges improvements:
  - Added Variable.RSOCtoSOCBridge (#907).
  - Implemented `MOI.get` for `ConstraintFunction`/`ConstraintSet` for `Bridges.Constraint.SquareBridge` (#899).

v0.9.3 (September 20, 2019)

- Fixed ambiguity detected in Julia v1.3 (#891, #893).
- Fixed missing sets from `ListOfSupportedConstraints` (#880).
- Fixed copy of `VectorOfVariables` constraints with duplicate indices (#886).
- Added extension dictionary to MOIU.Model (#884).
- Implemented `MOI.get` for function and set for `GeoMeanBridge` (#888).
- Updated documentation for SingleVariable indices and bridges (#885).
- Testing improvements:
  - Added more comprehensive tests for names (#882).
  - Added tests for `SingleVariable` duals (#883).
  - Added tests for `DualExponentialCone` and `DualPowerCone` (#873).
- Improvements for arbitary coefficient type:
  - Fixed == for sets with mutable fields (#887).
  - Removed some `Float64` assumptions in bridges (#878).
  - Automatic selection of `Constraint.[Scalar|Vector]FunctionizeBridge` (#889).

v0.9.2 (September 5, 2019)

- Implemented model printing for `MOI.ModelLike` and specialized it for models defined in MOI (864).
- Generalized `contlinear` tests for arbitary coefficient type (#855).
- Fixed `supports_constraint` for `Semiinteger` and `Semicontinuous` and `supports` for `ObjectiveFunction` (#859).
- Fixed Allocate-Load copy for single variable constraints (#856).
- Bridges improvements:
  - Add objective bridges (#789).
  - Fixed `Variable.RSOCtoPSDBridge` for dimension 2 (#869).
  - Added `Variable.SOCtoRSOCBridge` (#865).
  - Added `Constraint.SOCRBridge` and disable `MOI.Bridges.Constraint.SOCtoPSDBridge` (#751).
  - Fixed `added_constraint_types` for `Contraint.LogDetBridge` and `Constraint.RootDetBridge` (#870).

v0.9.1 (August 22, 2019)

- Fix support for Julia v1.2 (#834).
- L1 and L∞ norm epigraph cones and corresponding bridges to LP were added (#818).
- Added tests to `MOI.Test.nametest` (#833).
- Fix `MOI.Test.soc3test` for solvers not supporting infeasibility certificates (#839).
- Implements `operate` for operators `*` and `/` between vector function and constant (#837).
- Implements `show` for `MOI.Utilities.IndexMap` (#847).
- Fix corner cases for mapping of variables in `MOI.Utilities.CachingOptimizer` and substitution of variables in `MOI.Bridges.AbstractBridgeOptimizer` (#848).
- Fix transformation of constant terms for `MOI.Bridges.Constraint.SOCtoPSDBridge` and `MOI.Bridges.Constraint.RSOCt` (#840).

v0.9.0 (August 13, 2019)

- Support for Julia v0.6 and v0.7 was dropped (#714, #717).
- A `MOI.Utilities.Model` implementation of `ModelLike`, this should replace most use cases of `MOI.Utilities.@model` (#781).
- `add_constrained_variable` and `add_constrained_variables` were added (#759).
- Support for indicator constraints was added (#709, #712).
- `DualObjectiveValue` attribute was added (#473).
- `RawParameter` attribute was added (#733).
- A `dual_set` function was added (#804).

