JuMP

The JuMP core developers and contributors

May 25, 2023

Contents

Part I

Introduction

Chapter 1

Introduction

Welcome to the documentation for JuMP.

> **Note**
>
> This documentation is also available in PDF format: JuMP.pdf.

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 you 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 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 (preprint):

```
@article{Lubin2023,
    author = {Miles Lubin and Oscar Dowson and Joaquim Dias Garcia and Joey Huchette and Beno{\^i}t
    ↪  Legat and Juan Pablo Vielma},
    title = {JuMP 1.0: Recent improvements to a modeling language for mathematical optimization},
    journal = {Mathematical Programming Computation},
    year = {2023},
    note = {In press.}
}
```

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.

Figure 1.1: NumFOCUS logo

You can support JuMP by donating.

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.

1.6 License

JuMP is licensed under the MPL-2.0 software license. Consult the license and the Mozilla FAQ for more information. In addition, JuMP is typically used in conjunction with solver packages and extensions which have their own licences. Consult their package repositories for the specific licenses that apply.

Chapter 2

Should you use JuMP?

JuMP is an algebraic modeling language for mathematical optimization written in the Julia language.

This page explains when you should consider using JuMP, and importantly, when you should not use JuMP.

2.1   When should you use JuMP?

You should use JuMP if you have a constrained optimization problem for which you can formulate using the language of mathematical programming, that is:

- a set of decision variables

- a scalar- or vector-valued objective function

- a set of constraints.

Key reasons to use JuMP include:

- User friendliness

  - JuMP has syntax that mimics natural mathematical expressions. (See the section on algebraic modeling languages.)

- 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.

- Ease of embedding

  - JuMP itself is written purely in Julia. Solvers are the only binary 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.

  - Because it is embedded in a general-purpose programming language, JuMP makes it easy to solve optimization problems as part of a larger workflow, for example, 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 and functionality available in Julia.

- **–** JuMP is MPL licensed, meaning that it can be embedded in commercial software that complies with the terms of the license.

- 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.

- Access to advanced algorithmic techniques

    - **–** JuMP supports efficient in-memory re-solves of linear programs, which previously required using solver-specific or low-level C++ libraries.
    - **–** JuMP provides access to solver-independent and solver-dependent Callbacks.

2.2 When should you 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.

You want to optimize a complicated Julia function

Packages in Julia compose well. It's common for people to pick two unrelated packages and use them in conjunction to create novel behavior. JuMP isn't one of those packages.

If you want to optimize an ordinary differential equation from DifferentialEquations.jl or tune a neural network from Flux.jl, consider using other packages such as:

- Optim.jl

- Optimization.jl

- NLPModels.jl

- Nonconvex.jl

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 (for example, it is 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

- Optimization.jl

- NLopt.jl

Optimal control problems

JuMP supports formulating optimal control problems as large nonlinear programs (see, for example, Optimal control for a Space Shuttle reentry trajectory). However, the nonlinear interface has a number of limitations (for example, the need to write out the dynamics in algebraic form) that mean JuMP might not be the right tool for the job.

Alternatives to consider are:

- CasADi [MATLAB/Python], CasADi.jl

- InfiniteOpt.jl

- pyomo.DAE [Python]

Disciplined convex programming

JuMP does not support disciplined convex programming (DCP).

Alternatives to consider are:

- Convex.jl

- CVXPY [Python]

- YALMIP [MATLAB]

> **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.

Stochastic programming

JuMP requires deterministic input data.

If you have stochastic input data, consider using a JuMP extension such as:

- InfiniteOpt.jl

- StochasticPrograms.jl

- SDDP.jl

Chapter 3

Installation Guide

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

3.1 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.

Choosing a version

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 (for example, 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.

3.2 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")
```

3.3 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_attribute to set solver-specific options:

```
using JuMP
using Clp
model = Model(Clp.Optimizer)
set_attribute(model, "LogLevel" => 1)
set_attribute(model, "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.

3.4 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.

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
Clarabel.jl			Apache	LP, QP, SOCP, SDP
Clp	Clp.jl		EPL	LP
COPT	COPT.jl		Comm.	(MI)LP, SOCP, SDP
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
DAQP	DAQP.jl		MIT	(Mixed-binary) QP
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	(MI)LP, QP
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
Pajarito.jl			MPL-2	(MI)NLP, (MI)SOCP, (MI)SDP
Pavito.jl			MPL-2	(MI)NLP
Penbmi	Penopt.jl		Comm.	Bilinear SDP
ProxSDP.jl			MIT	LP, SOCP, SDP
RAPOSa	AmplNLWriter.jl	Manual	RAPOSa	(MI)NLP
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
StatusSwitchingQP.jl			MIT	LP, QP
Tulip.jl			MPL-2	LP

Where:

- LP = Linear programming

- QP = Quadratic programming

- SOCP = Second-order conic programming (including problems with convex quadratic constraints 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. Edit the installation.md file, and use the checklist Adding a new solver to the documentation when opening the pull request.

> **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.

3.5   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.

3.6   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.

3.7   NEOS-based solvers

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

3.8   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 4

Getting started

4.1 Introduction

The purpose of these "Getting started" tutorials is to teach new users the basics of Julia and JuMP.

How these tutorials are structured

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

- The "Getting started with" tutorials are basic introductions to different aspects of JuMP and Julia. If you are new to JuMP and Julia, start by reading them in the following order:

    - Getting started with Julia
    - Getting started with JuMP
    - Getting started with sets and indexing
    - Getting started with data and plotting

- Julia has a reputation for being "fast." Unfortunately, it is also easy to write slow Julia code. Performance tips contains a number of important tips on how to improve the performance of models you write in JuMP.

- Design patterns for larger models is a more advanced tutorial that is aimed at users writing large JuMP models. It's in the "Getting started" section to give you an early preview of how JuMP makes it easy to structure larger models. If you are new to JuMP you may want to skip or briefly skim this tutorial, and come back to it once you have written a few JuMP models.

4.2 Getting started with Julia

Because JuMP is embedded in Julia, knowing some basic Julia is important before you start 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.

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.

Julia can also be used with Jupyter notebooks or the reactive notebooks of Pluto.jl.

The Julia REPL

The main way of interacting with Julia is via its REPL (Read Evaluate Print Loop). To access the REPL, start the Julia executable to arrive at the `julia>` prompt, and then start coding:

```julia
julia> 1 + 1
2
```

As your programs become larger, write a script as a text file, and then run that file using:

```julia
julia> include("path/to/file.jl")
```

> **Warning**
>
> Because of Julia's startup latency, running scripts from the command line like the following is slow:
>
> ```
> $ julia path/to/file.jl
> ```
>
> Use the REPL or a notebook instead, and read The "time-to-first-solve" issue for more information.

Code blocks in this documentation

In this documentation you'll see a mix of code examples with and without the `julia>`.

The Julia prompt is mostly used to demonstrate short code snippets, and the output is exactly what you will see if run from the REPL.

Blocks without the `julia>` can be copy-pasted into the REPL, but they are used because they enable richer output like plots or LaTeX to be displayed in the online and PDF versions of the documentation. If you run them from the REPL you may see different output.

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 ? to enter help-mode, followed by the name of the function to lookup:

```
help?> print
search: print println printstyled sprint isprint prevind parentindices precision escape_string

  print([io::IO], xs...)

  Write to io (or to the default output stream stdout if io is not given) a canonical
  (un-decorated) text representation. The representation used by print includes minimal formatting
  and tries to avoid Julia-specific details.

  print falls back to calling show, so most types should just define show. Define print if your
  type has a separate "plain" representation. For example, show displays strings with quotes, and
  print displays strings without quotes.

  string returns the output of print as a string.

  Examples
  ==========

  julia> print("Hello World!")
  Hello World!
  julia> io = IOBuffer();

  julia> print(io, "Hello", ' ', :World!)

  julia> String(take!(io))
  "Hello World!"
```

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:

```
julia> 1 + 1
2

julia> 1 - 2
-1

julia> 2 * 2
4

julia> 4 / 5
0.8

julia> 3^2
9
```

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:

```julia
julia> typeof(1)
Int64

julia> 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`:

```julia
julia> x = 2 + 1im
2 + 1im

julia> real(x)
2

julia> imag(x)
1

julia> typeof(x)
Complex{Int64}

julia> 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.

```julia
julia> π
π = 3.1415926535897...
```

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

```
julia> typeof(π)
Irrational{:π}
```

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

```
julia> 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:

```
julia> 0.1 * 3 == 0.3
false
```

A more complicated example is:

```
julia> sin(2π / 3) == √3 / 2
false
```

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

Let's see what the differences are:

```
julia> 0.1 * 3 - 0.3
5.551115123125783e-17

julia> 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 ==:

```julia
julia> 0.1 * 3 ≈ 0.3
true

julia> sin(2π / 3) ≈ √3 / 2
true
```

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

```julia
julia> 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).

```julia
julia> 1e-300 ≈ 0.0
false
```

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

```julia
julia> 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:

```julia
julia> 1 + 1e-16 == 1
true
```

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

```julia
julia> (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:

```julia
julia> b = [5, 6]
2-element Vector{Int64}:
 5
 6
```

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

```julia
julia> 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:

```julia
julia> 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].

```julia
julia> A * x
2-element Vector{Float64}:
 5.0
 6.0

julia> A * x ≈ b
true
```

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

```julia
julia>    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:

```julia
julia> b' * b
61

julia> b * b'
2×2 Matrix{Int64}:
 25  30
 30  36
```

Other common types

Comments

Although not technically a type, code comments begin with the # character:

```julia
julia> 1 + 1  # This is a comment
2
```

Multiline comments begin with #= and end with =#:

```julia
#=
Here is a
multiline comment
=#
```

Comments can even be nested inside expressions. This is sometimes helpful when documenting inputs to functions:

```julia
julia> isapprox(
           sin(π),
           0.0;
           #= We need an explicit atol here because we are comparing with 0 =#
           atol = 0.001,
       )
true
```

Strings

Double quotes are used for strings:

```julia
julia> typeof("This is Julia")
String
```

Unicode is fine in strings:

```julia
julia> typeof("π is about 3.1415")
String
```

Use `println` to print a string:

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

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

```
julia> x = 123
123

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

Use triple-quotes for multiline strings:

```
julia> s = """
       Here is
       a
       multiline string
       """
"Here is\na\nmultiline string\n"

julia> println(s)
Here is
a
multiline string
```

Symbols

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

```
julia> println("The value of x is: $(eval(:x))")
The value of x is: 123
```

> **Warning**
>
> 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.

```
julia> 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:

```julia
julia> String(:abc)
"abc"

julia> Symbol("abc")
:abc
```

> **Tip**
>
> Symbols are often (ab)used to stand in for a `String` or an Enum, when one of the latter is likely a better choice. The JuMP Style guide recommends reserving Symbols 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:

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

julia> typeof(t)
Tuple{String, Float64, Symbol}
```

Tuples can be accessed by index, similar to arrays:

```julia
julia> t[2]
1.2
```

And they can be "unpacked" like so:

```julia
julia> a, b, c = t
("hello", 1.2, :foo)

julia> b
1.2
```

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

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

Values can be accessed using dot syntax:

```julia
julia> 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:

```julia
julia> 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 value uses the bracket syntax:

```julia
julia> d1[2]
"B"
```

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

```julia
julia> 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}`:

```julia
julia> 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:

```julia
julia> 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)
```

```julia
julia> d2["B"]
2

julia> d2["D"][:foo]
3
```

Structs

You can define custom datastructures with `struct`:

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


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

julia> a.x
1
```

By default, these are not mutable

```julia
julia>    a.x = 2
setfield! immutable struct of type MyStruct cannot be changed
```

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

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


julia> a = MyStructMutable(1, "a", Dict(2 => 3))
Main.MyStructMutable(1, "a", Dict(2 => 3))

julia> a.x
1

julia> a.x = 2
2

julia> 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`:

```julia
julia> for i in 1:5
           println(i)
       end
1
2
3
4
5
```

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

```julia
julia> 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:

```julia
julia> 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:

```julia
julia> 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:

```
julia> [i for i in 1:5]
5-element Vector{Int64}:
 1
 2
 3
 4
 5
```

Matrices can be built by including multiple indices:

```
julia> [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:

```
julia> [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:

```
julia> 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:

```julia
julia> function print_hello()
            return println("hello")
       end
print_hello (generic function with 1 method)

julia> print_hello()
hello
```

Arguments can be added to a function:

```julia
julia> function print_it(x)
            return println(x)
       end
print_it (generic function with 1 method)

julia> print_it("hello")
hello

julia> print_it(1.234)
1.234

julia> print_it(:my_id)
my_id
```

Optional keyword arguments are also possible:

```julia
julia> function print_it(x; prefix = "value:")
            return println("$(prefix) $(x)")
       end
print_it (generic function with 1 method)

julia> print_it(1.234)
value: 1.234

julia> print_it(1.234; prefix = "val:")
val: 1.234
```

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

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

```
mult (generic function with 1 method)

julia> mult(4.0)
8.0

julia> 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:

```
julia> f = x -> x^2
#11 (generic function with 1 method)

julia> f(2)
4

julia> 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:

```
julia> function foo(x::Int)
           return x^2
       end
foo (generic function with 1 method)

julia> function foo(x::Float64)
           return exp(x)
       end
foo (generic function with 2 methods)

julia> function foo(x::Number)
           return x + 1
       end
foo (generic function with 3 methods)

julia> foo(2)
4

julia> foo(2.0)
7.38905609893065

julia> foo(1 + 1im)
2 + 1im
```

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

```
julia>     foo([1, 2, 3])
MethodError: no method matching foo(::Vector{Int64})
Closest candidates are:
  foo(!Matched::Int64) at REPL[1]:1
  foo(!Matched::Float64) at REPL[2]:1
  foo(!Matched::Number) at REPL[3]:1
```

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:

```
julia> 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.

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 cannot modify the contents of a `Float64`.

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

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

```
julia> mutable_type = [1, 2, 3]
3-element Vector{Int64}:
 1
 2
 3

julia> immutable_type = 1
1

julia> mutability_example(mutable_type, immutable_type)


julia> println("mutable_type: $(mutable_type)")
mutable_type: [2, 2, 3]

julia> println("immutable_type: $(immutable_type)")
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:

```
julia> isimmutable([1, 2, 3])
false

julia> 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.

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


julia> import Random  # The equivalent of Python's `import Random`


julia> Random.seed!(33)
MersenneTwister(33)

julia> [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.

4.3 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, start by reading 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, HiGHS 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.

Because JuMP is built on top of MOI, you'll often see the `MathOptInterface.` prefix displayed when JuMP types are printed. However, you'll only need to understand and interact with MOI to accomplish advanced tasks such as creating solver-independent callbacks.

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 `HiGHS.Optimizer`, which is provided by the HiGHS.jl package.

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

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

An example

Let's solve the following linear programming problem using JuMP and HiGHS. 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:

```julia
julia> using JuMP


julia> using HiGHS


julia> model = Model(HiGHS.Optimizer)
A JuMP Model
Feasibility problem with:
Variables: 0
Model mode: AUTOMATIC
CachingOptimizer state: EMPTY_OPTIMIZER
Solver name: HiGHS

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

julia> @variable(model, 0 <= y <= 3)
y

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

julia> @constraint(model, c1, 6x + 8y >= 100)
c1 : 6 x + 8 y ≥ 100

julia> @constraint(model, c2, 7x + 12y >= 120)
c2 : 7 x + 12 y ≥ 120

julia> print(model)
Min 12 x + 20 y
Subject to
 c1 : 6 x + 8 y ≥ 100
 c2 : 7 x + 12 y ≥ 120
 x ≥ 0
 y ≥ 0
 y ≤ 3

julia> optimize!(model)
Running HiGHS 1.5.1 [date: 1970-01-01, git hash: 93f1876e4]
Copyright (c) 2023 HiGHS under MIT licence terms
Presolving model
2 rows, 2 cols, 4 nonzeros
2 rows, 2 cols, 4 nonzeros
Presolve : Reductions: rows 2(-0); columns 2(-0); elements 4(-0) - Not reduced
```

```
Problem not reduced by presolve: solving the LP
Using EKK dual simplex solver - serial
  Iteration        Objective      Infeasibilities num(sum)
          0     0.0000000000e+00 Pr: 2(220) 0s
          2     2.0500000000e+02 Pr: 0(0) 0s
Model   status      : Optimal
Simplex   iterations: 2
Objective value     :   2.0500000000e+02
HiGHS run time      :            0.00

julia> termination_status(model)
OPTIMAL::TerminationStatusCode = 1

julia> primal_status(model)
FEASIBLE_POINT::ResultStatusCode = 1

julia> dual_status(model)
FEASIBLE_POINT::ResultStatusCode = 1

julia> objective_value(model)
204.99999999999997

julia> value(x)
15.000000000000005

julia> value(y)
1.249999999999996

julia> shadow_price(c1)
-0.24999999999999922

julia> shadow_price(c2)
-1.5000000000000007
```

Step-by-step

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

```
julia> using JuMP
```

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

```
julia> using HiGHS
```

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

```
julia> model = Model(HiGHS.Optimizer)
A JuMP Model
Feasibility problem with:
Variables: 0
```

```
Model mode: AUTOMATIC
CachingOptimizer state: EMPTY_OPTIMIZER
Solver name: HiGHS
```

Variables are modeled using @variable:

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

> **Info**
>
> The macro creates a new Julia object, x, in the current scope. We could have made this more explicit by writing:
>
> ```
> x = @variable(model, x >= 0)
> ```
>
> but the macro does this automatically for us to save writing x twice.

Variables can have lower and upper bounds:

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

The objective is set using @objective:

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

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

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

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

Call print to display the model:

```
julia> print(model)
Min 12 x + 20 y
Subject to
 c1 : 6 x + 8 y ≥ 100
 c2 : 7 x + 12 y ≥ 120
 x ≥ 0
 y ≥ 0
 y ≤ 30
```

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

```
julia> optimize!(model)
Running HiGHS 1.5.1 [date: 1970-01-01, git hash: 93f1876e4]
Copyright (c) 2023 HiGHS under MIT licence terms
Presolving model
2 rows, 2 cols, 4 nonzeros
2 rows, 2 cols, 4 nonzeros
Presolve : Reductions: rows 2(-0); columns 2(-0); elements 4(-0) - Not reduced
Problem not reduced by presolve: solving the LP
Using EKK dual simplex solver - serial
  Iteration        Objective     Infeasibilities num(sum)
          0     0.0000000000e+00 Pr: 2(220) 0s
          2     2.0500000000e+02 Pr: 0(0) 0s
Model   status      : Optimal
Simplex   iterations: 2
Objective value     :  2.0500000000e+02
HiGHS run time      :          0.00
```

> **Info**
>
> The `!` after optimize is 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:

```
julia> 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:

```
julia> primal_status(model)
FEASIBLE_POINT::ResultStatusCode = 1
```

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

```
julia> 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`:

```
julia> objective_value(model)
205.0
```

the primal solution using `value`:

```
julia> value(x)
15.000000000000004

julia> value(y)
1.2499999999999976
```

and the dual solution using shadow_price:

```
julia> shadow_price(c1)
-0.24999999999999917

julia> 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:

```
julia> model = Model(HiGHS.Optimizer)
A JuMP Model
Feasibility problem with:
Variables: 0
Model mode: AUTOMATIC
CachingOptimizer state: EMPTY_OPTIMIZER
Solver name: HiGHS
```

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

```
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, HiGHS.Optimizer)
```

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

```
julia> model = direct_model(HiGHS.Optimizer())
A JuMP Model
Feasibility problem with:
Variables: 0
Model mode: DIRECT
Solver name: HiGHS
```

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

Solver Options

Pass options to solvers with `optimizer_with_attributes`:

```
julia> model =
           Model(optimizer_with_attributes(HiGHS.Optimizer, "output_flag" => false))
A JuMP Model
Feasibility problem with:
Variables: 0
Model mode: AUTOMATIC
CachingOptimizer state: EMPTY_OPTIMIZER
Solver name: HiGHS
```

> **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_attribute`:

```
julia> model = Model(HiGHS.Optimizer)
A JuMP Model
Feasibility problem with:
Variables: 0
Model mode: AUTOMATIC
CachingOptimizer state: EMPTY_OPTIMIZER
Solver name: HiGHS

julia> set_attribute(model, "output_flag", false)
```

Solution basics

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

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

```
julia> function solve_infeasible()
           model = Model(HiGHS.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
        end
        return value(x), value(y)
    end
solve_infeasible (generic function with 1 method)

julia> solve_infeasible()
Running HiGHS 1.5.1 [date: 1970-01-01, git hash: 93f1876e4]
Copyright (c) 2023 HiGHS under MIT licence terms
Presolving model
Problem status detected on presolve: Infeasible
Model   status      : Infeasible
Objective value     :  0.0000000000e+00
HiGHS run time      :            0.00
ERROR:   No invertible representation for getDualRay
⌐ Warning: The model was not solved correctly.
└ @ Main REPL[1]:9
```

Variable basics

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

```
julia> 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.

```
julia> @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.

```
julia> @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.

```
julia> has_upper_bound(keyword_x)
true

julia> upper_bound(keyword_x)
2.0
```

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

```julia
julia>    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: Arrays, DenseAxisArrays, and SparseAxisArrays.

Arrays

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

```julia
julia> @variable(model, a[1:2, 1:2])
2×2 Matrix{VariableRef}:
 a[1,1]  a[1,2]
 a[2,1]  a[2,2]
```

Index elements in a as follows:

```julia
julia> a[1, 1]
a[1,1]
```

```julia
julia> a[2, :]
2-element Vector{VariableRef}:
 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:

```julia
julia> n = 10
10

julia> l = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];

julia> u = [10, 11, 12, 13, 14, 15, 16, 17, 18, 19];

julia> @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:

```
julia> @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

DenseAxisArrays 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:

```
julia> @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:

```
julia> @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]
```

Index elements in a DenseAxisArray as follows:

```
julia> z[2, 1]
z[2,1]
```

```
julia> w[2:3, ["red", "blue"]]
2-dimensional DenseAxisArray{VariableRef,2,...} with index sets:
    Dimension 1, [2, 3]
    Dimension 2, ["red", "blue"]
And data, a 2×2 Matrix{VariableRef}:
 w[2,red]  w[2,blue]
 w[3,red]  w[3,blue]
```

See Forcing the container type for more details.

SparseAxisArrays

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

```julia
julia> @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 ;:

```julia
julia> @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]
```

Index elements in a DenseAxisArray as follows:

```julia
julia> u[1, 2]
u[1,2]
```

```julia
julia> v[[3, 6]]
JuMP.Containers.SparseAxisArray{VariableRef, 1, Tuple{Int64}} with 2 entries:
  [3]  =  v[3]
  [6]  =  v[6]
```

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:

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

or setting the integer keyword to true:

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

Binary variables

Create a binary variable by passing `Bin`:

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

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

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

Constraint basics

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

```julia
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> @variable(model, y)
y

julia> @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

Create an `Array` of constraints:

```julia
julia> @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
 2 x ≤ 3
 3 x ≤ 4
```

DenseAxisArrays

Create an `DenseAxisArray` of constraints:

```
julia> @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    x ≤ 4
 2 x ≤ 3  2 x ≤ 4
```

SparseAxisArrays

Create an `SparseAxisArray` of constraints:

```
julia> @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
  [2, 1]  =  2 x ≤ 2
```

Constraints in a loop

We can add constraints using regular Julia loops:

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

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

```
julia> @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
 6 x + 4 y ≥ 10
 6 x + 4 y ≥ 15
```

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

```
julia> @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] ≤ 1
```

Objective functions

Set an objective function with @objective:

```julia
julia> model = Model(HiGHS.Optimizer)
A JuMP Model
Feasibility problem with:
Variables: 0
Model mode: AUTOMATIC
CachingOptimizer state: EMPTY_OPTIMIZER
Solver name: HiGHS

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

julia> @variable(model, y >= 0)
y

julia> @objective(model, Min, 2x + y)
2 x + y
```

Create a maximization objective using Max:

```julia
julia> @objective(model, Max, 2x + y)
2 x + y
```

> **Tip**
>
> Calling @objective multiple times will over-write the previous objective. This can be useful when you want to solve the same problem with different objectives.