- A `Benchmarks` submodule was added to facilitate solver benchmarking (#769).
- A `submit` function was added, this may for intance allow the user to submit solutions or cuts to the solver from a callback (#775).
- The field of `ObjectiveValue` was renamed to `result_index` (#729).
- The `_constant` and `Utilities.getconstant` function were renamed to `constant`
- `REDUCTION_CERTIFICATE` result status was added (#734).
- Abstract matrix sets were added (#731).
- Testing improvements:
  - The testing guideline was updated (#728).
  - Quadratic tests were added (#697).
  - Unit tests for `RawStatusString`, `SolveTime`, `Silent` and `SolverName` were added (#726, #741).
  - A rotated second-order cone test was added (#759).
  - A power cone test was added (#768).
  - Tests for `ZeroOne` variables with variable bounds were added (#772).
  - An unbounded test was added (#773).
  - Existing tests had a few updates (#702, #703, #763).
- Documentation improvements:
  - Added a section on `CachingOptimizer` (#777).
  - Added a section on `UniversalFallback`, `Model` and `@model` (#762).
  - Transition the knapsack example to a doctest with MockOptimizer (#786).
- Utilities improvements:
  - A `CleverDict` utility was added for a vector that automatically transform into a dictionary once a first index is removed (#767).
  - The `Utilities.constant` function was renamed to `Utilities.constant_vector` (#740).
  - Implement optimizer attributes for CachingOptimizer (#745).
  - Rename `Utilities.add_scalar_constraint` to `Utilities.normalize_and_add_constraint` (#801).
  - `operate` with `vcat`, `SingleVariable` and `VectorOfVariables` now returns a `VectorOfVariables` (#616).
  - Fix a type piracy of `operate` (#784).
  - The `load_constraint` fallback signature was fixed (#760).
  - The `set_dot` function was extended to work with sparse arrays (#805).
- Bridges improvements:
  - The bridges no longer store the constraint function and set before it is briged, the bridges now have to implement `ConstraintFunction` and `ConstraintSet` if the user wants to recover them. As a consequence, the `@bridge` macro was removed (#722).
  - Bridge are now instantiated with a `bridge_constraint` function instead of using a constructor (#730).

  - Fix constraint attributes for bridges (#699).
  - Constraint bridges were moved to the `Bridges/Constraint` submodule so they should now inherit from `MOI.Bridges.Constraint.Abstract` and should implement `MOI.Bridges.Constraint.concrete_bridge_type` instead of `MOI.Bridges.concrete_bridge_type` (#756).
  - Variable bridges were added in (#759).
  - Various improvements (#746, #747).

v0.8.4 (March 13, 2019)

- Performance improvement in `default_copy_to` and bridge optimizer (#696).
- Add `Silent` and implement setting optimizer attributes in caching and mock optimizers (#695).
- Add functionize bridges (SingleVariable and VectorOfVariables) (#659).
- Minor typo fixes (#694).

v0.8.3 (March 6, 2019)

- Use zero constant in scalar constraint function of `MOI.Test.copytest` (#691).
- Fix variable deletion with `SingleVariable` objective function (#690).
- Fix `LazyBridgeOptimizer` with bridges that add no constraints (#689).
- Error message improvements (#673, #685, #686, #688).
- Documentation improvements (#682, #683, #687).
- Basis status:
  - Remove `VariableBasisStatus` (#679).
  - Test `ConstraintBasisStatus` and implement it in bridges (#678).
- Fix inference of `NumberOfVariables` and `NumberOfConstraints` (#677).
- Implement division between a quadratic function and a number (#675).

v0.8.2 (February 7, 2019)

- Add `RawStatusString` attribute (#629).
- Do not set names to the optimizer but only to the cache in `CachingOptimizer` (#638).
- Make scalar MOI functions act as scalars in broadcast (#646).
- Add function utilities:
  - Implement `Base.zero` (#634), `Base.iszero` (#643), add missing arithmetic operations (#644, #645) and fix division (#648).
  - Add a `vectorize` function that turns a vector of `ScalarAffineFunction` into a `VectorAffineFunction` (#642).
- Improve support for starting values:

  - Show a warning in copy when starting values are not supported instead of throwing an error (#630).
  - Fix `UniversalFallback` for getting an variable or constraint attribute set to no indices (#623).
  - Add a test in `contlineartest` with partially set `VariablePrimalStart`.
- Bridges improvements:
  - Fix `StackOverFlow` in `LazyBridgeOptimizer` when there is a cycle in the graph of bridges.
  - Add `Slack` bridges (#610, #650).
  - Add `FlipSign` bridges (#658).
- Add tests with duplicate coefficients in `ScalarAffineFunction` and `VectorAffineFunction` (#639).
- Use tolerance to compare `VariablePrimal` in `rotatedsoc1` test (#632).
- Use a zero constant in `ScalarAffineFunction` of constraints in `psdt2` (#622).

v0.8.1 (January 7, 2019)

- Adding an NLP objective now overrides any objective set using the `ObjectiveFunction` attribute (#619).
- Rename `fullbridgeoptimizer` into `full_bridge_optimizer` (#621).
- Allow custom constraint types with `full_bridge_optimizer` (#617).
- Add `Vectorize` bridge which transforms scalar linear constraints into vector linear constraints (#615).

v0.8.0 (December 18, 2018)

- Rename all enum values to follow the JuMP naming guidelines for constants, e.g., `Optimal` becomes `OPTIMAL`, and `DualInfeasible` becomes `DUAL_INFEASIBLE`.
- Rename CachingOptimizer methods for style compliance.
- Add an `MOI.TerminationStatusCode` called `ALMOST_DUAL_INFEASIBLE`.

v0.7.0 (December 13, 2018)

- Test that `MOI.TerminationStatus` is `MOI.OptimizeNotCalled` before `MOI.optimize!` is called.
- Check `supports_default_copy_to` in tests (#594).
- Key pieces of information like optimality, infeasibility, etc., are now reported through `TerminationStatusCode`. It is typically no longer necessary to check the result statuses in addition to the termination status.
- Add perspective dimension to log-det cone (#593).

v0.6.4 (November 27, 2018)

- Add `OptimizeNotCalled` termination status (#577) and improve documentation of other statuses (#575).
- Add a solver naming guideline (#578).
- Make `FeasibilitySense` the default `ObjectiveSense` (#579).
- Fix `Utilities.@model` and `Bridges.@bridge` macros for functions and sets defined outside MOI (#582).
- Document solver-specific attributes (#580) and implement them in `Utilities.CachingOptimizer` (#565).

v0.6.3 (November 16, 2018)

- Variables and constraints are now allowed to have duplicate names. An error is thrown only on lookup. This change breaks some existing tests. (#549)
- Attributes may now be partially set (some values could be `nothing`). (#563)
- Performance improvements in Utilities.Model (#549, #567, #568)
- Fix bug in QuadtoSOC (#558).
- New `supports_default_copy_to` method that optimizers should implement to control caching behavior.
- Documentation improvements.

v0.6.2 (October 26, 2018)

- Improve hygiene of `@model` macro (#544).
- Fix bug in copy tests (#543).
- Fix bug in UniversalFallback attribute getter (#540).
- Allow all correct solutions for `solve_blank_obj` unit test (#537).
- Add errors for Allocate-Load and bad constraints (#534).

performance Add specialized implementation of `hash` for `VariableIndex` (#533).

performance Construct the name to object dictionaries lazily in model (#535).

- Add the `QuadtoSOC` bridge which transforms `ScalarQuadraticFunction` constraints into `RotatedSecondOrderCone` (#483).

v0.6.1 (September 22, 2018)

- Enable `PositiveSemidefiniteConeSquare` set and quadratic functions in `MOIB.fullbridgeoptimizer` (#524).
- Add warning in the bridge between `PositiveSemidefiniteConeSquare` and `PositiveSemidefiniteConeTriangle` when the matrix is almost symmetric (#522).
- Modify `MOIT.copytest` to not add multiples constraints on the same variable (#521).
- Add missing keyword argument in one of `MOIU.add_scalar_constraint` methods (#520).

v0.6.0 (August 30, 2018)

- The `MOIU.@model` and `MOIB.@bridge` macros now support functions and sets defined in external modules. As a consequence, function and set names in the macro arguments need to be prefixed by module name.
- Rename functions according to the JuMP style guide:
  - `copy!` with keyword arguments `copynames` and `warnattributes` -> `copy_to` with keyword arguments `copy_names` and `warn_attributes`;