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 that is,

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

```julia
julia> vector_model = Model(HiGHS.Optimizer)
A JuMP Model
Feasibility problem with:
Variables: 0
Model mode: AUTOMATIC
CachingOptimizer state: EMPTY_OPTIMIZER
Solver name: HiGHS

julia> A = [1 1 9 5; 3 5 0 8; 2 0 6 13]
```

```
3×4 Matrix{Int64}:
 1  1  9   5
 3  5  0   8
 2  0  6  13

julia> b = [7, 3, 5]
3-element Vector{Int64}:
 7
 3
 5

julia> c = [1, 3, 5, 2]
4-element Vector{Int64}:
 1
 3
 5
 2

julia> @variable(vector_model, x[1:4] >= 0)
4-element Vector{VariableRef}:
 x[1]
 x[2]
 x[3]
 x[4]

julia> @constraint(vector_model, A * x .== b)
3-element Vector{ConstraintRef{Model,
↪   MathOptInterface.ConstraintIndex{MathOptInterface.ScalarAffineFunction{Float64},
↪   MathOptInterface.EqualTo{Float64}}, ScalarShape}}:
 x[1] + x[2] + 9 x[3] + 5 x[4] = 7
 3 x[1] + 5 x[2] + 8 x[4] = 3
 2 x[1] + 6 x[3] + 13 x[4] = 5

julia> @objective(vector_model, Min, c' * x)
x[1] + 3 x[2] + 5 x[3] + 2 x[4]

julia> optimize!(vector_model)
Running HiGHS 1.5.1 [date: 1970-01-01, git hash: 93f1876e4]
Copyright (c) 2023 HiGHS under MIT licence terms
Presolving model
3 rows, 4 cols, 10 nonzeros
3 rows, 4 cols, 10 nonzeros
Presolve : Reductions: rows 3(-0); columns 4(-0); elements 10(-0) - Not reduced
Problem not reduced by presolve: solving the LP
Using EKK dual simplex solver - serial
  Iteration        Objective     Infeasibilities num(sum)
         0     0.0000000000e+00 Pr: 3(13.5) 0s
         4     4.9230769231e+00 Pr: 0(0) 0s
Model   status      : Optimal
Simplex   iterations: 4
Objective value     :  4.9230769231e+00
HiGHS run time      :          0.00

julia> objective_value(vector_model)
4.923076923076922
```

4.4 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 (Chapter 5, "Simple Sets and Indexing," and Chapter 6, "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 describe 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"), ("B", "D"), ("C", "D"), ("A", "E"), ("B", "E"), ("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.

Sets to watch out for

JuMP supports any sets which are iterable, that is, the set set supports a for-loop like: [i for i in set]. This causes a few common errors.

First, if T = 3, you may pass the integer T by mistake instead of a range like 1:T:

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

```
1-dimensional DenseAxisArray{VariableRef,1,...} with index sets:
    Dimension 1, [3]
And data, a 1-element Vector{VariableRef}:
 x[3]
```

This results in a single variable being created, instead of three as desired. Because this is a common error, a warning is printed, advising you to pass a Vector{Int} instead:

```
@variable(model, x_fixed[[T]])
```

```
1-dimensional DenseAxisArray{VariableRef,1,...} with index sets:
    Dimension 1, [3]
And data, a 1-element Vector{VariableRef}:
 x_fixed[3]
```

Second, because `Strings` are iterable, passing a "index" as a singleton index is the same as passing `['i', 'n', 'd', 'e', 'x']`:

```
@variable(model, y["index"])
```

```
1-dimensional DenseAxisArray{VariableRef,1,...} with index sets:
    Dimension 1, ['i', 'n', 'd', 'e', 'x']
And data, a 5-element Vector{VariableRef}:
 y[i]
 y[n]
 y[d]
 y[e]
 y[x]
```

This time, a warning is not printed, but the work-around is similar, pass a `Vector{String}` instead:

```
@variable(model, y_fixed[["index"]])
```

```
1-dimensional DenseAxisArray{VariableRef,1,...} with index sets:
    Dimension 1, ["index"]
And data, a 1-element Vector{VariableRef}:
 y_fixed[index]
```

> **Tip**
>
> As a rule of thumb, if you want an index with one element, avoid confusion by passing `[index]` instead of `index`.

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 (that is, 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))
```

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

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.

4.5   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
```

> **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	Died	Jedi	Species	Weapon
	String31	String7	Float64	String7	String15	String7	String15	String15	String15	String15	String7	String15	String15
1	Anakin Skywalker	male	1.88	84	blue	blond	fair	Tatooine	41.9BBY	4ABY	jedi	human	lightsaber
2	Padme Amidala	female	1.65	45	brown	brown	light	Naboo	46BBY	19BBY	no_jedi	human	unarmed
3	Luke Skywalker	male	1.72	77	blue	blond	fair	Tatooine	19BBY	unk_died	jedi	human	lightsaber
4	Leia Skywalker	female	1.5	49	brown	brown	light	Alderaan	19BBY	unk_died	no_jedi	human	blaster
5	Qui-Gon Jinn	male	1.93	88.5	blue	brown	light	unk_planet	92BBY	32BBY	jedi	human	lightsaber
6	Obi-Wan Kenobi	male	1.82	77	bluegray	auburn	fair	Stewjon	57BBY	0BBY	jedi	human	lightsaber
7	Han Solo	male	1.8	80	brown	brown	light	Corellia	29BBY	unk_died	no_jedi	human	blaster
8	Sheev Palpatine	male	1.73	75	blue	red	pale	Naboo	82BBY	10ABY	no_jedi	human	force-lightning
9	R2-D2	male	0.96	32	NA	NA	NA	Naboo	33BBY	unk_died	no_jedi	droid	unarmed
10	C-3PO	male	1.67	75	NA	NA	NA	Tatooine	112BBY	3ABY	no_jedi	droid	unarmed
11	Yoda	male	0.66	17	brown	brown	green	unk_planet	896BBY	4ABY	jedi	yoda	lightsaber
12	Darth Maul	male	1.75	80	yellow	none	red	Dathomir	54BBY	unk_died	no_jedi	dathomirian	lightsaber
13	Dooku	male	1.93	86	brown	brown	light	Serenno	102BBY	19BBY	jedi	human	lightsaber
14	Chewbacca	male	2.28	112	blue	brown	NA	Kashyyyk	200BBY	25ABY	no_jedi	wookiee	bowcaster
15	Jabba	male	3.9	NA	yellow	none	tan-green	Tatooine	unk_born	4ABY	no_jedi	hutt	unarmed
16	Lando Calrissian	male	1.78	79	brown	blank	dark	Socorro	31BBY	unk_died	no_jedi	human	blaster
17	Boba Fett	male	1.83	78	brown	black	brown	Kamino	31.5BBY	unk_died	no_jedi	human	blaster
18	Jango Fett	male	1.83	79	brown	black	brown	ConcordDawn	66BBY	22BBY	no_jedi	human	blaster
19	Grievous	male	2.16	159	gold	black	orange	Kalee	unk_born	19BBY	no_jedi	kaleesh	slugthrower
20	Chief Chirpa	male	1.0	50	black	gray	brown	Endor	unk_born	4ABY	no_jedi	ewok	spear

Let's try plotting some of this data

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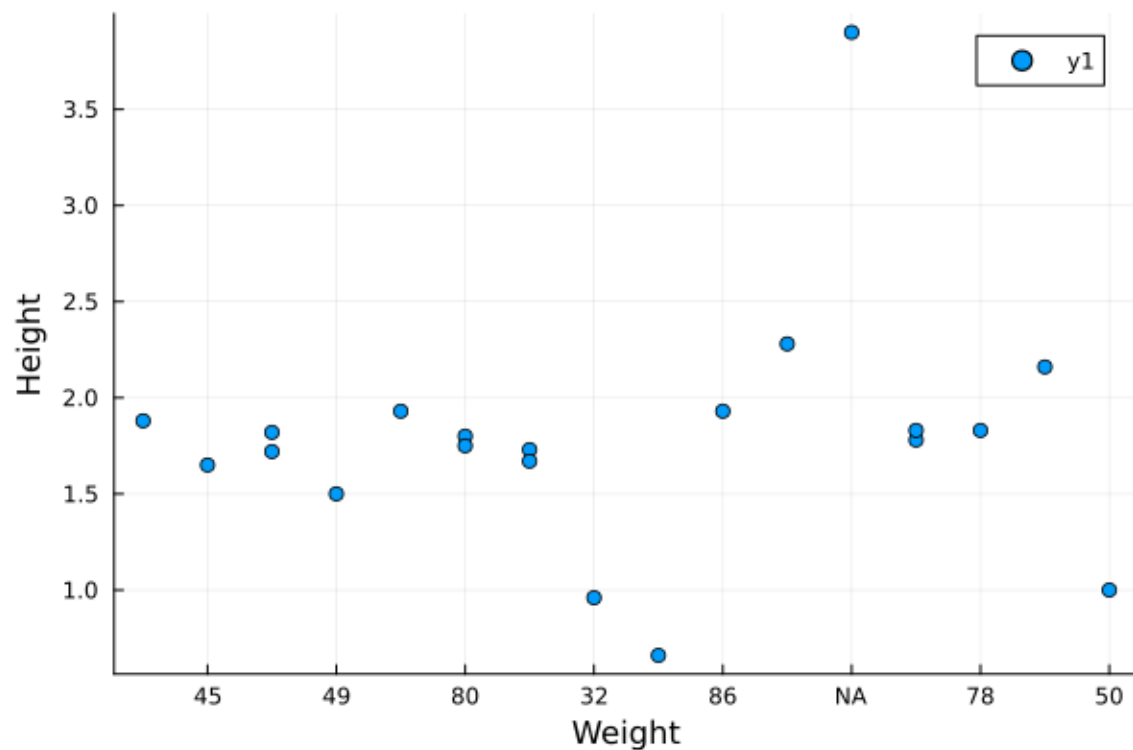
That doesn't look right. What happened? If you look at the dataframe above, it read `Weight` in as a `String` column because there are "NA" fields. Let's correct that, by telling CSV to consider "NA" as `missing`.

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

	Name	Gender	Height	Weight	Eyecolor	Haircolor	Skincolor	Homeland	Born	Died	Jedi	Species	Weapon
	String31	String7	Float64	Float64?	String15	String7	String15	String15	String15	String15	String7	String15	String15
1	Anakin Skywalker	male	1.88	84.0	blue	blond	fair	Tatooine	41.9BBY	4ABY	jedi	human	lightsaber
2	Padme Amidala	female	1.65	45.0	brown	brown	light	Naboo	46BBY	19BBY	no_jedi	human	unarmed
3	Luke Skywalker	male	1.72	77.0	blue	blond	fair	Tatooine	19BBY	unk_died	jedi	human	lightsaber
4	Leia Skywalker	female	1.5	49.0	brown	brown	light	Alderaan	19BBY	unk_died	no_jedi	human	blaster
5	Qui-Gon Jinn	male	1.93	88.5	blue	brown	light	unk_planet	92BBY	32BBY	jedi	human	lightsaber
6	Obi-Wan Kenobi	male	1.82	77.0	bluegray	auburn	fair	Stewjon	57BBY	0BBY	jedi	human	lightsaber
7	Han Solo	male	1.8	80.0	brown	brown	light	Corellia	29BBY	unk_died	no_jedi	human	blaster
8	Sheev Palpatine	male	1.73	75.0	blue	red	pale	Naboo	82BBY	10ABY	no_jedi	human	force-lightning
9	R2-D2	male	0.96	32.0	missing	missing	missing	Naboo	33BBY	unk_died	no_jedi	droid	unarmed
10	C-3PO	male	1.67	75.0	missing	missing	missing	Tatooine	112BBY	3ABY	no_jedi	droid	unarmed
11	Yoda	male	0.66	17.0	brown	brown	green	unk_planet	896BBY	4ABY	jedi	yoda	lightsaber
12	Darth Maul	male	1.75	80.0	yellow	none	red	Dathomir	54BBY	unk_died	no_jedi	dathomirian	lightsaber
13	Dooku	male	1.93	86.0	brown	brown	light	Serenno	102BBY	19BBY	jedi	human	lightsaber
14	Chewbacca	male	2.28	112.0	blue	brown	missing	Kashyyyk	200BBY	25ABY	no_jedi	wookiee	bowcaster
15	Jabba	male	3.9	missing	yellow	none	tan-green	Tatooine	unk_born	4ABY	no_jedi	hutt	unarmed
16	Lando Calrissian	male	1.78	79.0	brown	blank	dark	Socorro	31BBY	unk_died	no_jedi	human	blaster
17	Boba Fett	male	1.83	78.0	brown	black	brown	Kamino	31.5BBY	unk_died	no_jedi	human	blaster
18	Jango Fett	male	1.83	79.0	brown	black	brown	ConcordDawn	66BBY	22BBY	no_jedi	human	blaster
19	Grievous	male	2.16	159.0	gold	black	orange	Kalee	unk_born	19BBY	no_jedi	kaleesh	slugthrower
20	Chief Chirpa	male	1.0	50.0	black	gray	brown	Endor	unk_born	4ABY	no_jedi	ewok	spear

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),
)
```

Height vs Weight of StarWars characters
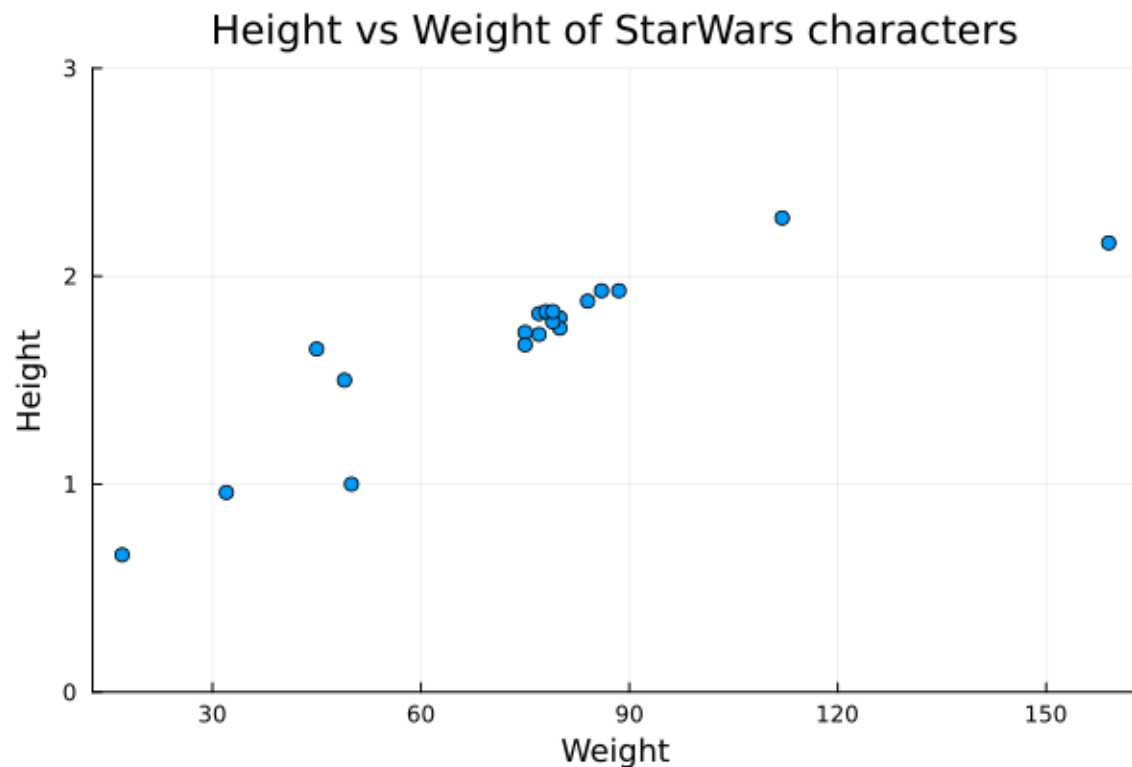
That looks better.

> **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
	String31	String7	Float64	Float64?	String15	String7	String15	String15	String15	String15
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	S
	String31	String7	Float64	Float64?	String15	String7	String15	String15	String15	String15	St
1	Chief Chirpa	male	1.0	50.0	black	gray	brown	Endor	unk_born	4ABY	n

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
	String15	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
	String15	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
	String15	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:

```julia
sort!(eyecolor_count, :count; rev = true)
Plots.bar(
    eyecolor_count.Eyecolor,
    eyecolor_count.count;
    xlabel = "Eye color",
    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.

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

We can also specify the delimiter as follows:

```julia
delim_df = CSV.read(
    joinpath(DATA_DIR, "Soccer.txt"),
```

	Name	Cups	Calories	Carbs	Fat	Fiber	Potassium	Protein	Sodium	Sugars
	String31	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

```
    DataFrames.DataFrame;
    delim = "::",
)
```

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)
```

	Team	Played	Wins	Draws	Losses	Goals_for	Goals_against
	String31	Int64	Int64	Int64	Int64	String15	String15
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

```
(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	String31
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{InlineStrings.String31}:
 "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{InlineStrings.String31}:
 "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{InlineStrings.String31}:
 "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{InlineStrings.String31}:
 "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	Jedi	Species	Weapon
	String31	String7	Float64	Float64?	String15	String7	String15	String15	String15	String15	String7	String15	String15
1	Anakin Skywalker	male	1.88	84.0	blue	blond	fair	Tatooine	41.9BBY	4ABY	jedi	human	lightsaber

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

	Name	Gender	Height	Weight	Eyecolor	Haircolor	Skincolor	Homeland	Born	Died	Jedi	Species	Weapon
	String31	String7	Float64	Float64?	String15	String7	String15	String15	String15	String15	String7	String15	String15
1	Anakin Skywalker	male	1.88	84.0	blue	blond	fair	Tatooine	41.9BBY	4ABY	jedi	human	lightsaber

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
```

> **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

	Name	Gender	Height	Weight	Eyecolor	Haircolor	Skincolor	Homeland	Born	Died	Jedi	Species	Weapon
	String31	String7	Float64	Float64?	String15	String7	String15	String15	String15	String15	String7	String15	String15
1	Anakin Skywalker	male	1.83	84.0	blue	blond	fair	Tatooine	41.9BBY	4ABY	jedi	human	lightsaber
2	Padme Amidala	female	1.83	45.0	brown	brown	light	Naboo	46BBY	19BBY	no_jedi	human	unarmed
3	Luke Skywalker	male	1.83	77.0	blue	blond	fair	Tatooine	19BBY	unk_died	jedi	human	lightsaber
4	Leia Skywalker	female	1.8	49.0	brown	brown	light	Alderaan	19BBY	unk_died	no_jedi	human	blaster
5	Qui-Gon Jinn	male	1.6	88.5	blue	brown	light	unk_planet	92BBY	32BBY	jedi	human	lightsaber
6	Obi-Wan Kenobi	male	1.8	77.0	bluegray	auburn	fair	Stewjon	57BBY	0BBY	jedi	human	lightsaber
7	Han Solo	male	1.8	80.0	brown	brown	light	Corellia	29BBY	unk_died	no_jedi	human	blaster
8	Sheev Palpatine	male	1.73	75.0	blue	red	pale	Naboo	82BBY	10ABY	no_jedi	human	force-lightning
9	R2-D2	male	0.96	32.0	missing	missing	missing	Naboo	33BBY	unk_died	no_jedi	droid	unarmed
10	C-3PO	male	1.67	75.0	missing	missing	missing	Tatooine	112BBY	3ABY	no_jedi	droid	unarmed
11	Yoda	male	0.66	17.0	brown	brown	green	unk_planet	896BBY	4ABY	jedi	yoda	lightsaber
12	Darth Maul	male	1.75	80.0	yellow	none	red	Dathomir	54BBY	unk_died	no_jedi	dathomirian	lightsaber
13	Dooku	male	1.93	86.0	brown	brown	light	Serenno	102BBY	19BBY	jedi	human	lightsaber
14	Chewbacca	male	2.28	112.0	blue	brown	missing	Kashyyyk	200BBY	25ABY	no_jedi	wookiee	bowcaster
15	Jabba	male	3.9	missing	yellow	none	tan-green	Tatooine	unk_born	4ABY	no_jedi	hutt	unarmed
16	Lando Calrissian	male	1.78	79.0	brown	blank	dark	Socorro	31BBY	unk_died	no_jedi	human	blaster
17	Boba Fett	male	1.83	78.0	brown	black	brown	Kamino	31.5BBY	unk_died	no_jedi	human	blaster
18	Jango Fett	male	1.83	79.0	brown	black	brown	ConcordDawn	66BBY	22BBY	no_jedi	human	blaster
19	Grievous	male	2.16	159.0	gold	black	orange	Kalee	unk_born	19BBY	no_jedi	kaleesh	slugthrower
20	Chief Chirpa	male	1.0	50.0	black	gray	brown	Endor	unk_born	4ABY	no_jedi	ewok	spear

Example: the passport problem

Let's now apply what we have learned to solve a real 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,
);
```

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
```

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 must 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 HiGHS
```

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(HiGHS.Optimizer)
set_silent(model)
@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: HiGHS
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 : HiGHS

* Status
  Result count       : 1
  Termination status : OPTIMAL
```

```
  Message from the solver:
  "kHighsModelStatusOptimal"

* Candidate solution (result #1)
  Primal status       : FEASIBLE_POINT
  Dual status         : NO_SOLUTION
  Objective value     : 2.30000e+01
  Objective bound     : 2.30000e+01
  Relative gap        : 0.00000e+00

* Work counters
  Solve time (sec)   : 7.77626e-03
  Simplex iterations : 26
  Barrier iterations : -1
  Node count         : 1
```

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
 * Chad
 * Comoros
 * Djibouti
 * Georgia
 * Hong Kong
 * India
 * Luxembourg
 * Madagascar
 * Maldives
 * Mali
 * New Zealand
 * North Korea
 * Papua New Guinea
 * Singapore
 * Somalia
```

```
 * Sri Lanka
 * Tunisia
 * Turkey
 * Uganda
 * United Arab Emirates
 * United States
 * Zimbabwe
```

We need some passports, like New Zealand 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.

4.6 Debugging

Dealing with bugs is an unavoidable part of coding optimization models in any framework, including JuMP. Sources of bugs include not only generic coding errors (method errors, typos, off-by-one issues), but also semantic mistakes in the formulation of an optimization problem and the incorrect use of a solver.

This tutorial explains some common sources of bugs and modeling issues that you might encounter when writing models in JuMP, and it suggests a variety of strategies to deal with them.

> **Tip**
>
> This tutorial is more advanced than the other "Getting started" tutorials. It's in the "Getting started" section to give you an early preview of how to debug JuMP models. However, if you are new to JuMP, you may want to briefly skim the tutorial, and come back to it once you have written a few JuMP models.

```julia
julia> using JuMP

julia> import HiGHS
```

Getting help

Debugging can be a frustrating part of modeling, particularly if you're new to optimization and programming. If you're stuck, join the community forum to search for answers to commonly asked questions.

Before asking a new 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.

Above all else, take time to simplify your code as much as possible. The fewer lines of code you can post that reproduces the same issue, the faster someone can answer your question.

Debugging Julia code

Read the Debugging chapter in the book ThinkJulia.jl. It has a number of great tips and tricks for debugging Julia code.

Solve failures

When a solver experiences an issue that prevents it from finding an optimal solution (or proving that one does not exist), JuMP may return one of a number of termination_statuses.

For example, if the solver found a solution, but experienced numerical imprecision, it may return a status such as ALMOST_OPTIMAL or ALMOST_LOCALLY_SOLVED indicating that the problem was solved to a relaxed set of tolerances. Alternatively, the solver may return a problematic status such as NUMERICAL_ERROR, SLOW_PROGRESS, or OTHER_ERROR, indicating that it could not find a solution to the problem.

Most solvers can experience numerical imprecision because they use floating-point arithmetic to perform operations such as addition, subtraction, and multiplication. These operations aren't exact, and small errors can accrue between the theoretical value and the value that the computer computes. For example:

```julia
julia> 0.1 * 3 == 0.3
false
```

> **Tip**
>
> Read the Guidelines for numerical issues section of the Gurobi documentation, along with the Debugging numerical problems section of the YALMIP documentation.

Common sources

Common sources of solve failures are:

- Very large numbers and very small numbers as problem coefficients. Exactly what "large" is depends on the solver and the problem, but in general, values above 1e6 or smaller than 1e-6 cause problems.

- Nonlinear problems with functions that are not defined in parts of their domain. For example, minimizing log(x) where x >= 0 is undefined when x = 0 (a common starting value).

Strategies

Strategies to debug sources of solve failures include:

- Rescale variables in the problem and their associated coefficients to make the magnitudes of all coefficients in the 1e-4 to 1e4 range. For example, that might mean rescaling a variable from measuring distance in centimeters to kilometers.

- Try a different solver. Some solvers might be more robust than others for a particular problem.

- Read the documentation of your solver, and try settings that encourage numerical robustness.

- Set bounds or add constraints so that all nonlinear functions are defined across all of the feasible region. This particularly applies for functions like 1 / x and log(x) which are not defined for x = 0.

Incorrect results

Sometimes, you might find that the solver returns an "optimal" solution that is incorrect according to the model you are trying to solve (perhaps the solution is suboptimal, or it doesn't satisfy some of the constraints).

Incorrect results can be hard to detect and debug, because the solver gives no hints that there is a problem. Indeed, the `termination_status` will likely be OPTIMAL and a solution will be available.

Common sources

Common sources of incorrect results are:

- A modeling error, so that your JuMP model does not match the formulation you have on paper

- Not accounting for the tolerances that solvers use (for example, if x is binary, a value like x = 1.0000001 may still be considered feasible)

- A bug in JuMP or the solver.

The probability of the issue being a bug in JuMP or the solver is much smaller than a modeling error. When in doubt, first assume there is a bug in your code before assuming that there is a bug in JuMP.

Strategies

Strategies to debug sources of incorrect results include:

- Print your JuMP model to see if it matches the formulation you have on paper. Look out for incorrect signs + instead of -, and off-by-one errors such as x[t] instead of x[t-1].

- Check that you are not using exact comparisons like value(x) == 1.0; always use isapprox(value(x), 1.0; atol = 1e-6) where you manually specify the comparison tolerance.

- Try a different solver. If one solver succeeds where another doesn't this is a sign that the problem is a numerical issue or a bug in the solver.

Debugging an infeasible model

A model is infeasible if there is no primal solution that satisfies all of the constraints. In general, an infeasible model means one of two things:

- Your problem really has no feasible solution

- There is a mistake in your model.

Example

A simple example of an infeasible model is:

```
julia> model = Model(HiGHS.Optimizer);


julia> set_silent(model)


```

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

julia> @objective(model, Max, 2x + 1)
2 x + 1

julia> @constraint(model, con, 2x - 1 <= -2)
con : 2 x ≤ -1
```

because the bound says that x >= 0, but we can rewrite the constraint to be x <= -1/2. When the problem is infeasible, JuMP may return one of a number of statuses. The most common is INFEASIBLE:

```julia
julia> optimize!(model)


julia> termination_status(model)
INFEASIBLE::TerminationStatusCode = 2
```

Depending on the solver, you may also receive INFEASIBLE_OR_UNBOUNDED or LOCALLY_INFEASIBLE.

A termination status of INFEASIBLE_OR_UNBOUNDED means that the solver could not prove if the solver was infeasible or unbounded, only that the model does not have a finite feasible optimal solution.

Nonlinear optimizers such as Ipopt may return the status LOCALLY_INFEASIBLE. This does not mean that the solver proved no feasible solution exists, only that it could not find one. If you know a primal feasible point, try providing it as a starting point using set_start_value and re-optimize.

Common sources

Common sources of infeasibility are:

- Incorrect units, for example, using a lower bound of megawatts and an upper bound of kilowatts

- Using + instead of - in a constraint

- Off-by-one and related errors, for example, using x[t] instead of x[t-1] in part of a constraint

- Otherwise invalid mathematical formulations

Strategies

Strategies to debug sources of infeasibility include:

- Iteratively comment out a constraint (or block of constraints) and re-solve the problem. When you find a constraint that makes the problem infeasible when added, check the constraint carefully for errors.

- If the problem is still infeasible with all constraints commented out, check all variable bounds. Do they use the right data?

- If you have a known feasible solution, use primal_feasibility_report to evaluate the constraints and check for violations. You'll probably find that you have a typo in one of the constraints.

- Try a different solver. Sometimes, solvers have bugs, and they can incorrectly report a problem as infeasible when it isn't. If you find such a case where one solver reports the problem is infeasible and another can find an optimal solution, please report it by opening an issue on the GitHub repository of the solver that reports infeasibility.

> **Tip**
>
> Some solvers also have specialized support for debugging sources of infeasibility via an irreducible infeasible subsystem. To see if your solver has support, try calling compute_conflict!:
>
> ```
> julia> compute_conflict!(model)
> ERROR: ArgumentError: The optimizer HiGHS.Optimizer does not support `compute_conflict!`
> ```
>
> In this case, HiGHS does not support computing conflicts, but other solvers such as Gurobi and CPLEX do. If the solver does support computing conflicts, read Conflicts for more details.

Penalty relaxation

Another strategy to debug sources of infeasibility is the relax_with_penalty! function.

The penalty relaxation modifies constraints of the form $f(x) \in S$ into $f(x) + y - z \in S$, where $y, z \geq 0$, and then it introduces a penalty term into the objective of $a \times (y + z)$ (if minimizing, else $-a$), where a is a penalty.

```
julia> map = relax_with_penalty!(model)
┌ Warning: Skipping PenaltyRelaxation for
└   ConstraintIndex{MathOptInterface.VariableIndex,MathOptInterface.GreaterThan{Float64}}
└ @ MathOptInterface.Utilities
└   ~/.julia/packages/MathOptInterface/goW8i/src/Utilities/penalty_relaxation.jl:289
Dict{ConstraintRef{Model,
└   MathOptInterface.ConstraintIndex{MathOptInterface.ScalarAffineFunction{Float64},
└   MathOptInterface.LessThan{Float64}}, ScalarShape}, AffExpr} with 1 entry:
  con : 2 x - _[2] ≤ -1 => _[2]
```

Here map is a dictionary which maps constraint indices to an affine expression representing $(y + z)$.

If we optimize the relaxed model, this time we get a feasible solution:

```
julia> optimize!(model)
```

```
julia> termination_status(model)
OPTIMAL::TerminationStatusCode = 1
```

Iterate over the contents of map to see which constraints are violated:

```
julia> for (con, penalty) in map
           violation = value(penalty)
           if violation > 0
               println("Constraint `$(name(con))` is violated by $violation")
```

```
            end
        end
Constraint `con` is violated by 1.0
```

Once you find a violated constraint in the relaxed problem, take a look to see if there is a typo or other common mistake in that particular constraint.

Consult the docstring `relax_with_penalty!` for information on how to modify the penalty cost term a, either for every constraint in the model or a particular subset of the constraints.

When using `relax_with_penalty!`, you should be aware that:

- Variable bounds and integrality restrictions are not relaxed. If the problem is still infeasible after calling `relax_with_penalty!`, check the variable bounds.

- You cannot undo the penalty relaxation. If you need an unmodified model, rebuild the problem, or call `copy_model` before calling `relax_with_penalty!`.

Debugging an unbounded model

A model is unbounded if there is no limit on how good the objective value can get. Most often, an unbounded model means that you have an error in your modeling, because all physical systems have limits. (You cannot make an infinite amount of profit.)

Example

A simple example of an unbounded model is:

```
julia> model = Model(HiGHS.Optimizer);


julia> set_silent(model)


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

julia> @objective(model, Max, 2x + 1)
2 x + 1
```

because we can increase x without limit, and the objective value 2x + 1 gets better as x increases.

When the problem is unbounded, JuMP may return one of a number of statuses. The most common is DUAL_INFEASIBLE:

```
julia> optimize!(model)


julia> termination_status(model)
DUAL_INFEASIBLE::TerminationStatusCode = 3
```

Depending on the solver, you may also receive INFEASIBLE_OR_UNBOUNDED or an error code like NORM_LIMIT.

Common sources

Common sources of unboundedness are:

- Using Max instead of Min

- Omitting variable bounds, such as 0 <= x <= 1

- Using + instead of - in a term of the objective function.

Strategies

Strategies to debug sources of unboundedness include:

- Double check whether you intended Min or Max in the @objective line.

- Print the objective function with print(objective_function(model)) and verify that the value and sign of each coefficient is as you expect.

- Add large bounds to all variables that are free or have one-sided bounds, then re-solve the problem. Because all variables are now bounded, the problem will have a finite optimal solution. Look at the value of each variable in the optimal solution to see if it is at one of the new bounds. If it is, you either need to specify a better bound for that variable, or there might be a mistake in the objective function associated with that variable (for example, a + instead of a -).

If there are too many variables to add bounds to, or there are too many terms to examine by hand, another strategy is to create a new variable with a large upper bound (if maximizing, lower bound if minimizing) and a constraint that the variable must be less-than or equal to the expression of the objective function. For example:

```julia
julia> model = Model(HiGHS.Optimizer);


julia> set_silent(model)


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

julia> # @objective(model, Max, 2x + 1)
       @variable(model, objective <= 10_000)
objective

julia> @constraint(model, objective <= 2x + 1)
-2 x + objective ≤ 1

julia> @objective(model, Max, objective)
objective
```

This new model has a finite optimal solution, so we can solve it and then look for variables with large positive or negative values in the optimal solution.

```julia
julia> optimize!(model)


julia> for var in all_variables(model)
           if var == objective
               continue
           end
           if abs(value(var)) > 1e3
               println("Variable `$(name(var))` may be unbounded")
           end
       end
Variable `x` may be unbounded
```

4.7 Design patterns for larger models

JuMP makes it easy to build and solve optimization models. However, once you start to construct larger models, and especially ones that interact with external data sources or have customizable sets of variables and constraints based on client choices, you may find that your scripts become unwieldy. This tutorial demonstrates a variety of ways in which you can structure larger JuMP models to improve their readability and maintainability.

> **Tip**
>
> This tutorial is more advanced than the other "Getting started" tutorials. It's in the "Getting started" section to give you an early preview of how JuMP makes it easy to structure larger models. However, if you are new to JuMP you may want to briefly skim the tutorial, and come back to it once you have written a few JuMP models.

Overview

This tutorial uses explanation-by-example. We're going to start with a simple knapsack model, and then expand it to add various features and structure.

A simple script

Your first prototype of a JuMP model is probably a script that uses a small set of hard-coded data.

```julia
using JuMP, HiGHS
profit = [5, 3, 2, 7, 4]
weight = [2, 8, 4, 2, 5]
capacity = 10
N = 5
model = Model(HiGHS.Optimizer)
@variable(model, x[1:N], Bin)
@objective(model, Max, sum(profit[i] * x[i] for i in 1:N))
@constraint(model, sum(weight[i] * x[i] for i in 1:N) <= capacity)
optimize!(model)
value.(x)
```

```
5-element Vector{Float64}:
  1.0
```

```
   0.0
  -0.0
   1.0
   1.0
```

The benefits of this approach are:

- it is quick to code

- it is quick to make changes.

The downsides include:

- all variables are global (read Performance tips)

- it is easy to introduce errors, for example, having `profit` and `weight` be vectors of different lengths, or not match N

- the solution, `x[i]`, is hard to interpret without knowing the order in which we provided the data.

Wrap the model in a function

A good next step is to wrap your model in a function. This is useful for a few reasons:

- it removes global variables

- it encapsulates the JuMP model and forces you to clarify your inputs and outputs

- we can add some error checking.

```julia
function solve_knapsack_1(profit::Vector, weight::Vector, capacity::Real)
    if length(profit) != length(weight)
        throw(DimensionMismatch("profit and weight are different sizes"))
    end
    N = length(weight)
    model = Model(HiGHS.Optimizer)
    @variable(model, x[1:N], Bin)
    @objective(model, Max, sum(profit[i] * x[i] for i in 1:N))
    @constraint(model, sum(weight[i] * x[i] for i in 1:N) <= capacity)
    optimize!(model)
    return value.(x)
end

solve_knapsack_1([5, 3, 2, 7, 4], [2, 8, 4, 2, 5], 10)
```

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

Create better data structures

Although we can check for errors like mis-matched vector lengths, if you start to develop models with a lot of data, keeping track of vectors and lengths and indices is fragile and a common source of bugs. A good solution is to use Julia's type system to create an abstraction over your data.

For example, we can create a `struct` that represents a single object, with a constructor that lets us validate assumptions on the input data:

```julia
struct KnapsackObject
    profit::Float64
    weight::Float64
    function KnapsackObject(profit::Float64, weight::Float64)
        if weight < 0
            throw(DomainError("Weight of object cannot be negative"))
        end
        return new(profit, weight)
    end
end
```

as well as a `struct` that holds a dictionary of objects and the knapsack's capacity:

```julia
struct KnapsackData
    objects::Dict{String,KnapsackObject}
    capacity::Float64
end
```

Here's what our data might look like now:

```julia
objects = Dict(
    "apple" => KnapsackObject(5.0, 2.0),
    "banana" => KnapsackObject(3.0, 8.0),
    "cherry" => KnapsackObject(2.0, 4.0),
    "date" => KnapsackObject(7.0, 2.0),
    "eggplant" => KnapsackObject(4.0, 5.0),
)
data = KnapsackData(objects, 10.0)
```

```
Main.KnapsackData(Dict{String, Main.KnapsackObject}("cherry" => Main.KnapsackObject(2.0, 4.0),
↪   "banana" => Main.KnapsackObject(3.0, 8.0), "date" => Main.KnapsackObject(7.0, 2.0), "eggplant"
↪   => Main.KnapsackObject(4.0, 5.0), "apple" => Main.KnapsackObject(5.0, 2.0)), 10.0)
```

If you want, you can add custom printing to make it easier to visualize:

```julia
function Base.show(io::IO, data::KnapsackData)
    println(io, "A knapsack with capacity $(data.capacity) and possible items:")
    for (k, v) in data.objects
        println(
            io,
            "  $(rpad(k, 8)) : profit = $(v.profit), weight = $(v.weight)",
```

```
        )
    end
    return
end

data
```

```
A knapsack with capacity 10.0 and possible items:
  cherry   : profit = 2.0, weight = 4.0
  banana   : profit = 3.0, weight = 8.0
  date     : profit = 7.0, weight = 2.0
  eggplant : profit = 4.0, weight = 5.0
  apple    : profit = 5.0, weight = 2.0
```

Then, we can re-write our solve_knapsack function to take our KnapsackData as input:

```julia
function solve_knapsack_2(data::KnapsackData)
    model = Model(HiGHS.Optimizer)
    @variable(model, x[keys(data.objects)], Bin)
    @objective(model, Max, sum(v.profit * x[k] for (k, v) in data.objects))
    @constraint(
        model,
        sum(v.weight * x[k] for (k, v) in data.objects) <= data.capacity,
    )
    optimize!(model)
    return value.(x)
end

solve_knapsack_2(data)
```

```
1-dimensional DenseAxisArray{Float64,1,...} with index sets:
    Dimension 1, ["cherry", "banana", "date", "eggplant", "apple"]
And data, a 5-element Vector{Float64}:
 -0.0
  0.0
  1.0
  1.0
  1.0
```

Read in data from files

Having a data structure is a good step. But it is still annoying that we have to hard-code the data into Julia. A good next step is to separate the data into an external file format; JSON is a common choice.

The JuMP repository has a file we're going to use for this tutorial. To run this tutorial locally, download the file and then update data_filename as appropriate.

To build this version of the JuMP documentation, we needed to set the filename:

```
data_filename = joinpath(@__DIR__, "data", "knapsack.json");
```

`knapsack.json` has the following contents:

```
println(read(data_filename, String))
```

```
{
    "objects": {
        "apple": {"profit": 5.0, "weight": 2.0},
        "banana": {"profit": 3.0, "weight": 8.0},
        "cherry": {"profit": 2.0, "weight": 4.0},
        "date": {"profit": 7.0, "weight": 2.0},
        "eggplant": {"profit": 4.0, "weight": 5.0}
    },
    "capacity": 10.0
}
```

Now let's write a function that reads this file and builds a `KnapsackData` object:

```
import JSON

function read_data(filename)
    d = JSON.parsefile(filename)
    return KnapsackData(
        Dict(
            k => KnapsackObject(v["profit"], v["weight"]) for
            (k, v) in d["objects"]
        ),
        d["capacity"],
    )
end

data = read_data(data_filename)
```

```
A knapsack with capacity 10.0 and possible items:
  cherry   : profit = 2.0, weight = 4.0
  banana   : profit = 3.0, weight = 8.0
  date     : profit = 7.0, weight = 2.0
  eggplant : profit = 4.0, weight = 5.0
  apple    : profit = 5.0, weight = 2.0
```

Add options via if-else

At this point, we have data in a file format which we can load and solve a single problem. For many users, this might be sufficient. However, at some point you may be asked to add features like "but what if we want to take more than one of a particular item?"

If this is the first time that you've been asked to add a feature, adding options via `if-else` statements is a good approach. For example, we might write:

```julia
function solve_knapsack_3(data::KnapsackData; binary_knapsack::Bool)
    model = Model(HiGHS.Optimizer)
    if binary_knapsack
        @variable(model, x[keys(data.objects)], Bin)
    else
        @variable(model, x[keys(data.objects)] >= 0, Int)
    end
    @objective(model, Max, sum(v.profit * x[k] for (k, v) in data.objects))
    @constraint(
        model,
        sum(v.weight * x[k] for (k, v) in data.objects) <= data.capacity,
    )
    optimize!(model)
    return value.(x)
end
```

```
solve_knapsack_3 (generic function with 1 method)
```

Now we can solve the binary knapsack:

```julia
solve_knapsack_3(data; binary_knapsack = true)
```

```
1-dimensional DenseAxisArray{Float64,1,...} with index sets:
    Dimension 1, ["cherry", "banana", "date", "eggplant", "apple"]
And data, a 5-element Vector{Float64}:
 -0.0
  0.0
  1.0
  1.0
  1.0
```

And an integer knapsack where we can take more than one copy of each item:

```julia
solve_knapsack_3(data; binary_knapsack = false)
```

```
1-dimensional DenseAxisArray{Float64,1,...} with index sets:
    Dimension 1, ["cherry", "banana", "date", "eggplant", "apple"]
And data, a 5-element Vector{Float64}:
 0.0
 0.0
 5.0
 0.0
 0.0
```

Add configuration options via dispatch

If you get repeated requests to add different options, you'll quickly find yourself in a mess of different flags and if-else statements. It's hard to write, hard to read, and hard to ensure you haven't introduced any bugs. A good solution is to use Julia's type dispatch to control the configuration of the model. The easiest way to explain this is by example.

First, start by defining a new abstract type, as well as new subtypes for each of our options. These types are going to control the configuration of the knapsack model.

```julia
abstract type AbstractConfiguration end

struct BinaryKnapsackConfig <: AbstractConfiguration end

struct IntegerKnapsackConfig <: AbstractConfiguration end
```

Then, we rewrite our solve_knapsack function to take a config argument, and we introduce an add_knapsack_variables function to abstract the creation of our variables.

```julia
function solve_knapsack_4(data::KnapsackData, config::AbstractConfiguration)
    model = Model(HiGHS.Optimizer)
    x = add_knapsack_variables(model, data, config)
    @objective(model, Max, sum(v.profit * x[k] for (k, v) in data.objects))
    @constraint(
        model,
        sum(v.weight * x[k] for (k, v) in data.objects) <= data.capacity,
    )
    optimize!(model)
    return value.(x)
end
```

```
solve_knapsack_4 (generic function with 1 method)
```

For the binary knapsack problem, add_knapsack_variables looks like this:

```julia
function add_knapsack_variables(
    model::Model,
    data::KnapsackData,
    ::BinaryKnapsackConfig,
)
    return @variable(model, x[keys(data.objects)], Bin)
end
```

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

For the integer knapsack problem, add_knapsack_variables looks like this:

```julia
function add_knapsack_variables(
    model::Model,
    data::KnapsackData,
    ::IntegerKnapsackConfig,
)
    return @variable(model, x[keys(data.objects)] >= 0, Int)
end
```

```
add_knapsack_variables (generic function with 2 methods)
```

Now we can solve the binary knapsack:

```julia
solve_knapsack_4(data, BinaryKnapsackConfig())
```

```
1-dimensional DenseAxisArray{Float64,1,...} with index sets:
    Dimension 1, ["cherry", "banana", "date", "eggplant", "apple"]
And data, a 5-element Vector{Float64}:
 -0.0
  0.0
  1.0
  1.0
  1.0
```

and the integer knapsack problem:

```julia
solve_knapsack_4(data, IntegerKnapsackConfig())
```

```
1-dimensional DenseAxisArray{Float64,1,...} with index sets:
    Dimension 1, ["cherry", "banana", "date", "eggplant", "apple"]
And data, a 5-element Vector{Float64}:
 0.0
 0.0
 5.0
 0.0
 0.0
```

The main benefit of the dispatch approach is that you can quickly add new options without needing to modify the existing code. For example:

```julia
struct UpperBoundedKnapsackConfig <: AbstractConfiguration
    limit::Int
end

function add_knapsack_variables(
    model::Model,
```

```
    data::KnapsackData,
    config::UpperBoundedKnapsackConfig,
)
    return @variable(model, 0 <= x[keys(data.objects)] <= config.limit, Int)
end

solve_knapsack_4(data, UpperBoundedKnapsackConfig(3))
```

```
1-dimensional DenseAxisArray{Float64,1,...} with index sets:
    Dimension 1, ["cherry", "banana", "date", "eggplant", "apple"]
And data, a 5-element Vector{Float64}:
 0.0
 0.0
 3.0
 0.0
 2.0
```

Generalize constraints and objectives

It's easy to extend the dispatch approach to constraints and objectives as well. The key points to notice in the next two functions are that:

- we can access registered variables via model[:x]

- we can define generic functions which accept any AbstractConfiguration as a configuration argument. That means we can implement a single method and have it apply to multiple configuration types.

```
function add_knapsack_constraints(
    model::Model,
    data::KnapsackData,
    ::AbstractConfiguration,
)
    x = model[:x]
    @constraint(
        model,
        capacity_constraint,
        sum(v.weight * x[k] for (k, v) in data.objects) <= data.capacity,
    )
    return
end

function add_knapsack_objective(
    model::Model,
    data::KnapsackData,
    ::AbstractConfiguration,
)
    x = model[:x]
    @objective(model, Max, sum(v.profit * x[k] for (k, v) in data.objects))
    return
end
```

```julia
function solve_knapsack_5(data::KnapsackData, config::AbstractConfiguration)
    model = Model(HiGHS.Optimizer)
    add_knapsack_variables(model, data, config)
    add_knapsack_constraints(model, data, config)
    add_knapsack_objective(model, data, config)
    optimize!(model)
    return value.(model[:x])
end

solve_knapsack_5(data, BinaryKnapsackConfig())
```

```
1-dimensional DenseAxisArray{Float64,1,...} with index sets:
    Dimension 1, ["cherry", "banana", "date", "eggplant", "apple"]
And data, a 5-element Vector{Float64}:
 -0.0
  0.0
  1.0
  1.0
  1.0
```

Remove solver dependence, add error checks

Compared to where we started, our knapsack model is now significantly different. We've wrapped it in a function, defined some data types, and introduced configuration options to control the variables and constraints that get added. There are a few other steps we can do to further improve things:

- remove the dependence on HiGHS

- add checks that we found an optimal solution

- add a helper function to avoid the need to explicitly construct the data.