  - `set!` -> `set`;
  - `addvariable[s]!` -> `add_variable[s]`;
  - `supportsconstraint` -> `supports_constraint`;
  - `addconstraint[s]!` -> `add_constraint[s]`;
  - `isvalid` -> `is_valid`;
  - `isempty` -> `is_empty`;
  - `Base.delete!` -> `delete`;
  - `modify!` -> `modify`;
  - `transform!` -> `transform`;
  - `initialize!` -> `initialize`;
  - `write` -> `write_to_file`; and
  - `read!` -> `read_from_file`.

- Remove `free!` (use `Base.finalize` instead).
- Add the SquarePSD bridge which transforms `PositiveSemidefiniteConeTriangle` constraints into `PositiveSemidefinite`
- Add result fallback for `ConstraintDual` of variable-wise constraint, `ConstraintPrimal` and `ObjectiveValue`.
- Add tests for `ObjectiveBound`.
- Add test for empty rows in vector linear constraint.
- Rework errors: `CannotError` has been renamed `NotAllowedError` and the distinction between `UnsupportedError` and `NotAllowedError` is now about whether the element is not supported (i.e. it cannot be copied a model containing this element) or the operation is not allowed (either because it is not implemented, because it cannot be performemd in the current state of the model, because it cannot be performed for a specific index, ...)
- `canget` is removed. `NoSolution` is added as a result status to indicate that the solver does not have either a primal or dual solution available (See #479).

v0.5.0 (August 5, 2018)

- Fix names with CachingOptimizer.
- Cleanup thanks to @mohamed82008.
- Added a universal fallback for constraints.
- Fast utilities for function canonicalization thanks to @rdeits.
- Renamed `dimension` field to `side_dimension` in the context of matrix-like sets.
- New and improved tests for cases like duplicate terms and `ObjectiveBound`.
- Removed `cantransform`, `canaddconstraint`, `canaddvariable`, `canset`, `canmodify`, and `candelete` functions from the API. They are replaced by a new set of errors that are thrown: Subtypes of `UnsupportedError` indicate unsupported operations, while subtypes of `CannotError` indicate operations that cannot be performed in the current state.
- The API for `copy!` is updated to remove the CopyResult type.
- Updates for the new JuMP style guide.

v0.4.1 (June 28, 2018)

- Fixes vector function modification on 32 bits.
- Fixes Bellman-Ford algorithm for bridges.
- Added an NLP test with `FeasibilitySense`.
- Update modification documentation.

v0.4.0 (June 23, 2018)

- Helper constructors for `VectorAffineTerm` and `VectorQuadraticTerm`.
- Added `modify_lhs` to `TestConfig`.
- Additional unit tests for optimizers.
- Added a type parameter to `CachingOptimizer` for the `optimizer` field.
- New API for problem modification (#388)
- Tests pass without deprecation warnings on Julia 0.7.
- Small fixes and documentation updates.

v0.3.0 (May 25, 2018)

- Functions have been redefined to use arrays-of-structs instead of structs-of-arrays.
- Improvements to `MockOptimizer`.
- Significant changes to `Bridges`.
- New and improved unit tests.
- Fixes for Julia 0.7.

v0.2.0 (April 24, 2018)

- Improvements to and better coverage of `Tests`.
- Documentation fixes.
- `SolverName` attribute.
- Changes to the NLP interface (new definition of variable order and arrays of structs for bound pairs and sparsity patterns).
- Addition of NLP tests.
- Introduction of `UniversalFallback`.
- `copynames` keyword argument to `MOI.copy!`.
- Add Bridges submodule.

v0.1.0 (February 28, 2018)

- Initial public release.
- The framework for MOI was developed at the JuMP-dev workshop at MIT in June 2017 as a sorely needed replacement for MathProgBase.