```julia
function solve_knapsack_6(
    optimizer,
    data::KnapsackData,
    config::AbstractConfiguration,
)
    model = Model(optimizer)
    add_knapsack_variables(model, data, config)
    add_knapsack_constraints(model, data, config)
    add_knapsack_objective(model, data, config)
    optimize!(model)
    if termination_status(model) != OPTIMAL
        @warn("Model not solved to optimality")
        return nothing
    end
    return value.(model[:x])
end

function solve_knapsack_6(
    optimizer,
```

```
    data::String,
    config::AbstractConfiguration,
)
    return solve_knapsack_6(optimizer, read_data(data), config)
end

solution =
    solve_knapsack_6(HiGHS.Optimizer, data_filename, BinaryKnapsackConfig())
```

```
1-dimensional DenseAxisArray{Float64,1,...} with index sets:
    Dimension 1, ["cherry", "banana", "date", "eggplant", "apple"]
And data, a 5-element Vector{Float64}:
 -0.0
  0.0
  1.0
  1.0
  1.0
```

Create a module

Now we're ready to expose our model to the wider world. That might be as part of a larger Julia project that we're contributing to, or as a stand-alone script that we can run on-demand. In either case, it's good practice to wrap everything in a module. This further encapsulates our code into a single namespace, and we can add documentation in the form of docstrings.

Some good rules to follow when creating a module are:

- use import in a module instead of using to make it clear which functions are from which packages

- use _ to start function and type names that are considered private

- add docstrings to all public variables and functions.

```
module KnapsackModel

import JuMP
import JSON

struct _KnapsackObject
    profit::Float64
    weight::Float64
    function _KnapsackObject(profit::Float64, weight::Float64)
        if weight < 0
            throw(DomainError("Weight of object cannot be negative"))
        end
        return new(profit, weight)
    end
end

struct _KnapsackData
    objects::Dict{String,_KnapsackObject}
```

```julia
    capacity::Float64
end

function _read_data(filename)
    d = JSON.parsefile(filename)
    return _KnapsackData(
        Dict(
            k => _KnapsackObject(v["profit"], v["weight"]) for
            (k, v) in d["objects"]
        ),
        d["capacity"],
    )
end

abstract type _AbstractConfiguration end

"""
    BinaryKnapsackConfig()

Create a binary knapsack problem where each object can be taken 0 or 1 times.
"""
struct BinaryKnapsackConfig <: _AbstractConfiguration end

"""
    IntegerKnapsackConfig()

Create an integer knapsack problem where each object can be taken any number of
times.
"""
struct IntegerKnapsackConfig <: _AbstractConfiguration end

function _add_knapsack_variables(
    model::JuMP.Model,
    data::_KnapsackData,
    ::BinaryKnapsackConfig,
)
    return JuMP.@variable(model, x[keys(data.objects)], Bin)
end

function _add_knapsack_variables(
    model::JuMP.Model,
    data::_KnapsackData,
    ::IntegerKnapsackConfig,
)
    return JuMP.@variable(model, x[keys(data.objects)] >= 0, Int)
end

function _add_knapsack_constraints(
    model::JuMP.Model,
    data::_KnapsackData,
    ::_AbstractConfiguration,
)
    x = model[:x]
    JuMP.@constraint(
        model,
```

```julia
        capacity_constraint,
        sum(v.weight * x[k] for (k, v) in data.objects) <= data.capacity,
    )
    return
end

function _add_knapsack_objective(
    model::JuMP.Model,
    data::_KnapsackData,
    ::_AbstractConfiguration,
)
    x = model[:x]
    JuMP.@objective(model, Max, sum(v.profit * x[k] for (k, v) in data.objects))
    return
end

function _solve_knapsack(
    optimizer,
    data::_KnapsackData,
    config::_AbstractConfiguration,
)
    model = JuMP.Model(optimizer)
    _add_knapsack_variables(model, data, config)
    _add_knapsack_constraints(model, data, config)
    _add_knapsack_objective(model, data, config)
    JuMP.optimize!(model)
    if JuMP.termination_status(model) != JuMP.OPTIMAL
        @warn("Model not solved to optimality")
        return nothing
    end
    return JuMP.value.(model[:x])
end

"""
    solve_knapsack(
        optimizer,
        data_filename::String,
        config::_AbstractConfiguration,
    )

Solve the knapsack problem and return the optimal primal solution

# Arguments

 * `optimizer` : an object that can be passed to `JuMP.Model` to construct a new
   JuMP model.
 * `data_filename` : the filename of a JSON file containing the data for the
   problem.
 * `config` : an object to control the type of knapsack model constructed.
   Valid options are:
    * `BinaryKnapsackConfig()`
    * `IntegerKnapsackConfig()`

# Returns
```

```julia
 * If an optimal solution exists: a `JuMP.DenseAxisArray` that maps the `String`
   name of each object to the number of objects to pack into the knapsack.
 * Otherwise, `nothing`, indicating that the problem does not have an optimal
   solution.

# Examples

```julia
solution = solve_knapsack(
    HiGHS.Optimizer,
    "path/to/data.json",
    BinaryKnapsackConfig(),
)
```

```julia
solution = solve_knapsack(
    MOI.OptimizerWithAttributes(HiGHS.Optimizer, "output_flag" => false),
    "path/to/data.json",
    IntegerKnapsackConfig(),
)
```
"""
function solve_knapsack(
    optimizer,
    data_filename::String,
    config::_AbstractConfiguration,
)
    return _solve_knapsack(optimizer, _read_data(data_filename), config)
end

end
```

```
Main.KnapsackModel
```

Finally, you can call your model:

```julia
import .KnapsackModel

KnapsackModel.solve_knapsack(
    HiGHS.Optimizer,
    joinpath(@__DIR__, "data", "knapsack.json"),
    KnapsackModel.BinaryKnapsackConfig(),
)
```

```
1-dimensional DenseAxisArray{Float64,1,...} with index sets:
    Dimension 1, ["cherry", "banana", "date", "eggplant", "apple"]
And data, a 5-element Vector{Float64}:
 -0.0
  0.0
```

```
1.0
1.0
1.0
```

> **Note**
>
> The `.` in `.KnapsackModel` denotes that it is a submodule and not a separate package that we installed with `Pkg.add`. If you put the KnapsackModel in a separate file, load it with:
>
> ```julia
> include("path/to/KnapsackModel.jl")
> import .KnapsackModel
> ```

Add tests

As a final step, you should add tests for your model. This often means testing on a small problem for which you can work out the optimal solution by hand. The Julia standard library `Test` has good unit-testing functionality.

```julia
import .KnapsackModel
using Test

@testset "KnapsackModel" begin
    @testset "feasible_binary_knapsack" begin
        x = KnapsackModel.solve_knapsack(
            HiGHS.Optimizer,
            joinpath(@__DIR__, "data", "knapsack.json"),
            KnapsackModel.BinaryKnapsackConfig(),
        )
        @test isapprox(x["apple"], 1, atol = 1e-5)
        @test isapprox(x["banana"], 0, atol = 1e-5)
        @test isapprox(x["cherry"], 0, atol = 1e-5)
        @test isapprox(x["date"], 1, atol = 1e-5)
        @test isapprox(x["eggplant"], 1, atol = 1e-5)
    end
    @testset "feasible_integer_knapsack" begin
        x = KnapsackModel.solve_knapsack(
            HiGHS.Optimizer,
            joinpath(@__DIR__, "data", "knapsack.json"),
            KnapsackModel.IntegerKnapsackConfig(),
        )
        @test isapprox(x["apple"], 0, atol = 1e-5)
        @test isapprox(x["banana"], 0, atol = 1e-5)
        @test isapprox(x["cherry"], 0, atol = 1e-5)
        @test isapprox(x["date"], 5, atol = 1e-5)
        @test isapprox(x["eggplant"], 0, atol = 1e-5)
    end
    @testset "infeasible_binary_knapsack" begin
        x = KnapsackModel.solve_knapsack(
            HiGHS.Optimizer,
            # This file contains data that makes the problem infeasible.
            joinpath(@__DIR__, "data", "knapsack_infeasible.json"),
            KnapsackModel.BinaryKnapsackConfig(),
```

```
        )
        @test x === nothing
    end
end
```

```
Test.DefaultTestSet("KnapsackModel", Any[Test.DefaultTestSet("feasible_binary_knapsack", Any[], 5,
↪  false, false), Test.DefaultTestSet("feasible_integer_knapsack", Any[], 5, false, false),
↪  Test.DefaultTestSet("infeasible_binary_knapsack", Any[], 1, false, false)], 0, false, false)
```

> **Tip**
>
> Place these tests in a separate file `test_knapsack_model.jl` so that you can run the tests by adding
> `include("test_knapsack_model.jl")` to any file where needed.

Next steps

We've only briefly scratched the surface of ways to create and structure large JuMP models, so consider this tutorial a starting point, rather than a comprehensive list of all the possible ways to structure JuMP models. If you are embarking on a large project that uses JuMP, a good next step is to look at ways people have written large JuMP projects "in the wild."

Here are some good examples (all co-incidentally related to energy):

- AnyMOD.jl

    - JuMP-dev 2021 talk
    - source code

- PowerModels.jl

    - JuMP-dev 2021 talk
    - source code

- PowerSimulations.jl

    - JuliaCon 2021 talk
    - source code

- UnitCommitment.jl

    - JuMP-dev 2021 talk
    - source code

4.8   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(HiGHS.Optimizer; add_bridges = false)
```

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

Use PackageCompiler

As an example of compilation latency, consider the following linear program with two variables and two constraints:

```
using JuMP, HiGHS
model = Model(HiGHS.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)
optimize!(model)
open("model.log", "w") do io
    print(io, solution_summary(model; verbose = true))
```

```
    return
end
```

```
Running HiGHS 1.5.1 [date: 1970-01-01, git hash: 93f1876e4]
Copyright (c) 2023 HiGHS under MIT licence terms
Presolving model
2 rows, 2 cols, 4 nonzeros
2 rows, 2 cols, 4 nonzeros
Presolve : Reductions: rows 2(-0); columns 2(-0); elements 4(-0) - Not reduced
Problem not reduced by presolve: solving the LP
Using EKK dual simplex solver - serial
  Iteration        Objective     Infeasibilities num(sum)
          0     0.0000000000e+00 Pr: 2(220) 0s
          2     2.0500000000e+02 Pr: 0(0) 0s
Model   status      : Optimal
Simplex   iterations: 2
Objective value     :   2.0500000000e+02
HiGHS run time      :            0.00
```

Saving the problem in `model.jl` and calling from the command line results in:

```
$ time julia model.jl
15.78s user 0.48s system 100% cpu 16.173 total
```

Clearly, 16 seconds is a large overhead to pay for solving this trivial model. However, the compilation latency is independent on the problem size, and so 16 seconds of additional overhead may be tolerable for larger models that take minutes or hours to solve.

In cases where the compilation latency is intolerable, JuMP is compatible with the PackageCompiler.jl package, which makes it easy to generate a custom sysimage (a binary extension to Julia that caches compiled code) that dramatically reduces the compilation latency. A custom image for our problem can be created as follows:

```
using PackageCompiler, Libdl
PackageCompiler.create_sysimage(
    ["JuMP", "HiGHS"],
    sysimage_path = "customimage." * Libdl.dlext,
    precompile_execution_file = "model.jl",
)
```

When Julia is run with the custom image, the run time is now 0.7 seconds instead of 16:

```
$ time julia --sysimage customimage model.jl
0.68s user 0.22s system 153% cpu 0.587 total
```

Other performance tweaks, such as disabling bridges or using direct mode can reduce this time further.

> **Note**
>
> `create_sysimage` only needs to be run once, and the same sysimage can be used–to a slight detriment of performance–even if we modify `model.jl` or run a different file.

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]
```

```
1584
```

> **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])
```

```
1040
```

Disable string names

By default, JuMP creates String names for variables and constraints and passes these to the solver. The benefit of passing names is that it improves the readability of log messages from the solver (for example, "variable x has invalid bounds" instead of "variable v1203 has invalid bounds"), but for larger models the overhead of passing names can be non-trivial.

Disable the creation of String names by setting set_string_name = false in the @variable and @constraint macros, or by calling set_string_names_on_creation to disable all names for a particular model:

```
model = Model()
set_string_names_on_creation(model, false)
@variable(model, x)
```

$$_{-1}$$

```
@constraint(model, c, 2x <= 1)
```

$$2_{-1} \leq 1$$

Note that this doesn't change how symbolic names and bindings are stored:

```
x
```

$$_{-1}$$

```
model[:x]
```

$$_{-1}$$

```
x === model[:x]
```

```
true
```

But you can no longer look up the variable by the string name:

```
variable_by_name(model, "x") === nothing
```

```
true
```

> **Info**
>
> For more information on the difference between string names, symbolic names, and bindings, see
> String names, symbolic names, and bindings.

Chapter 5

Linear programs

5.1 Introduction

Linear programs (LPs) are a fundamental class of optimization problems of the form:

$$\min_{x \in \mathbb{R}^n} \sum_{i=1}^{n} c_i x_i \tag{5.1}$$

$$\text{s.t.} l_j \leq \sum_{i=1}^{n} a_{ij} x_i \leq u_j \qquad\qquad j = 1 \ldots m \tag{5.2}$$

$$l_i \leq x_i \leq u_i \qquad\qquad i = 1 \ldots n. \tag{5.3}$$

The most important thing to note is that all terms are of the form `coefficient * variable`, and that there are no nonlinear terms or multiplications between variables.

Mixed-integer linear programs (MILPs) are extensions of linear programs in which some (or all) of the decision variables take discrete values.

How to choose a solver

Almost all solvers support linear programs; look for "LP" in the list of Supported solvers. However, fewer solvers support mixed-integer linear programs. Solvers supporting discrete variables start with "(MI)" in the list of Supported solvers.

How these tutorials are structured

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

- The following tutorials are worked examples that present a problem in words, then formulate it in mathematics, and then solve it in JuMP. This usually involves some sort of visualization of the solution. Start here if you are new to JuMP.

  - The diet problem
  - The cannery problem
  - The facility location problem

- The Tips and tricks tutorial contains a number of helpful reformulations and tricks you can use when modeling linear programs. Look here if you are stuck trying to formulate a problem as a linear program.

- The Sensitivity analysis of a linear program tutorial explains how to create sensitivity reports like those produced by the Excel Solver.

- The Callbacks tutorial explains how to write a variety of solver-independent callbacks. Look here if you want to write a callback.

- The remaining tutorials are less verbose and styled in the form of short code examples. These tutorials have less explanation, but may contain useful code snippets, particularly if they are similar to a problem you are trying to solve.

5.2 The diet problem

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

Required packages

This tutorial requires the following packages:

```
using JuMP
import DataFrames
import HiGHS
```

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 macro-nutrient profile $a_{m,f}$ for each macro-nutrient $m \in M$.

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:

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

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 "macro-nutrient" 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
    ],
    ["nutrient", "min", "max"],
)
```

	nutrient	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 HiGHS as our optimizer:

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

Next, we create a set of decision variables x, with one element for each row in the DataFrame, and 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]
```

To simplify things later on, we store the vector as a new column x in the DataFrame foods:

```
foods.x = Array(x)
```

```
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 minimize the total cost of purchasing food:

```
@objective(model, Min, sum(foods.cost .* foods.x));
```

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:

```
@constraint(
    model,
    [row in eachrow(limits)],
    row.min <= sum(foods[!, row.nutrient] .* foods.x) <= row.max,
);
```

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, 2200]
 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, 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, 65]
 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, 1779]
 x[hamburger] ≥ 0
 x[chicken] ≥ 0
 x[hot dog] ≥ 0
 x[fries] ≥ 0
 x[macaroni] ≥ 0
 x[pizza] ≥ 0
 x[salad] ≥ 0
 x[milk] ≥ 0
 x[ice cream] ≥ 0
```

Solution

Let's optimize and take a look at the solution:

```
optimize!(model)
solution_summary(model)
```

```
* Solver : HiGHS

* Status
```

```
  Result count       : 1
  Termination status : OPTIMAL
  Message from the solver:
  "kHighsModelStatusOptimal"

* Candidate solution (result #1)
  Primal status      : FEASIBLE_POINT
  Dual status        : FEASIBLE_POINT
  Objective value    : 1.18289e+01
  Objective bound    : 0.00000e+00
  Relative gap       : Inf
  Dual objective value : 1.18289e+01

* Work counters
  Solve time (sec)   : 2.87771e-04
  Simplex iterations : 6
  Barrier iterations : 0
  Node count         : -1
```

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

```
for row in eachrow(foods)
    println(row.name, " = ", value(row.x))
end
```

```
hamburger = 0.6045138888888871
chicken = 0.0
hot dog = 0.0
fries = 0.0
macaroni = 0.0
pizza = 0.0
salad = 0.0
milk = 6.9701388888888935
ice cream = 2.5913194444444447
```

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

We can also use the function `Containers.rowtable` to easily convert the result into a DataFrame:

```
table = Containers.rowtable(value, x; header = [:food, :quantity])
solution = DataFrames.DataFrame(table)
```

This makes it easy to perform analyses our solution:

```
filter!(row -> row.quantity > 0.0, solution)
```

	food	quantity
	String	Float64
1	hamburger	0.604514
2	chicken	0.0
3	hot dog	0.0
4	fries	0.0
5	macaroni	0.0
6	pizza	0.0
7	salad	0.0
8	milk	6.97014
9	ice cream	2.59132

	food	quantity
	String	Float64
1	hamburger	0.604514
2	milk	6.97014
3	ice cream	2.59132

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.

```
dairy_foods = ["milk", "ice cream"]
is_dairy = map(name -> name in dairy_foods, foods.name)
@constraint(model, sum(foods[is_dairy, :x]) <= 6)
optimize!(model)
solution_summary(model)
```

```
* Solver : HiGHS

* Status
  Result count       : 1
  Termination status : INFEASIBLE
  Message from the solver:
  "kHighsModelStatusInfeasible"

* Candidate solution (result #1)
  Primal status      : NO_SOLUTION
  Dual status        : INFEASIBILITY_CERTIFICATE
  Objective value    : 1.18289e+01
  Objective bound    : 0.00000e+00
  Relative gap       : Inf
  Dual objective value : 3.56146e+00

* Work counters
  Solve time (sec)   : 4.89473e-04
  Simplex iterations : 0
  Barrier iterations : 0
  Node count         : -1
```

There exists no feasible solution to our problem. Looks like we're stuck eating ice cream for dinner.

5.3   The cannery problem

This tutorial was originally contributed by Louis Luangkesorn.

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

This tutorial requires the following packages:

```julia
using JuMP
import HiGHS
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:

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

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.

```julia
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 HiGHS as our optimizer:

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

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

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
 x[San-Diego,Chicago] + x[San-Diego,Topeka] + x[San-Diego,New-York] ≤ 600
```

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
 x[Seattle,Topeka] + x[San-Diego,Topeka] ≥ 300
 x[Seattle,New-York] + x[San-Diego,New-York] ≥ 300
```

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));
```

Solution

Let's optimize and look at the solution:

```
optimize!(model)
solution_summary(model)
```

```
* Solver : HiGHS

* Status
  Result count       : 1
  Termination status : OPTIMAL
  Message from the solver:
  "kHighsModelStatusOptimal"

* Candidate solution (result #1)
  Primal status       : FEASIBLE_POINT
  Dual status         : FEASIBLE_POINT
  Objective value     : 1.68000e+03
  Objective bound     : 0.00000e+00
  Relative gap        : Inf
  Dual objective value : 1.68000e+03

* Work counters
  Solve time (sec)   : 2.77758e-04
  Simplex iterations : 3
  Barrier iterations : 0
  Node count         : -1
```

What's the optimal shipment?

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

```
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
```

5.4 Tips and tricks

This tutorial was 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:

```julia
julia> using JuMP
```

Absolute value

To model the absolute value function $t \geq |x|$, there are a few options. In all cases, these reformulations only work if you are minimizing t "down" into $|x|$. They do not work if you are trying to maximize $|x|$.

Option 1

This option adds two linear inequality constraints:

```julia
julia> model = Model();

julia> @variable(model, x)
x

julia> @variable(model, t)
t

julia> @constraint(model, t >= x)
-x + t ≥ 0

julia> @constraint(model, t >= -x)
x + t ≥ 0
```

Option 2

This option uses two non-negative variables and forms expressions for x and t:

```julia
julia> model = Model();

julia> @variable(model, z[1:2] >= 0)
2-element Vector{VariableRef}:
 z[1]
 z[2]

julia> @expression(model, t, z[1] + z[2])
z[1] + z[2]

julia> @expression(model, x, z[1] - z[2])
z[1] - z[2]
```

Option 3

This option uses `MOI.NormOneCone` and lets JuMP choose the reformulation:

```julia
julia> model = Model();

julia> @variable(model, x)
x

julia> @variable(model, t)
t

julia> @constraint(model, [t; x] in MOI.NormOneCone(2))
[t, x] ∈ MathOptInterface.NormOneCone(2)
```

L1-norm

To model $\min \lVert x \rVert_1$, that is, $\min \sum_i |x_i|$, use the `MOI.NormOneCone`:

```julia
julia> model = Model();

julia> @variable(model, x[1:3])
3-element Vector{VariableRef}:
 x[1]
 x[2]
 x[3]

julia> @variable(model, t)
t

julia> @constraint(model, [t; x] in MOI.NormOneCone(1 + length(x)))
[t, x[1], x[2], x[3]] ∈ MathOptInterface.NormOneCone(4)

julia> @objective(model, Min, t)
t
```

Infinity-norm

To model $\min ||x||_\infty$, that is, $\min \max_i |x_i|$, use the MOI.NormInfinityCone:

```julia
julia> model = Model();


julia> @variable(model, x[1:3])
3-element Vector{VariableRef}:
 x[1]
 x[2]
 x[3]

julia> @variable(model, t)
t

julia> @constraint(model, [t; x] in MOI.NormInfinityCone(1 + length(x)))
[t, x[1], x[2], x[3]] ∈ MathOptInterface.NormInfinityCone(4)

julia> @objective(model, Min, t)
t
```

Max

To model $t \geq \max\{x, y\}$, do:

```julia
julia> model = Model();


julia> @variable(model, t)
t

julia> @variable(model, x)
x

julia> @variable(model, y)
y

julia> @constraint(model, t >= x)
t - x ≥ 0

julia> @constraint(model, t >= y)
t - y ≥ 0
```

This reformulation does not work for $t \geq \min\{x, y\}$.

Min

To model $t \leq \min\{x, y\}$, do:

```julia
julia> model = Model();

```

```
julia> @variable(model, t)
t

julia> @variable(model, x)
x

julia> @variable(model, y)
y

julia> @constraint(model, t <= x)
t - x ≤ 0

julia> @constraint(model, t <= y)
t - y ≤ 0
```

This reformulation does not work for $t \leq \max\{x, y\}$.

Boolean operators

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

Or

$$x_3 = x_1 \vee x_2$$

```
julia> model = Model();


julia> @variable(model, x[1:3], Bin)
3-element Vector{VariableRef}:
 x[1]
 x[2]
 x[3]

julia> @constraints(model, begin
           x[1] <= x[3]
           x[2] <= x[3]
           x[3] <= x[1] + x[2]
       end)
(x[1] - x[3] ≤ 0, x[2] - x[3] ≤ 0, -x[1] - x[2] + x[3] ≤ 0)
```

And

$$x_3 = x_1 \wedge x_2$$

```
julia> model = Model();


julia> @variable(model, x[1:3], Bin)
3-element Vector{VariableRef}:
 x[1]
```

```
 x[2]
 x[3]

julia> @constraints(model, begin
          x[3] <= x[1]
          x[3] <= x[2]
          x[3] >= x[1] + x[2] - 1
       end)
(-x[1] + x[3] ≤ 0, -x[2] + x[3] ≤ 0, -x[1] - x[2] + x[3] ≥ -1)
```

Not

$$x_1 \neg x_2$$

```
julia> model = Model();


julia> @variable(model, x[1:2], Bin)
2-element Vector{VariableRef}:
 x[1]
 x[2]

julia> @constraint(model, x[1] == 1 - x[2])
x[1] + x[2] = 1
```

Implies

$$x_1 \implies x_2$$

```
julia> model = Model();


julia> @variable(model, x[1:2], Bin)
2-element Vector{VariableRef}:
 x[1]
 x[2]

julia> @constraint(model, x[1] <= x[2])
x[1] - x[2] ≤ 0
```

Disjunctions

Problem

Suppose that we have two constraints $a^\top x \le b$ and $c^\top x \le d$, 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 $x_1 \le 1$ or $x_2 \le 2$.

```julia
julia> model = Model();


julia> @variable(model, x[1:2])
2-element Vector{VariableRef}:
 x[1]
 x[2]

julia> @variable(model, y, Bin)
y

julia> M = 100
100

julia> @constraint(model, x[1] <= 1 + M * y)
x[1] - 100 y ≤ 1

julia> @constraint(model, x[2] <= 2 + M * (1 - y))
x[2] + 100 y ≤ 102
```

> **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

Problem

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

$$z = 1 \implies a^\top x \le b$$

Trick 1

Some solvers have native support for indicator constraints.

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

```julia
julia> model = Model();


julia> @variable(model, x[1:2])
2-element Vector{VariableRef}:
 x[1]
 x[2]

julia> @variable(model, z, Bin)
z
```

```julia
julia> @constraint(model, z => {sum(x) <= 1})
z => {x[1] + x[2] ≤ 1}
```

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

```julia
julia> model = Model();


julia> @variable(model, x[1:2])
2-element Vector{VariableRef}:
 x[1]
 x[2]

julia> @variable(model, z, Bin)
z

julia> @constraint(model, !z => {sum(x) <= 1})
!z => {x[1] + x[2] ≤ 1}
```

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 $z = 1$.

```julia
julia> model = Model();


julia> @variable(model, x[1:2])
2-element Vector{VariableRef}:
 x[1]
 x[2]

julia> @variable(model, z, Bin)
z

julia> M = 100
100

julia> @constraint(model, sum(x) <= 1 + M * (1 - z))
x[1] + x[2] + 100 z ≤ 101
```

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

```julia
julia> model = Model();


julia> @variable(model, x[1:2])
2-element Vector{VariableRef}:
 x[1]
 x[2]
```

```
julia> @variable(model, z, Bin)
z

julia> M = 100
100

julia> @constraint(model, sum(x) <= 1 + M * z)
x[1] + x[2] - 100 z ≤ 1
```

Semi-continuous variables

> **Info**
>
> This section uses sets from MathOptInterface. By default, JuMP exports the MOI symbol as an alias for the MathOptInterface.jl package. We recommend making this more explicit in your code by adding the following lines:
>
> ```
> import MathOptInterface as MOI
> ```

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

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

```
julia> model = Model();


julia> @variable(model, x in 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: $x \in \{0\} \cup [l, u] \cap \mathbb{Z}$.

```
julia> model = Model();


julia> @variable(model, x in Semiinteger(5.0, 10.0))
x
```

Special Ordered Sets of Type 1

A Special Ordered Set of Type 1 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.

```julia
julia> model = Model();


julia> @variable(model, x[1:3], Bin)
3-element Vector{VariableRef}:
 x[1]
 x[2]
 x[3]

julia> @constraint(model, x in SOS1())
[x[1], x[2], x[3]] ∈ MathOptInterface.SOS1{Float64}([1.0, 2.0, 3.0])
```

You can optionally pass SOS1 a weight vector like

```julia
julia> @constraint(model, x in SOS1([0.2, 0.5, 0.3]))
[x[1], x[2], x[3]] ∈ 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 2

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.

```julia
julia> model = Model();


julia> @variable(model, x[1:3])
3-element Vector{VariableRef}:
 x[1]
 x[2]
 x[3]

julia> @constraint(model, x in SOS2([3.0, 1.0, 2.0]))
[x[1], x[2], x[3]] ∈ 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:

```julia
julia> x̂ = -1:0.5:2
-1.0:0.5:2.0
```

and a set of corresponding points for y:

```julia
julia> 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 x̂ and ŷ .

```julia
julia> N = length(x̂ )
7

julia> model = Model();

julia> @variable(model, -1 <= x <= 2)
x

julia> @variable(model, y)
y

julia> @variable(model, 0 <= λ[1:N] <= 1)
7-element Vector{VariableRef}:
 λ[1]
 λ[2]
 λ[3]
 λ[4]
 λ[5]
 λ[6]
 λ[7]

julia> @objective(model, Max, y)
y

julia> @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)
(x + λ[1] + 0.5 λ[2] - 0.5 λ[4] - λ[5] - 1.5 λ[6] - 2 λ[7] = 0, y - λ[1] - 0.25 λ[2] - 0.25 λ[4] -
↪  λ[5] - 2.25 λ[6] - 4 λ[7] = 0, λ[1] + λ[2] + λ[3] + λ[4] + λ[5] + λ[6] + λ[7] = 1, [λ[1], λ[2],
↪  λ[3], λ[4], λ[5], λ[6], λ[7]] ∈ MathOptInterface.SOS2{Float64}([1.0, 2.0, 3.0, 4.0, 5.0, 6.0,
↪  7.0]))
```

5.5 The facility location problem

This tutorial was originally contributed by Mathieu Tanneau and Alexis Montoison.

Required packages

This tutorial requires the following packages:

```
using JuMP
import HiGHS
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:

$$\min_{x,y} \quad \sum_{i,j} c_{i,j} x_{i,j} + \sum_{j} f_j y_j$$

$$s.t. \sum_{j} x_{i,j} = 1, \qquad \forall i \in M$$

$$x_{i,j} \leq y_j, \qquad \forall i \in M, j \in N$$

$$x_{i,j}, y_j \in \{0, 1\}, \qquad \forall i \in M, j \in N$$

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

To ensure reproducibility, we set the random number seed:

```
Random.seed!(314)
```

```
MersenneTwister(314)
```

Here's the data we need:

```
# Number of clients
m = 12
# Number of facility locations
n = 5

# Clients' locations
x_c, y_c = rand(m), rand(m)

# Facilities' potential locations
x_f, y_f = rand(n), 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([x_c[i] - x_f[j], y_c[i] - y_f[j]], 2)
    end
end
```

Display the data

```
Plots.scatter(
    x_c,
    y_c;
    label = "Clients",
    markershape = :circle,
    markercolor = :blue,
)
Plots.scatter!(
    x_f,
    y_f;
    label = "Facility",
    markershape = :square,
    markercolor = :white,
    markersize = 6,
    markerstrokecolor = :red,
    markerstrokewidth = 2,
)
```


JuMP implementation

Create a JuMP model

```
model = Model(HiGHS.Optimizer)
set_silent(model)
@variable(model, y[1:n], Bin);
@variable(model, x[1:m, 1:n], Bin);
# Each client is served exactly once
```

```
@constraint(model, 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(model, open_facility[i in 1:m, j in 1:n], x[i, j] <= y[j]);
@objective(model, Min, f' * y + sum(c .* x));
```

Solve the uncapacitated facility location problem with HiGHS

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

```
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_is_selected = isapprox.(value.(x), 1; atol = 1e-5);
y_is_selected = isapprox.(value.(y), 1; atol = 1e-5);

p = Plots.scatter(
    x_c,
    y_c;
    markershape = :circle,
    markercolor = :blue,
    label = nothing,
)

Plots.scatter!(
    x_f,
    y_f;
    markershape = :square,
    markercolor = [(y_is_selected[j] ? :red : :white) for j in 1:n],
    markersize = 6,
    markerstrokecolor = :red,
    markerstrokewidth = 2,
    label = nothing,
)

for i in 1:m, j in 1:n
    if x_is_selected[i, j]
        Plots.plot!(
            [x_c[i], x_f[j]],
            [y_c[i], y_f[j]];
            color = :black,
            label = nothing,
        )
    end
end

p
```

Capacitated facility location

Problem formulation

The capacitated variant introduces a capacity constraint on each facility, that is, 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

$$\min_{x,y} \quad \sum_{i,j} c_{i,j} x_{i,j} + \sum_{j} f_j y_j$$

$$s.t. \sum_{j} x_{i,j} = 1, \qquad\qquad \forall i \in M$$

$$\sum_{i} a_i x_{i,j} \leq q_j y_j, \qquad \forall j \in N$$

$$x_{i,j}, y_j \in \{0, 1\}, \qquad\qquad \forall i \in M, j \in N$$

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

We need some new data:

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

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

Display the data

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

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

JuMP implementation

Create a JuMP model

```
model = Model(HiGHS.Optimizer)
set_silent(model)
@variable(model, y[1:n], Bin);
@variable(model, x[1:m, 1:n], Bin);
# Each client is served exactly once
@constraint(model, client_service[i in 1:m], sum(x[i, :]) == 1);
# Capacity constraint
@constraint(model, capacity, x' * a .<= (q .* y));
# Objective
@objective(model, Min, f' * y + sum(c .* x));
```

Solve the problem

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

```
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_is_selected = isapprox.(value.(x), 1; atol = 1e-5);
y_is_selected = isapprox.(value.(y), 1; atol = 1e-5);
```

Display the solution

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

Plots.scatter!(
    x_f,
    y_f;
    label = nothing,
    markershape = :rect,
    markercolor = [(y_is_selected[j] ? :red : :white) for j in 1:n],
    markersize = q,
    markerstrokecolor = :red,
    markerstrokewidth = 2,
)

for i in 1:m, j in 1:n
    if x_is_selected[i, j]
        Plots.plot!(
            [x_c[i], x_f[j]],
            [y_c[i], y_f[j]];
            color = :black,
            label = nothing,
        )
    end
end

p
```

5.6   The factory schedule example

This tutorial was originally contributed by @Crghilardi.

This tutorial is a Julia translation of Part 5 from Introduction to Linear Programming with Python.

The purpose of this tutorial is to demonstrate how to use DataFrames and delimited files, and to structure your code that is robust to infeasibilities and permits running with different datasets.

Required packages

This tutorial requires the following packages:

```julia
using JuMP
import CSV
import DataFrames
import HiGHS
import StatsPlots
```

Formulation

The Factory Scheduling Problem assumes we are optimizing the production of a good from factories $f \in F$ over the course of 12 months $m \in M$.

If a factory f runs during a month m, a fixed cost of a_f is incurred, the factory must produce $x_{m,f}$ units that is within some minimum and maximum production levels l_f and u_f respectively, and each unit of production

incurs a variable cost c_f. Otherwise, the factory can be shut for the month with zero production and no fixed-cost is incurred. We denote the run/not-run decision by $z_{m,f} \in \{0,1\}$, where $z_{m,f}$ is 1 if factory f runs in month m. The factory must produce enough units to satisfy demand d_m.

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

$$
\begin{aligned}
\min \quad & \sum_{f \in F, m \in M} a_f z_{m,f} + c_f x_{m,f} \\
\text{s.t.} \quad & x_{m,f} \le u_f z_{m,f} & \forall f \in F, m \in M \\
& x_{m,f} \ge l_f z_{m,f} & \forall f \in F, m \in M \\
& \sum_{f \in F} x_{m,f} = d_m & \forall f \in F, m \in M \\
& z_{m,f} \in \{0,1\} & \forall f \in F, m \in M.
\end{aligned}
$$

However, this formulation has a problem: if demand is too high, we may be unable to satisfy the demand constraint, and the problem will be infeasible.

> **Tip**
>
> When modeling, consider ways to formulate your model such that it always has a feasible solution. This greatly simplifies debugging data errors that would otherwise result in an infeasible solution. In practice, most practical decisions have a feasible solution. In our case, we could satisfy demand (at a high cost) by buying replacement items for the buyer, or running the factories in overtime to make up the difference.

We can improve our model by adding a new variable, δ_m, which represents the quantity of unmet demand in each month m. We penalize δ_m by an arbitrarily large value of \$10,000/unit in the objective.

$$
\begin{aligned}
\min \quad & \sum_{f \in F, m \in M} a_f z_{m,f} + c_f x_{m,f} + \sum_{m \in M} 10000 \delta_m \\
\text{s.t.} \quad & x_{m,f} \le u_f z_{m,f} & \forall f \in F, m \in M \\
& x_{m,f} \ge l_f z_{m,f} & \forall f \in F, m \in M \\
& \sum_{f \in F} x_{m,f} - \delta_m = d_m & \forall f \in F, m \in M \\
& z_{m,f} \in \{0,1\} & \forall f \in F, m \in M \\
& \delta_m \ge 0 & \forall m \in M.
\end{aligned}
$$

Data

The JuMP GitHub repository contains two text files with the data we need for this tutorial.

The first file contains a dataset of our factories, A and B, with their production and cost levels for each month. For the documentation, the file is located at:

```
factories_filename = joinpath(@__DIR__, "factory_schedule_factories.txt")
```

```
"/home/runner/work/JuMP.jl/JuMP.jl/docs/latex_build/tutorials/linear/factory_schedule_factories.txt"
```

and it has the following contents:

```
print(read(factories_filename, String))
```

```
factory month min_production max_production fixed_cost variable_cost
A       1     20000          100000         500        10
A       2     20000          110000         500        11
A       3     20000          120000         500        12
A       4     20000          145000         500        9
A       5     20000          160000         500        8
A       6     20000          140000         500        8
A       7     20000          155000         500        5
A       8     20000          200000         500        7
A       9     20000          210000         500        9
A       10    20000          197000         500        10
A       11    20000          80000          500        8
A       12    20000          150000         500        8
B       1     20000          50000          600        5
B       2     20000          55000          600        4
B       3     20000          60000          600        3
B       4     20000          100000         600        5
B       5     0              0              0          0
B       6     20000          70000          600        6
B       7     20000          60000          600        4
B       8     20000          100000         600        6
B       9     20000          100000         600        8
B       10    20000          100000         600        11
B       11    20000          120000         600        10
B       12    20000          150000         600        12
```

You can reproduce this tutorial locally by saving the contents to a new file and updating `factories_filename` appropriately.

We use the `CSV` and `DataFrames` packages to read it into Julia:

```
factory_df = CSV.read(
    factories_filename,
    DataFrames.DataFrame;
    delim = ' ',
    ignorerepeated = true,
)
```

The second file contains the demand data by month:

```
demand_filename = joinpath(@__DIR__, "factory_schedule_demand.txt")

print(read(demand_filename, String))
```

	factory	month	min_production	max_production	fixed_cost	variable_cost
	String1	Int64	Int64	Int64	Int64	Int64
1	A	1	20000	100000	500	10
2	A	2	20000	110000	500	11
3	A	3	20000	120000	500	12
4	A	4	20000	145000	500	9
5	A	5	20000	160000	500	8
6	A	6	20000	140000	500	8
7	A	7	20000	155000	500	5
8	A	8	20000	200000	500	7
9	A	9	20000	210000	500	9
10	A	10	20000	197000	500	10
11	A	11	20000	80000	500	8
12	A	12	20000	150000	500	8
13	B	1	20000	50000	600	5
14	B	2	20000	55000	600	4
15	B	3	20000	60000	600	3
16	B	4	20000	100000	600	5
17	B	5	0	0	0	0
18	B	6	20000	70000	600	6
19	B	7	20000	60000	600	4
20	B	8	20000	100000	600	6
21	B	9	20000	100000	600	8
22	B	10	20000	100000	600	11
23	B	11	20000	120000	600	10
24	B	12	20000	150000	600	12

```
month demand
1      120000
2      100000
3      130000
4      130000
5      140000
6      130000
7      150000
8      170000
9      200000
10     190000
11     140000
12     100000
```

```
demand_df = CSV.read(
    demand_filename,
    DataFrames.DataFrame;
    delim = ' ',
    ignorerepeated = true,
)
```

	month	demand
	Int64	Int64
1	1	120000
2	2	100000
3	3	130000
4	4	130000
5	5	140000
6	6	130000
7	7	150000
8	8	170000
9	9	200000
10	10	190000
11	11	140000
12	12	100000

Data validation

Before moving on, it's always good practice to validate the data you read from external sources. The more effort you spend here, the fewer issues you will have later. The following function contains a few simple checks, but we could add more. For example, you might want to check that none of the values are too large (or too small), which might indicate a typo or a unit conversion issue (perhaps the variable costs are in $/1000 units instead of $/unit).

```julia
function valiate_data(
    demand_df::DataFrames.DataFrame,
    factory_df::DataFrames.DataFrame,
)
    # Minimum production must not exceed maximum production.
    @assert all(factory_df.min_production .<= factory_df.max_production)
    # Demand, minimum production, fixed costs, and variable costs must all be
    # non-negative.
    @assert all(demand_df.demand .>= 0)
    @assert all(factory_df.min_production .>= 0)
    @assert all(factory_df.fixed_cost .>= 0)
    @assert all(factory_df.variable_cost .>= 0)
    return
end

valiate_data(demand_df, factory_df)
```

JuMP formulation

Next, we need to code our JuMP formulation. As shown in Design patterns for larger models, it's always good practice to code your model in a function that accepts well-defined input and returns well-defined output.

```julia
function solve_factory_scheduling(
    demand_df::DataFrames.DataFrame,
    factory_df::DataFrames.DataFrame,
)
    # Even though we validated the data above, it's good practice to do it here
    # too.
    valiate_data(demand_df, factory_df)
```

```julia
    months, factories = unique(factory_df.month), unique(factory_df.factory)
    model = Model(HiGHS.Optimizer)
    set_silent(model)
    @variable(model, status[months, factories], Bin)
    @variable(model, production[months, factories], Int)
    @variable(model, unmet_demand[months] >= 0)
    # We use `eachrow` to loop through the rows of the dataframe and add the
    # relevant constraints.
    for r in eachrow(factory_df)
        m, f = r.month, r.factory
        @constraint(model, production[m, f] <= r.max_production * status[m, f])
        @constraint(model, production[m, f] >= r.min_production * status[m, f])
    end
    @constraint(
        model,
        [r in eachrow(demand_df)],
        sum(production[r.month, :]) + unmet_demand[r.month] == r.demand,
    )
    @objective(
        model,
        Min,
        10_000 * sum(unmet_demand) + sum(
            r.fixed_cost * status[r.month, r.factory] +
            r.variable_cost * production[r.month, r.factory] for
            r in eachrow(factory_df)
        )
    )
    optimize!(model)
    schedules = Dict{Symbol,Vector{Float64}}(
        Symbol(f) => value.(production[:, f]) for f in factories
    )
    schedules[:unmet_demand] = value.(unmet_demand)
    return (
        termination_status = termination_status(model),
        cost = objective_value(model),
        # This `select` statement re-orders the columns in the DataFrame.
        schedules = DataFrames.select(
            DataFrames.DataFrame(schedules),
            [:unmet_demand, :A, :B],
        ),
    )
end
```

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

Solution

Now we can call our `solve_factory_scheduling` function using the data we read in above.

```julia
solution = solve_factory_scheduling(demand_df, factory_df);
```

Let's see what `solution` contains:

```
solution.termination_status
```

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

```
solution.cost
```

```
1.29064e7
```

```
solution.schedules
```

	unmet_demand	A	B
	Float64	Float64	Float64
1	0.0	70000.0	50000.0
2	0.0	45000.0	55000.0
3	0.0	70000.0	60000.0
4	0.0	30000.0	100000.0
5	0.0	140000.0	-0.0
6	0.0	60000.0	70000.0
7	0.0	90000.0	60000.0
8	0.0	70000.0	100000.0
9	0.0	100000.0	100000.0
10	0.0	190000.0	-0.0
11	0.0	80000.0	60000.0
12	0.0	100000.0	-0.0

These schedules will be easier to visualize as a graph:

```
StatsPlots.groupedbar(
    Matrix(solution.schedules);
    bar_position = :stack,
    labels = ["unmet demand" "A" "B"],
    xlabel = "Month",
    ylabel = "Production",
    legend = :topleft,
    color = ["#20326c" "#4063d8" "#a0b1ec"],
)
```

Note that we don't have any unmet demand.

What happens if demand increases?

Let's run an experiment by increasing the demand by 50% in all time periods:

```
demand_df.demand .*= 1.5
```

```
12-element Vector{Int64}:
 180000
 150000
 195000
 195000
 210000
 195000
 225000
 255000
 300000
 285000
 210000
 150000
```

Now we resolve the problem:

```
high_demand_solution = solve_factory_scheduling(demand_df, factory_df);
```

and visualize the solution:

```
StatsPlots.groupedbar(
    Matrix(high_demand_solution.schedules);
    bar_position = :stack,
    labels = ["unmet demand" "A" "B"],
    xlabel = "Month",
    ylabel = "Production",
    legend = :topleft,
    color = ["#20326c" "#4063d8" "#a0b1ec"],
)
```


Uh oh, we can't satisfy all of the demand.

How sensitive is the solution to changes in variable cost?

Let's run another experiment, this time seeing how the optimal objective value changes as we vary the variable costs of each factory.

First though, let's reset the demand to it's original level:

```
demand_df.demand ./= 1.5;
```

For our experiment, we're going to scale the variable costs of both factories by a set of values from 0.0 to 1.5:

```
scale_factors = 0:0.1:1.5
```

```
0.0:0.1:1.5
```

At a high level, we're going to loop over the scale factors for A, then the scale factors for B, rescale the input data, call our solve_factory_scheduling example, and then store the optimal objective value in the following cost matrix:

```
cost = zeros(length(scale_factors), length(scale_factors));
```

Because we're modifying factory_df in-place, we need to store the original variable costs in a new column:

```
factory_df[!, :old_variable_cost] = copy(factory_df.variable_cost);
```

Then, we need a function to scale the :variable_cost column for a particular factory by a value scale:

```
function scale_variable_cost(df, factory, scale)
    rows = df.factory .== factory
    df[rows, :variable_cost] .=
        round.(Int, df[rows, :old_variable_cost] .* scale)
    return
end
```

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

Our experiment is just a nested for-loop, modifying A and B and storing the cost:

```
for (j, a) in enumerate(scale_factors)
    scale_variable_cost(factory_df, "A", a)
    for (i, b) in enumerate(scale_factors)
        scale_variable_cost(factory_df, "B", b)
        cost[i, j] = solve_factory_scheduling(demand_df, factory_df).cost
    end
end
```

Let's visualize the cost matrix:

```
StatsPlots.contour(
    scale_factors,
    scale_factors,
    cost;
    xlabel = "Scale of factory A",
    ylabel = "Scale of factory B",
)
```


What can you infer from the solution?

> **Info**
>
> The Power Systems tutorial explains a number of other ways you can structure a problem to perform a parametric analysis of the solution. In particular, you can use in-place modification to reduce the time it takes to build and solve the resulting models.

5.7 Financial modeling problems

This tutorial was 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 HiGHS
```

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(HiGHS.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:

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

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

auction = Model(HiGHS.Optimizer)
@variable(auction, y[1:6], Bin)
```

```julia
@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
```

```julia
value.(y)
```

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

5.8   Geographical clustering

This tutorial was originally contributed by Matthew Helm and Mathieu Tanneau.

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:

```julia
using JuMP
import DataFrames
import HiGHS
import LinearAlgebra
```

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

```julia
cities = DataFrames.DataFrame(
    Union{String,Float64}[
        "New York, NY" 8.405 40.7127 -74.0059
        "Los Angeles, CA" 3.884 34.0522 -118.2436
        "Chicago, IL" 2.718 41.8781 -87.6297
        "Houston, TX" 2.195 29.7604 -95.3698
        "Philadelphia, PA" 1.553 39.9525 -75.1652
        "Phoenix, AZ" 1.513 33.4483 -112.0740
        "San Antonio, TX" 1.409 29.4241 -98.4936
        "San Diego, CA" 1.355 32.7157 -117.1610
```

```
      "Dallas, TX" 1.257 32.7766 -96.7969
      "San Jose, CA" 0.998 37.3382 -121.8863
      "Austin, TX" 0.885 30.2671 -97.7430
      "Indianapolis, IN" 0.843 39.7684 -86.1580
      "Jacksonville, FL" 0.842 30.3321 -81.6556
      "San Francisco, CA" 0.837 37.7749 -122.4194
      "Columbus, OH" 0.822 39.9611 -82.9987
      "Charlotte, NC" 0.792 35.2270 -80.8431
      "Fort Worth, TX" 0.792 32.7554 -97.3307
      "Detroit, MI" 0.688 42.3314 -83.0457
      "El Paso, TX" 0.674 31.7775 -106.4424
      "Memphis, TN" 0.653 35.1495 -90.0489
    ],
    ["city", "population", "lat", "lon"],
)
```

	city	population	lat	lon
	Union…	Union…	Union…	Union…
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(HiGHS.Optimizer)
set_silent(model)
@variable(model, x[1:n, 1:k], Bin)
@constraint(model, [i = 1:n], sum(x[i, :]) == 1);
# 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 i-th 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:

```julia
@variable(model, z[i = 1:n, j = 1:i], Bin)
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.

```julia
@objective(model, Min, sum(dm[i, j] * z[i, j] for i in 1:n, j in 1:i));
```

We can then call `optimize!` and review the results.

```julia
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.

```julia
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
     │ Union…           Union…      Union…   Union…    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 = 7×5 SubDataFrame
 Row │ city             population  lat      lon       group
     │ Union…           Union…      Union…   Union…    Float64
─────┼───────────────────────────────────────────────────────
   1 │ Chicago, IL      2.718       41.8781  -87.6297      2.0
   2 │ Houston, TX      2.195       29.7604  -95.3698      2.0
   3 │ San Antonio, TX  1.409       29.4241  -98.4936      2.0
   4 │ Dallas, TX       1.257       32.7766  -96.7969      2.0
   5 │ Austin, TX       0.885       30.2671  -97.743       2.0
   6 │ Fort Worth, TX   0.792       32.7554  -97.3307      2.0
   7 │ Memphis, TN      0.653       35.1495  -90.0489      2.0

sum(group.population) = 9.909

group = 6×5 SubDataFrame
 Row │ city             population  lat      lon       group
     │ Union…           Union…      Union…   Union…    Float64
─────┼───────────────────────────────────────────────────────
   1 │ Los Angeles, CA  3.884       34.0522  -118.244      3.0
   2 │ Phoenix, AZ      1.513       33.4483  -112.074      3.0
   3 │ San Diego, CA    1.355       32.7157  -117.161      3.0
   4 │ San Jose, CA     0.998       37.3382  -121.886      3.0
   5 │ San Francisco, CA 0.837      37.7749  -122.419      3.0
   6 │ El Paso, TX      0.674       31.7775  -106.442      3.0

sum(group.population) = 9.261000000000001
```

5.9 Multi-objective knapsack

This tutorial explains how to create and solve a multi-objective linear program. In addition, it demonstrates how to work with solvers which return multiple solutions.

Required packages

This tutorial requires the following packages:

```julia
using JuMP
import HiGHS
import MultiObjectiveAlgorithms as MOA
import Plots
```

MultiObjectiveAlgorithms.jl is a package which implements a variety of algorithms for solving multi-objective optimization problems. Because it is a long package name, we import it instead as MOA.

Formulation

The knapsack problem is a classic problem in mixed-integer programming. Given a collection of items $i \in I$, each of which has an associated weight, w_i, and profit, p_i, the knapsack problem determines which profit-maximizing subset of items to pack into a knapsack such that the total weight is less than a capacity c. The mathematical formulation is:

$$\max \sum_{i \in I} p_i x_i$$
$$\text{s.t.} \ \sum_{i \in I} w_i x_i \le c$$
$$x_i \in \{0, 1\} \qquad \forall i \in I$$

where x_i is 1 if we pack item i into the knapsack and 0 otherwise.

For this tutorial, we extend the single-objective knapsack problem by adding another objective: given a desirability rating, r_i, we wish to maximize the total desirability of the items in our knapsack. Thus, our mathematical formulation is now:

$$\max \sum_{i \in I} p_i x_i$$
$$\sum_{i \in I} r_i x_i$$
$$\text{s.t.} \ \sum_{i \in I} w_i x_i \le c$$
$$x_i \in \{0, 1\} \qquad \forall i \in I$$

Data

The data for this example was taken from vOptGeneric, and the original author was @xgandibleux.

```
profit = [77, 94, 71, 63, 96, 82, 85, 75, 72, 91, 99, 63, 84, 87, 79, 94, 90]
desire = [65, 90, 90, 77, 95, 84, 70, 94, 66, 92, 74, 97, 60, 60, 65, 97, 93]
weight = [80, 87, 68, 72, 66, 77, 99, 85, 70, 93, 98, 72, 100, 89, 67, 86, 91]
capacity = 900
N = length(profit)
```

```
17
```

Comparing the capacity to the total weight of all the items:

```
capacity / sum(weight)
```

```
0.6428571428571429
```

shows that we can take approximately 64% of the items.

Plotting the items, we see that there are a range of items with different profits and desirability. Some items have a high profit and a high desirability, others have a low profit and a high desirability (and vice versa).

```
Plots.scatter(
    profit,
    desire;
    xlabel = "Profit",
    ylabel = "Desire",
    legend = false,
)
```


The goal of the bi-objective knapsack problem is to choose a subset which maximizes both objectives.

JuMP formulation

Our JuMP formulation is a direct translation of the mathematical formulation:

```
model = Model()
@variable(model, x[1:N], Bin)
@constraint(model, sum(weight[i] * x[i] for i in 1:N) <= capacity)
@expression(model, profit_expr, sum(profit[i] * x[i] for i in 1:N))
@expression(model, desire_expr, sum(desire[i] * x[i] for i in 1:N))
@objective(model, Max, [profit_expr, desire_expr])
```

```
2-element Vector{AffExpr}:
 77 x[1] + 94 x[2] + 71 x[3] + 63 x[4] + 96 x[5] + 82 x[6] + 85 x[7] + 75 x[8] + 72 x[9] + 91 x[10]
 ↪   + 99 x[11] + 63 x[12] + 84 x[13] + 87 x[14] + 79 x[15] + 94 x[16] + 90 x[17]
 65 x[1] + 90 x[2] + 90 x[3] + 77 x[4] + 95 x[5] + 84 x[6] + 70 x[7] + 94 x[8] + 66 x[9] + 92 x[10]
 ↪   + 74 x[11] + 97 x[12] + 60 x[13] + 60 x[14] + 65 x[15] + 97 x[16] + 93 x[17]
```

Note how we form a multi-objective program by passing a vector of scalar objective functions.

Solution

To solve our model, we need an optimizer which supports multi-objective linear programs. One option is to use the MultiObjectiveAlgorithms.jl package.

```
set_optimizer(model, () -> MOA.Optimizer(HiGHS.Optimizer))
set_silent(model)
```

MultiObjectiveAlgorithms supports many different algorithms for solving multiobjective optimization problems. One option is the epsilon-constraint method:

```
set_attribute(model, MOA.Algorithm(), MOA.EpsilonConstraint())
```

Let's solve the problem and see the solution

```
optimize!(model)
solution_summary(model)
```

```
* Solver : MOA[algorithm=MultiObjectiveAlgorithms.EpsilonConstraint, optimizer=HiGHS]

* Status
  Result count       : 9
  Termination status : OPTIMAL
  Message from the solver:
  "Solve complete. Found 9 solution(s)"

* Candidate solution (result #1)
  Primal status       : FEASIBLE_POINT
  Dual status         : NO_SOLUTION
  Objective value     : [9.18000e+02,9.83000e+02]
  Objective bound     : [9.55000e+02,9.83000e+02]
  Relative gap        : 0.00000e+00

* Work counters
  Solve time (sec)   : 1.38314e-01
  Simplex iterations : 6
  Barrier iterations : -1
  Node count         : 1
```

There are 9 solutions available. We can also use result_count to see how many solutions are available:

```
result_count(model)
```

```
9
```

Accessing multiple solutions

Access the nine different solutions in the model using the `result` keyword to solution_summary, value, and objective_value:

```
solution_summary(model; result = 5)
```

```
* Solver : MOA[algorithm=MultiObjectiveAlgorithms.EpsilonConstraint, optimizer=HiGHS]

* Status
  Result count      : 9
  Termination status : OPTIMAL

* Candidate solution (result #5)
  Primal status       : FEASIBLE_POINT
  Dual status         : NO_SOLUTION
  Objective value     : [9.36000e+02,9.42000e+02]
```

```
objective_value(model; result = 5)
```

```
2-element Vector{Float64}:
 936.0
 942.0
```

Note that because we set a vector of two objective functions, the objective value is a vector with two elements. We can also query the value of each objective separately:

```
value(profit_expr; result = 5)
```

```
936.0
```

Visualizing objective space

Unlike single-objective optimization problems, multi-objective optimization problems do not have a single optimal solution. Instead, the solutions returned represent possible trade-offs that the decision maker can choose between the two objectives. A common way to visualize this is by plotting the objective values of each of the solutions:

```
plot = Plots.scatter(
    [value(profit_expr; result = i) for i in 1:result_count(model)],
    [value(desire_expr; result = i) for i in 1:result_count(model)];
    xlabel = "Profit",
    ylabel = "Desire",
    title = "Objective space",
    label = "",
    xlims = (915, 960),
)
for i in 1:result_count(model)
    y = objective_value(model; result = i)
    Plots.annotate!(y[1] - 1, y[2], (i, 10))
end
ideal_point = objective_bound(model)
Plots.scatter!([ideal_point[1]], [ideal_point[2]]; label = "Ideal point")
```
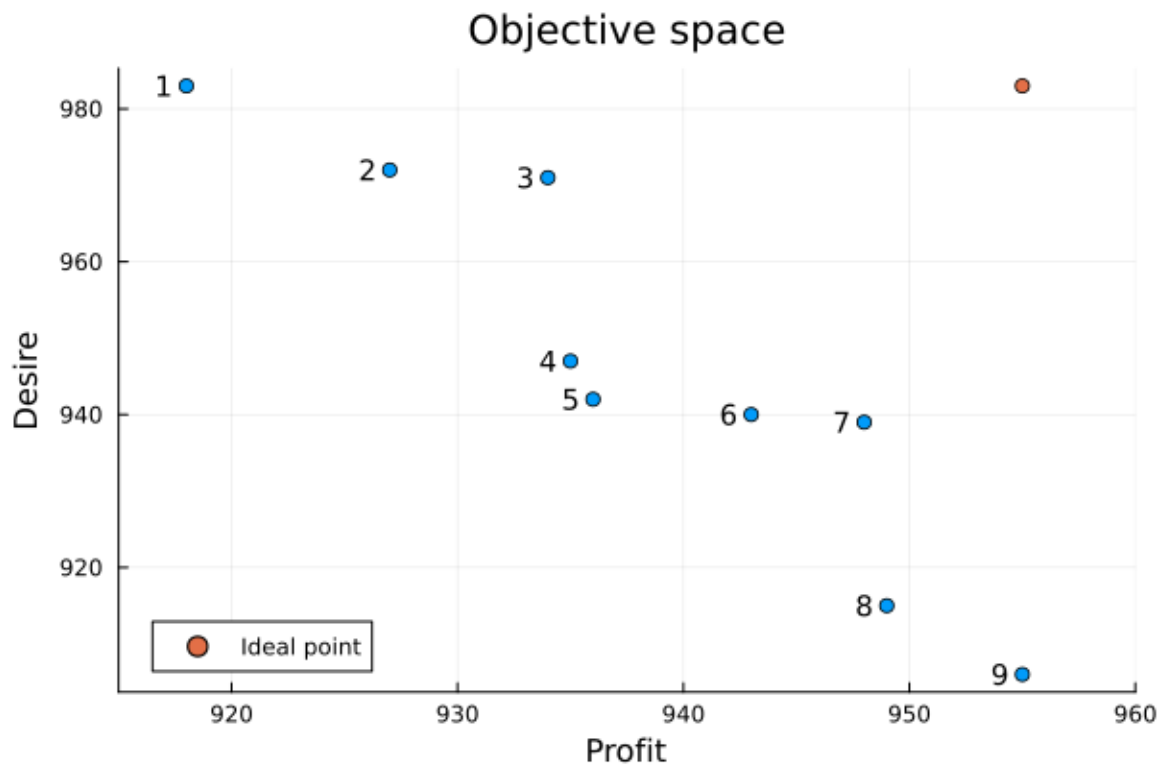

Visualizing the objective space lets the decision maker choose a solution that suits their personal preferences. For example, result #7 is close to the maximum value of profit, but offers significantly higher desirability compared with solutions #8 and #9.

The set of items that are chosen in solution #7 are:

```
items_chosen = [i for i in 1:N if value(x[i]; result = 7) > 0.9]
```

```
11-element Vector{Int64}:
  1
  2
  3
  5
  6
  8
 10
 11
 15
 16
 17
```

5.10   Simple multi-objective examples

This tutorial contains a number of examples of multi-objective programs from the literature.

Required packages

This tutorial requires the following packages:

```julia
using JuMP
import HiGHS
import MultiObjectiveAlgorithms as MOA
```

Bi-objective linear problem

This is example is taken from Example 6.3 (from Steuer, 1985), page 154 of Ehrgott, M. (2005). Multicriteria Optimization. Springer, Berlin. The code was adapted from an example in vOptGeneric by @xgandibleux.

```julia
model = Model()
set_silent(model)
@variable(model, x1 >= 0)
@variable(model, 0 <= x2 <= 3)
@objective(model, Min, [3x1 + x2, -x1 - 2x2])
@constraint(model, 3x1 - x2 <= 6)
set_optimizer(model, () -> MOA.Optimizer(HiGHS.Optimizer))
set_attribute(model, MOA.Algorithm(), MOA.Lexicographic())
optimize!(model)
solution_summary(model)
```

```
* Solver : MOA[algorithm=MultiObjectiveAlgorithms.Lexicographic, optimizer=HiGHS]

* Status
  Result count       : 2
  Termination status : OPTIMAL
  Message from the solver:
  "Solve complete. Found 2 solution(s)"

* Candidate solution (result #1)
```

```
  Primal status       : FEASIBLE_POINT
  Dual status         : NO_SOLUTION
  Objective value     : [0.00000e+00,0.00000e+00]
  Objective bound     : [0.00000e+00,-9.00000e+00]
  Relative gap        : Inf
  Dual objective value : -9.00000e+00

* Work counters
  Solve time (sec)    : 1.19209e-03
  Simplex iterations : 1
  Barrier iterations : 0
  Node count          : -1
```

```julia
for i in 1:result_count(model)
    print(i, ": z = ", round.(Int, objective_value(model; result = i)), " | ")
    println("x = ", value.([x1, x2]; result = i))
end
```

```
1: z = [0, 0] | x = [0.0, -0.0]
2: z = [12, -9] | x = [3.0, 3.0]
```

Bi-objective linear assignment problem

This is example is taken from Example 9.38 (from Ulungu and Teghem, 1994), page 255 of Ehrgott, M. (2005). Multicriteria Optimization. Springer, Berlin. The code was adapted from an example in vOptGeneric by @xgandibleux.

```julia
C1 = [5 1 4 7; 6 2 2 6; 2 8 4 4; 3 5 7 1]
C2 = [3 6 4 2; 1 3 8 3; 5 2 2 3; 4 2 3 5]
n = size(C2, 1)
model = Model()
set_silent(model)
@variable(model, x[1:n, 1:n], Bin)
@objective(model, Min, [sum(C1 .* x), sum(C2 .* x)])
@constraint(model, [i = 1:n], sum(x[i, :]) == 1)
@constraint(model, [j = 1:n], sum(x[:, j]) == 1)
set_optimizer(model, () -> MOA.Optimizer(HiGHS.Optimizer))
set_attribute(model, MOA.Algorithm(), MOA.EpsilonConstraint())
optimize!(model)
solution_summary(model)
```

```
* Solver : MOA[algorithm=MultiObjectiveAlgorithms.EpsilonConstraint, optimizer=HiGHS]

* Status
  Result count        : 6
  Termination status : OPTIMAL
  Message from the solver:
  "Solve complete. Found 6 solution(s)"
```

```
* Candidate solution (result #1)
  Primal status      : FEASIBLE_POINT
  Dual status        : NO_SOLUTION
  Objective value    : [6.00000e+00,2.40000e+01]
  Objective bound    : [6.00000e+00,7.00000e+00]
  Relative gap       : 0.00000e+00

* Work counters
  Solve time (sec)   : 8.80694e-03
  Simplex iterations : 5
  Barrier iterations : -1
  Node count         : 1
```

```
for i in 1:result_count(model)
    print(i, ": z = ", round.(Int, objective_value(model; result = i)), " | ")
    println("x = ", round.(Int, value.(x; result = i)))
end
```

```
1: z = [6, 24] | x = [0 1 0 0; 0 0 1 0; 1 0 0 0; 0 0 0 1]
2: z = [9, 17] | x = [0 0 1 0; 0 1 0 0; 1 0 0 0; 0 0 0 1]
3: z = [12, 13] | x = [1 0 0 0; 0 1 0 0; 0 0 1 0; 0 0 0 1]
4: z = [16, 11] | x = [0 0 0 1; 0 1 0 0; 0 0 1 0; 1 0 0 0]
5: z = [19, 10] | x = [0 0 1 0; 1 0 0 0; 0 0 0 1; 0 1 0 0]
6: z = [22, 7] | x = [0 0 0 1; 1 0 0 0; 0 0 1 0; 0 1 0 0]
```

Bi-objective shortest path problem

This is example is taken from Exercise 9.5 page 269 of Ehrgott, M. (2005). Multicriteria Optimization. Springer, Berlin. The code was adapted from an example in vOptGeneric by @xgandibleux.

```
M = 50
C1 = [
    M 4 5 M M M
    M M 2 1 2 7
    M M M 5 2 M
    M M 5 M M 3
    M M M M M 4
    M M M M M M
]
C2 = [
    M 3 1 M M M
    M M 1 4 2 2
    M M M 1 7 M
    M M 1 M M 2
    M M M M M 2
    M M M M M M
]
n = size(C2, 1)
model = Model()
```

```julia
set_silent(model)
@variable(model, x[1:n, 1:n], Bin)
@objective(model, Min, [sum(C1 .* x), sum(C2 .* x)])
@constraint(model, sum(x[1, :]) == 1)
@constraint(model, sum(x[:, n]) == 1)
@constraint(model, [i = 2:n-1], sum(x[i, :]) - sum(x[:, i]) == 0)
set_optimizer(model, () -> MOA.Optimizer(HiGHS.Optimizer))
set_attribute(model, MOA.Algorithm(), MOA.EpsilonConstraint())
optimize!(model)
solution_summary(model)
```

```
* Solver : MOA[algorithm=MultiObjectiveAlgorithms.EpsilonConstraint, optimizer=HiGHS]

* Status
  Result count       : 4
  Termination status : OPTIMAL
  Message from the solver:
  "Solve complete. Found 4 solution(s)"

* Candidate solution (result #1)
  Primal status      : FEASIBLE_POINT
  Dual status        : NO_SOLUTION
  Objective value    : [8.00000e+00,9.00000e+00]
  Objective bound    : [8.00000e+00,4.00000e+00]
  Relative gap       : 0.00000e+00

* Work counters
  Solve time (sec)   : 4.79102e-03
  Simplex iterations : 4
  Barrier iterations : -1
  Node count         : 1
```

```julia
for i in 1:result_count(model)
    print(i, ": z = ", round.(Int, objective_value(model; result = i)), " | ")
    X = round.(Int, value.(x; result = i))
    print("Path:")
    for ind in findall(val -> val ≈ 1, X)
        i, j = ind.I
        print(" $i->$j")
    end
    println()
end
```

```
1: z = [8, 9] | Path: 1->2 2->4 4->6
2: z = [10, 7] | Path: 1->2 2->5 5->6
3: z = [11, 5] | Path: 1->2 2->6
4: z = [13, 4] | Path: 1->3 3->4 4->6
```

5.11   The knapsack problem example

The purpose of this example is demonstrate how to formulate and solve a simple optimization problem. We use the knapsack problem for this purpose.

Required packages

This tutorial requires the following packages:

```julia
using JuMP
import HiGHS
import Test
```

Formulation

The simple knapsack problem is a well-known type of optimization problem: given a set of items and a container with a fixed capacity, choose a subset of items having the greatest combined value that will fit within the container without exceeding the capacity. The name of the problem suggests its analogy to packing for a trip, where the baggage weight limit is the capacity and the goal is to pack the most profitable combination of belongings.

We can formulate the problem as an integer linear program

$$\max \sum_{i=1}^{n} c_i x_i$$
$$s.t. \sum_{i=1}^{n} w_i x_i \leq C,$$
$$x_i \in \{0, 1\}, \quad \forall i = 1, \ldots, n$$

or more compactly as

$$\max c^\top x$$
$$s.t. \, w^\top x \leq C$$
$$x \text{ binary },$$

where there is a choice between n items, with item i having weight w_i, profit c_i and a decision variable x_i equal to 1 if the item is chosen and 0 if not. The capacity is a single real number C of the same number type as the individual weights.

Data

The data for the problem consists of two vectors (one for the profits and one for the weights) along with a capacity. For our example, we use a capacity of 10 units

```julia
capacity = 10;
```

and the vector data

```
profit = [5, 3, 2, 7, 4];
weight = [2, 8, 4, 2, 5];
```

JuMP formulation

Let's begin constructing the JuMP model for our knapsack problem. First, we'll create a `Model` object for holding model elements as we construct each part. We'll also set the solver that will ultimately be called to solve the model, once it's constructed.

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

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

Next we need the decision variables representing which items are chosen.

```
@variable(model, x[1:5], Bin)
```

```
5-element Vector{VariableRef}:
 x[1]
 x[2]
 x[3]
 x[4]
 x[5]
```

We now want to constrain those variables so that their combined weight is less than or equal to the given capacity.

```
@constraint(model, sum(weight[i] * x[i] for i in 1:5) <= capacity)
```

$$2x_1 + 8x_2 + 4x_3 + 2x_4 + 5x_5 \le 10$$

Finally, we set an objective: maximise the combined profit of the selected items.

```
@objective(model, Max, sum(profit[i] * x[i] for i in 1:5))
```

$$5x_1 + 3x_2 + 2x_3 + 7x_4 + 4x_5$$

Let's print a human-readable description of the model and check that the model looks as expected:

```
print(model)
```

```
Max 5 x[1] + 3 x[2] + 2 x[3] + 7 x[4] + 4 x[5]
Subject to
 2 x[1] + 8 x[2] + 4 x[3] + 2 x[4] + 5 x[5] ≤ 10
 x[1] binary
 x[2] binary
 x[3] binary
 x[4] binary
 x[5] binary
```

We can now solve the optimization problem and inspect the results.

```
optimize!(model)
solution_summary(model)
```

```
* Solver : HiGHS

* Status
  Result count       : 1
  Termination status : OPTIMAL
  Message from the solver:
  "kHighsModelStatusOptimal"

* Candidate solution (result #1)
  Primal status       : FEASIBLE_POINT
  Dual status         : NO_SOLUTION
  Objective value     : 1.60000e+01
  Objective bound     : 1.60000e+01
  Relative gap        : 0.00000e+00

* Work counters
  Solve time (sec)   : 5.70059e-04
  Simplex iterations : 1
  Barrier iterations : -1
  Node count         : 1
```

The items chosen are

```
items_chosen = [i for i in 1:5 if value(x[i]) > 0.5]
```

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

After working interactively, it is good practice to implement your model in a function. A function can be used to ensure that the model is given well-defined input data by validation checks, and that the solution process went as expected.

```julia
function solve_knapsack_problem(;
    profit,
    weight::Vector{T},
    capacity::T,
) where {T<:Real}
    N = length(weight)

    # The profit and weight vectors must be of equal length.
    @assert length(profit) == N

    model = Model(HiGHS.Optimizer)
    set_silent(model)

    # Declare the decision variables as binary (0 or 1).
    @variable(model, x[1:N], Bin)

    # Objective: maximize profit.
    @objective(model, Max, profit' * x)

    # Constraint: can carry all items.
    @constraint(model, weight' * x <= capacity)

    # Solve problem using MIP solver
    optimize!(model)
    println("Objective is: ", objective_value(model))
    println("Solution is:")
    for i in 1:N
        print("x[$i] = ", Int(value(x[i])))
        println(", c[$i]/w[$i] = ", profit[i] / weight[i])
    end
    Test.@test termination_status(model) == OPTIMAL
    Test.@test primal_status(model) == FEASIBLE_POINT
    Test.@test objective_value(model) == 16.0
    return
end

solve_knapsack_problem(; profit = profit, weight = weight, capacity = capacity)
```

```
Objective is: 16.0
Solution is:
x[1] = 1, c[1]/w[1] = 2.5
x[2] = 0, c[2]/w[2] = 0.375
x[3] = 0, c[3]/w[3] = 0.5
x[4] = 1, c[4]/w[4] = 3.5
x[5] = 1, c[5]/w[5] = 0.8
```

We observe that the chosen items (1, 4, and 5) have the best profit to weight ratio in this particular example.

5.12 The multi-commodity flow problem

This tutorial was originally contributed by Louis Luangkesorn.

This tutorial is a JuMP implementation of the multi-commodity transportation model described in AMPL: A Modeling Language for Mathematical Programming, by R. Fourer, D.M. Gay and B.W. Kernighan.

The purpose of this tutorial is to demonstrate creating a JuMP model from an SQLite database.

Required packages

This tutorial uses the following packages

```julia
using JuMP
import DataFrames
import HiGHS
import SQLite
import Tables

const DBInterface = SQLite.DBInterface
```

```
DBInterface
```

Formulation

The multi-commondity flow problem is a simple extension of The transportation problem to multiple types of products. Briefly, we start with the formulation of the transportation problem:

$$
\begin{aligned}
\min \quad & \sum_{i \in O, j \in D} c_{i,j} x_{i,j} \\
s.t. \quad & \sum_{j \in D} x_{i,j} \leq s_i && \forall i \in O \\
& \sum_{i \in O} x_{i,j} = d_j && \forall j \in D \\
& x_{i,j} \geq 0 && \forall i \in O, j \in D
\end{aligned}
$$

but introduce a set of products P, resulting in:

$$
\begin{aligned}
\min \quad & \sum_{i \in O, j \in D, k \in P} c_{i,j,k} x_{i,j,k} \\
s.t. \quad & \sum_{j \in D} x_{i,j,k} \leq s_{i,k} && \forall i \in O, k \in P \\
& \sum_{i \in O} x_{i,j,k} = d_{j,k} && \forall j \in D, k \in P \\
& x_{i,j,k} \geq 0 && \forall i \in O, j \in D, k \in P \\
& \sum_{k \in P} x_{i,j,k} \leq u_{i,j} && \forall i \in O, j \in D
\end{aligned}
$$

Note that the last constraint is new; it says that there is a maximum quantity of goods (of any type) that can be transported from origin i to destination j.

Data

For the purpose of this tutorial, the JuMP repository contains an example database called `multi.sqlite`:

```
db = SQLite.DB(joinpath(@__DIR__, "multi.sqlite"))
```

```
SQLite.DB("/home/runner/work/JuMP.jl/JuMP.jl/docs/latex_build/tutorials/linear/multi.sqlite")
```

A quick way to see the schema of the database is via `SQLite.tables`:

```
SQLite.tables(db)
```

```
5-element Vector{SQLite.DBTable}:
 SQLite.DBTable("locations", Tables.Schema:
 :location   Union{Missing, String}
 :type       Union{Missing, String})
 SQLite.DBTable("products", Tables.Schema:
 :product   Union{Missing, String})
 SQLite.DBTable("supply", Tables.Schema:
 :origin    Union{Missing, String}
 :product   Union{Missing, String}
 :supply    Union{Missing, Float64})
 SQLite.DBTable("demand", Tables.Schema:
 :destination   Union{Missing, String}
 :product       Union{Missing, String}
 :demand        Union{Missing, Float64})
 SQLite.DBTable("cost", Tables.Schema:
 :origin        Union{Missing, String}
 :destination   Union{Missing, String}
 :product       Union{Missing, String}
 :cost          Union{Missing, Float64})
```

We interact with the database by executing queries, and then piping the results to an appropriate table. One example is a `DataFrame`:

```
DBInterface.execute(db, "SELECT * FROM locations") |> DataFrames.DataFrame
```

But other table types are supported, such as `Tables.rowtable`:

```
DBInterface.execute(db, "SELECT * FROM locations") |> Tables.rowtable
```

	location	type
	String	String
1	GARY	origin
2	CLEV	origin
3	PITT	origin
4	FRA	destination
5	DET	destination
6	LAN	destination
7	WIN	destination
8	STL	destination
9	FRE	destination
10	LAF	destination

```
10-element Vector{NamedTuple{(:location, :type), Tuple{String, String}}}:
 (location = "GARY", type = "origin")
 (location = "CLEV", type = "origin")
 (location = "PITT", type = "origin")
 (location = "FRA", type = "destination")
 (location = "DET", type = "destination")
 (location = "LAN", type = "destination")
 (location = "WIN", type = "destination")
 (location = "STL", type = "destination")
 (location = "FRE", type = "destination")
 (location = "LAF", type = "destination")
```

A rowtable is a Vector of NamedTuples.

You can construct more complicated SQL queries:

```
origins =
    DBInterface.execute(
        db,
        "SELECT location FROM locations WHERE type = \"origin\"",
    ) |> Tables.rowtable
```

```
3-element Vector{NamedTuple{(:location,), Tuple{String}}}:
 (location = "GARY",)
 (location = "CLEV",)
 (location = "PITT",)
```

But for our purpose, we just want the list of strings:

```
origins = map(y -> y.location, origins)
```

```
3-element Vector{String}:
 "GARY"
```

```
 "CLEV"
 "PITT"
```

We can compose these two operations to get a list of destinations:

```
destinations =
    DBInterface.execute(
        db,
        "SELECT location FROM locations WHERE type = \"destination\"",
    ) |>
    Tables.rowtable |>
    x -> map(y -> y.location, x)
```

```
7-element Vector{String}:
 "FRA"
 "DET"
 "LAN"
 "WIN"
 "STL"
 "FRE"
 "LAF"
```

And a list of products from our `products` table:

```
products =
    DBInterface.execute(db, "SELECT product FROM products") |>
    Tables.rowtable |>
    x -> map(y -> y.product, x)
```

```
3-element Vector{String}:
 "bands"
 "coils"
 "plate"
```

JuMP formulation

We start by creating a model and our decision variables:

```
model = Model(HiGHS.Optimizer)
set_silent(model)
@variable(model, x[origins, destinations, products] >= 0)
```

```
3-dimensional DenseAxisArray{VariableRef,3,...} with index sets:
    Dimension 1, ["GARY", "CLEV", "PITT"]
    Dimension 2, ["FRA", "DET", "LAN", "WIN", "STL", "FRE", "LAF"]
```

```
    Dimension 3, ["bands", "coils", "plate"]
And data, a 3×7×3 Array{VariableRef, 3}:
[:, :, "bands"] =
 x[GARY,FRA,bands]  x[GARY,DET,bands]  …  x[GARY,LAF,bands]
 x[CLEV,FRA,bands]  x[CLEV,DET,bands]     x[CLEV,LAF,bands]
 x[PITT,FRA,bands]  x[PITT,DET,bands]     x[PITT,LAF,bands]

[:, :, "coils"] =
 x[GARY,FRA,coils]  x[GARY,DET,coils]  …  x[GARY,LAF,coils]
 x[CLEV,FRA,coils]  x[CLEV,DET,coils]     x[CLEV,LAF,coils]
 x[PITT,FRA,coils]  x[PITT,DET,coils]     x[PITT,LAF,coils]

[:, :, "plate"] =
 x[GARY,FRA,plate]  x[GARY,DET,plate]  …  x[GARY,LAF,plate]
 x[CLEV,FRA,plate]  x[CLEV,DET,plate]     x[CLEV,LAF,plate]
 x[PITT,FRA,plate]  x[PITT,DET,plate]     x[PITT,LAF,plate]
```

One approach when working with databases is to extract all of the data into a Julia datastructure. For example, let's pull the cost table into a DataFrame and then construct our objective by iterating over the rows of the DataFrame:

```
cost = DBInterface.execute(db, "SELECT * FROM cost") |> DataFrames.DataFrame
@objective(
    model,
    Max,
    sum(r.cost * x[r.origin, r.destination, r.product] for r in eachrow(cost)),
);
```

If we don't want to use a DataFrame, we can use a Tables.rowtable instead:

```
supply = DBInterface.execute(db, "SELECT * FROM supply") |> Tables.rowtable
for r in supply
    @constraint(model, sum(x[r.origin, :, r.product]) <= r.supply)
end
```

Another approach is to execute the query, and then to iterate through the rows of the query using Tables.rows:

```
demand = DBInterface.execute(db, "SELECT * FROM demand")
for r in Tables.rows(demand)
    @constraint(model, sum(x[:, r.destination, r.product]) == r.demand)
end
```

> **Warning**
>
> Iterating through the rows of a query result works by incrementing a cursor inside the database. As a consequence, you cannot call Tables.rows twice on the same query result.

The SQLite queries can be arbitrarily complex. For example, here's a query which builds every possible origin-destination pair:

```
od_pairs = DBInterface.execute(
    db,
    """
    SELECT a.location as 'origin',
           b.location as 'destination'
    FROM locations a
    INNER JOIN locations b
    ON a.type = 'origin' AND b.type = 'destination'
    """,
)
```

```
SQLite.Query{false}(SQLite.Stmt(SQLite.DB("/home/runner/work/JuMP.jl/JuMP.jl/docs/latex_build/tutorials/linear/multi.s
↪  Base.RefValue{Ptr{SQLite.C.sqlite3_stmt}}(Ptr{SQLite.C.sqlite3_stmt} @0x00000000378f8cd8),
↪  Dict{Int64, Any}()), Base.RefValue{Int32}(100), [:origin, :destination], Type[Union{Missing,
↪  String}, Union{Missing, String}], Dict(:origin => 1, :destination => 2),
↪  Base.RefValue{Int64}(0))
```

With a constraint that we cannot send more than 625 units between each pair:

```
for r in Tables.rows(od_pairs)
    @constraint(model, sum(x[r.origin, r.destination, :]) <= 625)
end
```

Solution

Finally, we can optimize the model:

```
optimize!(model)
solution_summary(model)
```

```
* Solver : HiGHS

* Status
  Result count       : 1
  Termination status : OPTIMAL
  Message from the solver:
  "kHighsModelStatusOptimal"

* Candidate solution (result #1)
  Primal status      : FEASIBLE_POINT
  Dual status        : FEASIBLE_POINT
  Objective value    : 2.25700e+05
  Objective bound    : 2.25700e+05
  Relative gap       : Inf
  Dual objective value : 2.25700e+05

* Work counters
  Solve time (sec)   : 9.16004e-04
  Simplex iterations : 54
```

```
  Barrier iterations : 0
  Node count         : -1
```

and print the solution:

```
begin
    println("          ", join(products, ' '))
    for o in origins, d in destinations
        v = lpad.([round(Int, value(x[o, d, p])) for p in products], 5)
        println(o, " ", d, " ", join(replace.(v, "    0" => "    . "), " "))
    end
end
```

```
          bands coils plate
GARY FRA    25   500   100
GARY DET   125    .     50
GARY LAN    .     .      .
GARY WIN    .     .     50
GARY STL   250   300    .
GARY FRE    .     .      .
GARY LAF    .     .      .
CLEV FRA   275    .      .
CLEV DET   100   200    50
CLEV LAN   100    .      .
CLEV WIN    .     .      .
CLEV STL    .    625     .
CLEV FRE   225   400     .
CLEV LAF    .    375   250
PITT FRA    .     .      .
PITT DET    75   550     .
PITT LAN    .    400     .
PITT WIN    75   250     .
PITT STL   400    25   200
PITT FRE    .    450   100
PITT LAF   250   125     .
```

5.13 N-Queens

This tutorial was originally contributed by Matthew Helm and Mathieu Tanneau.

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 HiGHS
import LinearAlgebra
```
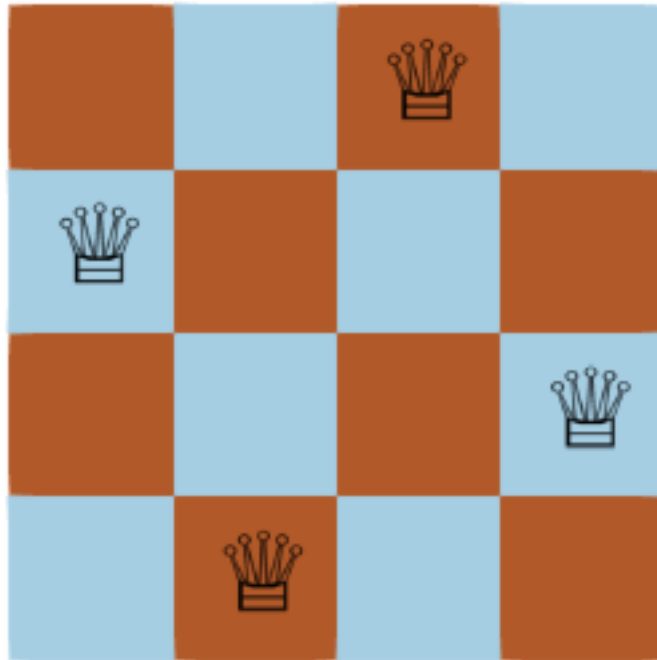
Figure 5.1: Four Queens

N-Queens

```
N = 8

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

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:

```
@variable(model, x[1:N, 1:N], Bin);
```

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
```

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 = round.(Int, value.(x))
```

```
8×8 Matrix{Int64}:
 0  0  0  0  1  0  0  0
 0  0  1  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  1  0
 1  0  0  0  0  0  0  0
 0  0  0  0  0  1  0  0
 0  1  0  0  0  0  0  0
```

5.14   Sensitivity analysis of a linear program

This tutorial explains how to use the `lp_sensitivity_report` function to create sensitivity reports like those that are produced by the Excel Solver. This is most often used in introductory classes to linear programming.

In brief, sensitivity analysis of a linear program is about asking two questions:

1. Given an optimal solution, how much can the objective coefficients change by before a different solution becomes optimal?

2. Given an optimal solution, how much can the right-hand side of a linear constraint change by before a different solution becomes optimal?

JuMP provides a function, `lp_sensitivity_report`, to help us compute these values, but this tutorial extends that to create more informative tables in the form of a `DataFrame`.

Setup

This tutorial uses the following packages:

```
using JuMP
import HiGHS
import DataFrames
```

as well as this small linear program:

```julia
model = Model(HiGHS.Optimizer)
@variable(model, x >= 0)
@variable(model, 0 <= y <= 3)
@variable(model, z <= 1)
@objective(model, Min, 12x + 20y - z)
@constraint(model, c1, 6x + 8y >= 100)
@constraint(model, c2, 7x + 12y >= 120)
@constraint(model, c3, x + y <= 20)
optimize!(model)
solution_summary(model; verbose = true)
```

```
* Solver : HiGHS

* Status
  Result count       : 1
  Termination status : OPTIMAL
  Message from the solver:
  "kHighsModelStatusOptimal"

* Candidate solution (result #1)
  Primal status      : FEASIBLE_POINT
  Dual status        : FEASIBLE_POINT
  Objective value    : 2.04000e+02
  Objective bound    : 0.00000e+00
  Relative gap       : Inf
  Dual objective value : 2.04000e+02
  Primal solution :
    x : 1.50000e+01
    y : 1.25000e+00
    z : 1.00000e+00
  Dual solution :
    c1 : 2.50000e-01
    c2 : 1.50000e+00
    c3 : 0.00000e+00

* Work counters
  Solve time (sec)   : 3.59774e-04
  Simplex iterations : 2
  Barrier iterations : 0
  Node count         : -1
```

Can you identify:

- The objective coefficient of each variable?

- The right-hand side of each constraint?

- The optimal primal and dual solutions?

Sensitivity reports

Now let's call lp_sensitivity_report:

```
report = lp_sensitivity_report(model)
```

```
SensitivityReport{Float64}(Dict{ConstraintRef, Tuple{Float64, Float64}}(y ≥ 0 => (-Inf, 1.25), y ≤
↪  3 => (-1.75, Inf), z ≤ 1 => (-Inf, Inf), c3 : x + y ≤ 20 => (-3.75, Inf), c1 : 6 x + 8 y ≥ 100
↪  => (-4.0, 2.857142857142857), c2 : 7 x + 12 y ≥ 120 => (-3.3333333333333335,
↪  4.666666666666667), x ≥ 0 => (-Inf, 15.0)), Dict{VariableRef, Tuple{Float64, Float64}}(y =>
↪  (-4.0, 0.5714285714285714), x => (-0.3333333333333333, 3.0), z => (-Inf, 1.0)))
```

It returns 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 an 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.

For example:

```
report[x]
```

```
(-0.3333333333333333, 3.0)
```

indicates that the objective coefficient on x, that is, 12, can decrease by -0.333 or increase by 3.0 and the primal solution (15, 1.25) will remain optimal. In addition:

```
report[c1]
```

```
(-4.0, 2.857142857142857)
```

means that the right-hand side of the c1 constraint (100), can decrease by 4 units, or increase by 2.85 units, and the primal solution (15, 1.25) will remain optimal.

Variable sensitivity

By themselves, the tuples aren't informative. Let's put them in context by collating a range of other information about a variable:

```julia
function variable_report(xi)
    return (
        name = name(xi),
        lower_bound = has_lower_bound(xi) ? lower_bound(xi) : -Inf,
        value = value(xi),
        upper_bound = has_upper_bound(xi) ? upper_bound(xi) : Inf,
        reduced_cost = reduced_cost(xi),
        obj_coefficient = coefficient(objective_function(model), xi),
        allowed_decrease = report[xi][1],
        allowed_increase = report[xi][2],
    )
end
```

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

Calling our function on x:

```julia
x_report = variable_report(x)
```

```
(name = "x", lower_bound = 0.0, value = 15.0, upper_bound = Inf, reduced_cost = 0.0,
↪   obj_coefficient = 12.0, allowed_decrease = -0.3333333333333333, allowed_increase = 3.0)
```

That's a bit hard to read, so let's call this on every variable in the model and put things into a DataFrame:

```julia
variable_df =
    DataFrames.DataFrame(variable_report(xi) for xi in all_variables(model))
```

	name	lower_bound	value	upper_bound	reduced_cost	obj_coefficient	allowed_decrease	allowed_increase
	String	Float64	Float64	Float64	Float64	Float64	Float64	Float64
1	x	0.0	15.0	Inf	0.0	12.0	-0.333333	3.0
2	y	0.0	1.25	3.0	0.0	20.0	-4.0	0.571429
3	z	-Inf	1.0	1.0	-1.0	-1.0	-Inf	1.0

Constraint sensitivity

We can do something similar with constraints:

```julia
function constraint_report(c::ConstraintRef)
    return (
        name = name(c),
        value = value(c),
        rhs = normalized_rhs(c),
        slack = normalized_rhs(c) - value(c),
        shadow_price = shadow_price(c),
        allowed_decrease = report[c][1],
        allowed_increase = report[c][2],
```

```
    )
end
```

```
c1_report = constraint_report(c1)
```

```
(name = "c1", value = 100.0, rhs = 100.0, slack = 0.0, shadow_price = -0.25, allowed_decrease =
↪   -4.0, allowed_increase = 2.857142857142857)
```

That's a bit hard to read, so let's call this on every variable in the model and put things into a DataFrame:

```
constraint_df = DataFrames.DataFrame(
    constraint_report(ci) for (F, S) in list_of_constraint_types(model) for
    ci in all_constraints(model, F, S) if F == AffExpr
)
```

	name	value	rhs	slack	shadow_price	allowed_decrease	allowed_increase
	String	Float64	Float64	Float64	Float64	Float64	Float64
1	c1	100.0	100.0	0.0	-0.25	-4.0	2.85714
2	c2	120.0	120.0	0.0	-1.5	-3.33333	4.66667
3	c3	16.25	20.0	3.75	0.0	-3.75	Inf

Analysis questions

Now we can use these dataframes to ask questions of the solution.

For example, we can find basic variables by looking for variables with a reduced cost of 0:

```
basic = filter(row -> iszero(row.reduced_cost), variable_df)
```

	name	lower_bound	value	upper_bound	reduced_cost	obj_coefficient	allowed_decrease	allowed_increase
	String	Float64	Float64	Float64	Float64	Float64	Float64	Float64
1	x	0.0	15.0	Inf	0.0	12.0	-0.333333	3.0
2	y	0.0	1.25	3.0	0.0	20.0	-4.0	0.571429

and non-basic variables by looking for non-zero reduced costs:

```
non_basic = filter(row -> !iszero(row.reduced_cost), variable_df)
```

	name	lower_bound	value	upper_bound	reduced_cost	obj_coefficient	allowed_decrease	allowed_increase
	String	Float64	Float64	Float64	Float64	Float64	Float64	Float64
1	z	-Inf	1.0	1.0	-1.0	-1.0	-Inf	1.0

we can also find constraints that are binding by looking for zero slacks:

```
binding = filter(row -> iszero(row.slack), constraint_df)
```

	name	value	rhs	slack	shadow_price	allowed_decrease	allowed_increase
	String	Float64	Float64	Float64	Float64	Float64	Float64
1	c1	100.0	100.0	0.0	-0.25	-4.0	2.85714
2	c2	120.0	120.0	0.0	-1.5	-3.33333	4.66667

or non-zero shadow prices:

```
binding2 = filter(row -> !iszero(row.shadow_price), constraint_df)
```

	name	value	rhs	slack	shadow_price	allowed_decrease	allowed_increase
	String	Float64	Float64	Float64	Float64	Float64	Float64
1	c1	100.0	100.0	0.0	-0.25	-4.0	2.85714
2	c2	120.0	120.0	0.0	-1.5	-3.33333	4.66667

5.15   Network flow problems

This tutorial was 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.

This tutorial requires the following packages:

```
using JuMP
import HiGHS
```

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.

$$
\begin{aligned}
\min \quad & \sum_{\forall e(i,j) \in E} a_{i,j} \times x_{i,j} \\
s.t. \quad & \sum_j x_{ij} - \sum_k x_{ki} = b_i \qquad \forall i \\
& x_e \in \{0,1\} \quad \forall e \in E
\end{aligned}
$$

where b_i is 1 if i is the starting node, -1 if i is the ending node, and 0 otherwise.

```
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]
b = [1, -1, 0, 0, 0]
shortest_path = Model(HiGHS.Optimizer)
set_silent(shortest_path)
@variable(shortest_path, x[1:n, 1:n], Bin)
# 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)
# Flow conservation constraint
@constraint(shortest_path, [i = 1:n], sum(x[i, :]) - sum(x[:, i]) == b[i],)
@objective(shortest_path, Min, sum(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.

$$\max \quad \sum_{(i,j)\in A} a_{i,j} \times y_{i,j}$$

$$s.t. \quad \sum_{\{j|(i,j)\in A\}} y_{i,j} = 1 \qquad \forall i = \{1, 2....n\}$$

$$\sum_{\{i|(i,j)\in A\}} y_{i,j} = 1 \qquad \forall j = \{1, 2....n\}$$

$$y_{i,j} \in \{0, 1\} \quad \forall (i, j) \in \{1, 2...k\}$$

```
G = [
    6 4 5 0
    0 3 6 0
    5 0 4 3
    7 5 5 5
]
n = size(G)[1]
assignment = Model(HiGHS.Optimizer)
set_silent(assignment)
@variable(assignment, y[1:n, 1:n], Bin)
# One person can only be assigned to one object
@constraint(assignment, [i = 1:n], sum(y[:, i]) == 1)
# One object can only be assigned to one person
@constraint(assignment, [j = 1:n], sum(y[j, :]) == 1)
@objective(assignment, Max, sum(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.

$$\max \qquad \sum_{v:(s,v)\in E} f(s,v)$$

$$s.t. \quad \sum_{u:(u,v)\in E} f(u,v) = \sum_{w:(v,w)\in E} f(v,w) \quad \forall v \in V - \{s,t\}$$

$$f(u,v) \leq c(u,v) \qquad \forall (u,v) \in E$$

$$f(u,v) \geq 0 \qquad \forall (u,v) \in E$$

```julia
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(HiGHS.Optimizer)
@variable(max_flow, f[1:n, 1:n] >= 0)
# Capacity constraints
@constraint(max_flow, [i = 1:n, j = 1:n], f[i, j] <= G[i, j])
# 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
```

```julia
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   2.0   1.0   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   1.0  -0.0  -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   2.0
  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
```

5.16 Sudoku

This tutorial was originally contributed by Iain Dunning.

Figure 5.2: Partially solved Sudoku

Sudoku is a popular number puzzle. The goal is to place the digits 1 to 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.

Mixed-integer linear programming formulation

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 HiGHS
```

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

```
sudoku = Model(HiGHS.Optimizer)
set_silent(sudoku)
```

Create our variables

```
@variable(sudoku, x[i = 1:9, j = 1:9, k = 1:9], Bin);
```

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, for example, 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);
```

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 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:

Constraint programming formulation

We can also model this problem using constraint programming and the all-different constraint, which says that no two elements of a vector can take the same value.

Because of the reformulation system in MathOptInterface, we can still solve this problem using HiGHS.

```
model = Model(HiGHS.Optimizer)
set_silent(model)
# HiGHS v1.2 has a bug in presolve which causes the problem to be classified as
# infeasible.
set_attribute(model, "presolve", "off")
```

Instead of the binary variables, we directly define a 9x9 grid of integer values between 1 and 9:

```
@variable(model, 1 <= x[1:9, 1:9] <= 9, Int);
```

Then, we enforce that the values in each row must be all-different:

```
@constraint(model, [i = 1:9], x[i, :] in MOI.AllDifferent(9));
```

That the values in each column must be all-different:

Figure 5.3: Solved Sudoku

```
@constraint(model, [j = 1:9], x[:, j] in MOI.AllDifferent(9));
```

And that the values in each 3x3 sub-grid must be all-different:

```
for i in (0, 3, 6), j in (0, 3, 6)
    @constraint(model, vec(x[i.+(1:3), j.+(1:3)]) in MOI.AllDifferent(9))
end
```

Finally, as before we set the initial solution and optimize:

```
for i in 1:9, j in 1:9
    if init_sol[i, j] != 0
        fix(x[i, j], init_sol[i, j]; force = true)
    end
end
```

```
optimize!(model)
```

Display the solution

```
csp_sol = round.(Int, value.(x))
```

```
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 same as we found before:

```
sol == csp_sol
```

```
true
```

5.17   The transportation problem

This tutorial was originally contributed by Louis Luangkesorn.

This tutorial is an adaptation of the transportation problem described in AMPL: A Modeling Language for Mathematical Programming, by R. Fourer, D.M. Gay and B.W. Kernighan.

The purpose of this tutorial is to demonstrate how to create a JuMP model from an ad-hoc structured text file.

Required packages

This tutorial uses the following packages:

```
using JuMP
import DelimitedFiles
import HiGHS
```

Formulation

Suppose that we have a set of factories that produce pogo sticks, and a set of retail stores in which to sell them. Each factory has a maximum number of pogo sticks that it can produce, and each retail store has a demand of pogo sticks that it can sell.

In the transportation problem, we want to choose the number of pogo sticks to make and ship from each factory to each retail store that minimizes the total shipping cost.

Mathematically, we represent our set of factories by a set of origins $i \in O$ and our retail stores by a set of destinations $j \in D$. The maximum supply at each factory is s_i and the demand from each retail store is d_j. The cost of shipping one pogo stick from i to j is $c_{i,j}$.

With a little effort, we can model the transportation problem as the following linear program:

$$
\begin{aligned}
\min \quad & \sum_{i \in O, j \in D} c_{i,j} x_{i,j} \\
s.t. \quad & \sum_{j \in D} x_{i,j} \leq s_i && \forall i \in O \\
& \sum_{i \in O} x_{i,j} = d_j && \forall j \in D \\
& x_{i,j} \geq 0 && \forall i \in O, j \in D
\end{aligned}
$$

Data

We assume our data is in the form of a text file that has the following form. In practice, we would obtain this text file from the user as input, but for the purpose of this tutorial we're going to create it from Julia.

```
open(joinpath(@__DIR__, "transp.txt"), "w") do io
    print(
        io,
        """
               .  FRA  DET LAN WIN   STL   FRE  LAF SUPPLY
           GARY  39   14  11  14    16    82    8   1400
           CLEV  27    .  12   .    26    95   17   2600
           PITT  24   14  17  13    28    99   20   2900
        DEMAND 900 1200 600 400  1700  1100 1000      0
        """,
    )
    return
end
```

Here the rows are the origins, the columns are the destinations, and the values are the cost of shipping one pogo stick from the origin to the destination. If pogo stick cannot be transported from a source to a destination, then the value is `.`. The final row and final column are the demand and supply of each location respectively.

We didn't account for arcs which do not exist in our formulation, but we can make a small change and fix $x_{i,j} = 0$ if $c_{i,j} = "."$.

Our first step is to convert this text format into an appropriate Julia datastructure that we can work with. Since our data is tabular with named rows and columns, one option is JuMP's `Containers.DenseAxisArray` object:

```
function read_data(filename::String)
    data = DelimitedFiles.readdlm(filename)
    rows, columns = data[2:end, 1], data[1, 2:end]
    return Containers.DenseAxisArray(data[2:end, 2:end], rows, columns)
end

data = read_data(joinpath(@__DIR__, "transp.txt"))
```

```
2-dimensional DenseAxisArray{Any,2,...} with index sets:
    Dimension 1, Any["GARY", "CLEV", "PITT", "DEMAND"]
    Dimension 2, Any["FRA", "DET", "LAN", "WIN", "STL", "FRE", "LAF", "SUPPLY"]
And data, a 4×8 Matrix{Any}:
  39    14      11   14       16    82      8  1400
  27     "."    12    "."     26    95     17  2600
  24    14      17   13       28    99     20  2900
 900  1200     600  400     1700  1100   1000     0
```

JuMP formulation

Following Design patterns for larger models, we code our JuMP model as a function which takes in an input. In this example, we print the output to stdout:

```julia
function solve_transportation_problem(data::Containers.DenseAxisArray)
    # Get the set of supplies and demands
    O, D = axes(data)
    # Drop the SUPPLY and DEMAND nodes from our sets
    O, D = setdiff(O, ["DEMAND"]), setdiff(D, ["SUPPLY"])
    model = Model(HiGHS.Optimizer)
    set_silent(model)
    @variable(model, x[o in O, d in D] >= 0)
    # Remove arcs with "." cost by fixing them to 0.0.
    for o in O, d in D
        if data[o, d] == "."
            fix(x[o, d], 0.0; force = true)
        end
    end
    @objective(
        model,
        Min,
        sum(data[o, d] * x[o, d] for o in O, d in D if data[o, d] != "."),
    )
    @constraint(model, [o in O], sum(x[o, :]) <= data[o, "SUPPLY"])
    @constraint(model, [d in D], sum(x[:, d]) == data["DEMAND", d])
    optimize!(model)
    # Pretty print the solution in the format of the input
    print("    ", join(lpad.(D, 7, ' ')))
    for o in O
        print("\n", o)
        for d in D
            if isapprox(value(x[o, d]), 0.0; atol = 1e-6)
                print("      .")
            else
                print(" ", lpad(value(x[o, d]), 6, ' '))
            end
        end
    end
    return
end
```

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

Solution

Let's solve and view the solution:

```
solve_transportation_problem(data)
```

```
        FRA     DET    LAN    WIN     STL    FRE    LAF
GARY      .       .      .      .   300.0 1100.0      .
CLEV      .       .  600.0      .  1000.0      . 1000.0
PITT  900.0 1200.0      .  400.0   400.0      .      .
```

5.18 Callbacks

The purpose of the tutorial is to demonstrate the various solver-independent and solver-dependent callbacks that are supported by JuMP.

The tutorial uses the following packages:

```
using JuMP
import GLPK
import Random
```

> **Info**
>
> This tutorial uses the MathOptInterface API. By default, JuMP exports the MOI symbol as an alias for the MathOptInterface.jl package. We recommend making this more explicit in your code by adding the following lines:
>
> ```
> import MathOptInterface as MOI
> ```

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)
```

```julia
        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 <= 3)
            println("Adding $(con)")
            MOI.submit(model, MOI.LazyConstraint(cb_data), con)
        end
    end
    set_attribute(model, MOI.LazyConstraintCallback(), my_callback_function)
    optimize!(model)
    println("Optimal solution (x, y) = ($(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-cuts

An example using a user-cut callback.

```julia
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 in 1:N if x_vals[i] > 1e-4)
        println("Called with accumulated = $(accumulated)")
        n_terms = sum(1 for i in 1:N if x_vals[i] > 1e-4)
        if accumulated > 10
            con = @build_constraint(
                sum(x[i] for i in 1:N if x_vals[i] > 0.5) <= n_terms - 1
            )
            println("Adding $(con)")
            MOI.submit(model, MOI.UserCut(cb_data), con)
        end
    end
    set_attribute(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
```

Heuristic solutions

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)
```

```julia
        ret =
            MOI.submit(model, MOI.HeuristicSolution(cb_data), x, floor.(x_vals))
        println("Heuristic solution status = $(ret)")
    end
    set_attribute(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.

```julia
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
    set_attribute(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
```

5.19 Constraint programming

JuMP supports a range of constraint-programming type constraints via the corresponding sets in MathOptInterface. For most constraints, there are reformulations built-in that convert the constraint programming constraint into a mixed-integer programming equivalent.

Because of this reformulation, all variables must be integer, and they must typically have finite bounds. An error will be thrown if the reformulation requires finiteness and you have a variable with non-finite bounds.

This tutorial uses the following packages:

```
using JuMP
import HiGHS
```

AllDifferent

The MOI.AllDifferent set ensures that every element in a list takes a different integer value.

```
model = Model(HiGHS.Optimizer)
set_silent(model)
@variable(model, 1 <= x[1:4] <= 4, Int)
@constraint(model, x in MOI.AllDifferent(4))
optimize!(model)
value.(x)
```

```
4-element Vector{Float64}:
 1.0
 4.0
 3.0000000000000004
 1.9999999999999996
```

BinPacking

The MOI.BinPacking set can be used to divide up a set of items into different groups, such that the sum of their weights does not exceed the capacity of a bin.

```
weights, capacity = Float64[1, 1, 2, 2, 3], 3.0;
number_of_bins = 3
model = Model(HiGHS.Optimizer)
set_silent(model)
```

```
@variable(model, 1 <= x[1:length(weights)] <= number_of_bins, Int)
@constraint(model, x in MOI.BinPacking(capacity, weights))
optimize!(model)
value.(x)
```

```
5-element Vector{Float64}:
 2.0
 1.0
 2.0
 1.0
 3.0
```

Here, the value of x[i] is the bin that item i was placed into.

Circuit

The MOI.Circuit set is used to construct a tour of a list of N variables. They will each be assigned an integer from 1 to N, that describes the successor to each variable in the list:

```
model = Model(HiGHS.Optimizer)
set_silent(model)
@variable(model, x[1:4], Int)
@constraint(model, x in MOI.Circuit(4))
optimize!(model)
```

Let's see what tour was found, starting at node number 1:

```
y = round.(Int, value.(x))
tour = Int[1]
while length(tour) < length(y)
    push!(tour, y[tour[end]])
end
tour
```

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

CountAtLeast

The MOI.CountAtLeast set is used to ensure that at least n elements in a set of variables belong to a set of values.

For example, here is a model with three variables, constrained between 0 and 5:

```
model = Model(HiGHS.Optimizer)
set_silent(model)
@variable(model, 0 <= x[1:3] <= 5, Int)
```

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

If we want to ensure that at least one element of each set {x[1], x[2]} and {x[2], x[3]} is in the set {3}, then we create a list of variables by concatenating the sets together:

```
variables = [x[1], x[2], x[2], x[3]]
```

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

Then we need a partition list that contains the number of elements in each set of variables:

```
partitions = [2, 2]
```

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

Finally, we need a set of values that the elements must be a part of:

```
values = Set([3])
```

```
Set{Int64} with 1 element:
  3
```

And the number of elements that must be part of the set values:

```
n = 1
```

```
1
```

The constraint is:

```
@constraint(model, variables in MOI.CountAtLeast(n, partitions, values))
```

$$[x_1, x_2, x_2, x_3] \in \text{MathOptInterface.CountAtLeast(1, [2, 2], Set([3]))}$$

To ensure the uniqueness of the solution, we'll add a constraint that x[2] must be <= 2. This ensures that the only feasible solution is for x[1] and x[3] to be 3:

```
@constraint(model, x[2] <= 2)
```

$$x_2 \leq 2$$

Let's check that we found a valid solution:

```
optimize!(model)
value.(x)
```

```
3-element Vector{Float64}:
 3.0
 0.0
 3.0
```

CountBelongs

The MOI.CountBelongs set is used to count how many elements in a set of variables belong to a set of values.

For example, to count how many elements in a set of 4 variables belong to the set {2, 3}, do:

```
model = Model(HiGHS.Optimizer)
set_silent(model)
@variable(model, 0 <= x[i = 1:4] <= i, Int)
@variable(model, n, Int)
@objective(model, Max, sum(x))
set = Set([2, 3])
@constraint(model, [n; x] in MOI.CountBelongs(1 + length(x), set))
optimize!(model)
value(n), value.(x)
```

```
(2.0, [1.0, 2.0, 3.0, 4.0])
```

CountDistinct

The MOI.CountDistinct set is used to count the number of distinct elements in a set of variables.

```
model = Model(HiGHS.Optimizer)
set_silent(model)
@variable(model, 0 <= x[i = 1:4] <= i, Int)
@variable(model, n, Int)
@objective(model, Max, sum(x))
@constraint(model, [n; x] in MOI.CountDistinct(1 + length(x)))
optimize!(model)
value(n), value.(x)
```

```
(4.0, [1.0, 2.0, 3.0, 4.0])
```

CountGreaterThan

The MOI.CountGreaterThan set is used to strictly upper-bound the number of distinct elements in a set of variables that have a value equal to another variable.

For example, to count the number n of times that y appears in the vector x, use:

```
model = Model(HiGHS.Optimizer)
set_silent(model)
@variable(model, 0 <= x[i = 1:4] <= i, Int)
@variable(model, n, Int)
@variable(model, 3 <= y <= 4, Int)
@objective(model, Max, sum(x))
@constraint(model, [n; y; x] in MOI.CountGreaterThan(1 + 1 + length(x)))
optimize!(model)
value(n), value(y), value.(x)
```

```
(2.0, 3.0, [1.0, 2.0, 3.0, 4.0])
```

Here n is strictly greater than the count, and there is no limit on how large n could be. For example, n = 100 is also a feasible solution. The only constraint is that n cannot be equal to or smaller than the number of times that y appears.

Table

The MOI.Table set is used to select a single row from a matrix of values.

For example, given a matrix:

```
table = Float64[1 1 0; 0 1 1; 1 0 1; 1 1 1]
```

```
4×3 Matrix{Float64}:
 1.0  1.0  0.0
 0.0  1.0  1.0
 1.0  0.0  1.0
 1.0  1.0  1.0
```

we can constraint a 3-element vector x to equal one of the rows in table via:

```
model = Model(HiGHS.Optimizer)
set_silent(model)
@variable(model, x[i = 1:3], Int)
@constraint(model, x in MOI.Table(table))
optimize!(model)
value.(x)
```

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

5.20 Computing the duals of a mixed-integer program

This tutorial explains how to compute the duals of a mixed-integer linear program by fixing the discrete variables to their optimal solution and resolving as a linear program.

This tutorial uses the following packages:

```
using JuMP
import HiGHS
```

The model

Our example model is the unit commitment example from Unit commitment. The details are unimportant, other than to note that there are two types of continuous variables, g and w, representing the quantity of generation from thermal and wind plants, and a discrete variable dispatch, which is 1 if plant i is operating, and 0 if not.

We are interested in the "dual" of the power_balance constraint, because it represents the marginal price of electricity that consumers should pay for their consumption.

```
generators = [
    (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),
]
N = length(generators)
model = Model(HiGHS.Optimizer)
set_silent(model)
@variables(model, begin
    generators[i].min <= g[i = 1:N] <= generators[i].max
```

```julia
    0 <= w <= 200
    dispatch[i = 1:N], Bin
end)
@constraints(model, begin
    power_balance, sum(g[i] for i in 1:N) + w == 1500
    [i = 1:N], g[i] <= generators[i].max * dispatch[i]
    [i = 1:N], g[i] >= generators[i].min * dispatch[i]
end)
@objective(
    model,
    Min,
    sum(
        generators[i].fixed_cost * dispatch[i] +
        generators[i].variable_cost * g[i] for i in 1:N
    )
)
print(model)
```

```
Min 1000 dispatch[1] + 50 g[1] + 100 g[2]
Subject to
 power_balance : g[1] + g[2] + w = 1500
 g[1] ≥ 0
 g[2] - 300 dispatch[2] ≥ 0
 g[1] - 1000 dispatch[1] ≤ 0
 g[2] - 1000 dispatch[2] ≤ 0
 g[1] ≥ 0
 g[2] ≥ 300
 w ≥ 0
 g[1] ≤ 1000
 g[2] ≤ 1000
 w ≤ 200
 dispatch[1] binary
 dispatch[2] binary
```

Manually fix the variables

If we optimize this model, we obtain a dual_status of NO_SOLUTION:

```julia
optimize!(model)
dual_status(model)
```

```
NO_SOLUTION::ResultStatusCode = 0
```

This is because HiGHS cannot compute the duals of a mixed-integer program. We can work around this problem by fixing the integer variables to their optimal solution, relaxing integrality, and re-solving as a linear program.

```julia
discrete_values = value.(dispatch)
fix.(dispatch, discrete_values; force = true)
```

```
unset_binary.(dispatch)
print(model)
```

```
Min 1000 dispatch[1] + 50 g[1] + 100 g[2]
Subject to
 power_balance : g[1] + g[2] + w = 1500
 g[1] ≥ 0
 g[2] - 300 dispatch[2] ≥ 0
 g[1] - 1000 dispatch[1] ≤ 0
 g[2] - 1000 dispatch[2] ≤ 0
 dispatch[1] = 1
 dispatch[2] = 1
 g[1] ≥ 0
 g[2] ≥ 300
 w ≥ 0
 g[1] ≤ 1000
 g[2] ≤ 1000
 w ≤ 200
```

Now if we re-solve the problem, we obtain a FEASIBLE_POINT for the dual:

```
optimize!(model)
dual_status(model)
```

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

and a marginal price of electricity of \$100/MWh:

```
dual(power_balance)
```

```
100.0
```

To reset the problem back to a mixed-integer linear program, we need to unfix and call set_binary:

```
unfix.(dispatch)
set_binary.(dispatch)
print(model)
```

```
Min 1000 dispatch[1] + 50 g[1] + 100 g[2]
Subject to
 power_balance : g[1] + g[2] + w = 1500
 g[1] ≥ 0
 g[2] - 300 dispatch[2] ≥ 0
```

```
g[1] - 1000 dispatch[1] ≤ 0
g[2] - 1000 dispatch[2] ≤ 0
g[1] ≥ 0
g[2] ≥ 300
w ≥ 0
g[1] ≤ 1000
g[2] ≤ 1000
w ≤ 200
dispatch[1] binary
dispatch[2] binary
```

Use `fix_discrete_variables`

Manually choosing the variables to relax and fix works for our small example, but it becomes more difficult in problems with a larger number of binary and integer variables. To automate the process we just did manually, JuMP provides the fix_discrete_variables function:

```
optimize!(model)
dual_status(model)
```

```
NO_SOLUTION::ResultStatusCode = 0
```

```
undo = fix_discrete_variables(model);
```

Here undo is a function that, when called with no arguments, returns the model to the original mixed-integer formulation.

> **Tip**
>
> After calling fix_discrete_variables, you can set a new solver with set_optimizer if your mixed-integer solver does not support computing a dual solution.

```
print(model)
```

```
Min 1000 dispatch[1] + 50 g[1] + 100 g[2]
Subject to
 power_balance : g[1] + g[2] + w = 1500
 g[1] ≥ 0
 g[2] - 300 dispatch[2] ≥ 0
 g[1] - 1000 dispatch[1] ≤ 0
 g[2] - 1000 dispatch[2] ≤ 0
 dispatch[1] = 1
 dispatch[2] = 1
 g[1] ≥ 0
 g[2] ≥ 300
```

```
 w ≥ 0
 g[1] ≤ 1000
 g[2] ≤ 1000
 w ≤ 200
```

```
optimize!(model)
dual_status(model)
```

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

```
dual(power_balance)
```

```
100.0
```

Finally, call undo to revert the reformulation

```
undo()
print(model)
```

```
Min 1000 dispatch[1] + 50 g[1] + 100 g[2]
Subject to
 power_balance : g[1] + g[2] + w = 1500
 g[1] ≥ 0
 g[2] - 300 dispatch[2] ≥ 0
 g[1] - 1000 dispatch[1] ≤ 0
 g[2] - 1000 dispatch[2] ≤ 0
 g[1] ≥ 0
 g[2] ≥ 300
 w ≥ 0
 g[1] ≤ 1000
 g[2] ≤ 1000
 w ≤ 200
 dispatch[1] binary
 dispatch[2] binary
```

Chapter 6

Nonlinear programs

6.1 Introduction

Nonlinear programs (NLPs) are a class of optimization problems in which some of the constraints or the objective function are nonlinear:

$$\min_{x \in \mathbb{R}^n} f_0(x) \tag{6.1}$$

$$\text{s.t.} \, l_j \leq f_j(x) \leq u_j \qquad\qquad j = 1 \ldots m \tag{6.2}$$

$$l_i \leq x_i \leq u_i \qquad\qquad i = 1 \ldots n. \tag{6.3}$$

Mixed-integer nonlinear linear programs (MINLPs) are extensions of nonlinear programs in which some (or all) of the decision variables take discrete values.

How to choose a solver

JuMP supports a range of nonlinear solvers; look for "NLP" in the list of Supported solvers. However, very few solvers support mixed-integer nonlinear linear programs. Solvers supporting discrete variables start with "(MI)" in the list of Supported solvers.

If the only nonlinearities in your model are quadratic terms (that is, multiplication between two decision variables), you can also use second-order cone solvers, which are indicated by "SOCP." In most cases, these solvers are restricted to convex quadratic problems and will error if you pass a nonconvex quadratic function; however, Gurobi has the ability to solve nonconvex quadratic terms.

How these tutorials are structured

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

- The following tutorials are worked examples that present a problem in words, then formulate it in mathematics, and then solve it in JuMP. This usually involves some sort of visualization of the solution. Start here if you are new to JuMP.

    - Rocket Control
    - Optimal control for a Space Shuttle reentry trajectory
    - Portfolio optimization

- The Tips and tricks tutorial contains a number of helpful reformulations and tricks you can use when modeling nonlinear programs. Look here if you are stuck trying to formulate a problem as a nonlinear program.

- The Computing Hessians is an advanced tutorial which explains how to compute the Hessian of the Lagrangian of a nonlinear program. This is useful only in particular cases.

- The remaining tutorials are less verbose and styled in the form of short code examples. These tutorials have less explanation, but may contain useful code snippets, particularly if they are similar to a problem you are trying to solve.

6.2   Simple examples

This tutorial is a collection of examples of small nonlinear programs. It uses the following packages:

```julia
using JuMP
import Ipopt
import Random
import Statistics
import Test
```

The Rosenbrock function

A nonlinear example of the classical Rosenbrock function.

```julia
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()
```

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.

```julia
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 3.14.4, running with linear solver MUMPS 5.4.1.

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
Variable bound 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 seconds in IPOPT                              = 0.102

EXIT: Optimal Solution Found.
termination_status = LOCALLY_SOLVED
primal_status      = FEASIBLE_POINT
objective_value    = 350.0000000000032
```

Maximum likelihood estimation

This example uses nonlinear optimization to compute the maximum likelihood estimate (MLE) of the parameters of a normal distribution, a.k.a., the sample mean and variance.

```julia
function example_mle()
    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)
```

```julia
    )
    optimize!(model)
    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))
    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
    println()
    println("With constraint μ == σ^2:")
    println("μ                         = ", value(μ))
    println("σ^2                       = ", value(σ)^2)
    println("Constrained MLE objective = ", objective_value(model))
    return
end

example_mle()
```

```
μ             = -0.022943864572027958
mean(data)    = -0.022943864572027916
σ^2           = 1.0096978289461431
var(data)     = 1.0107085374810936
MLE objective = -1423.76408661786


With constraint μ == σ^2:
μ                         = 0.6225971004178991
σ^2                       = 0.6225971004178991
Constrained MLE objective = -1827.5516590930729
```

Quadratically constrained programs

A simple quadratically constrained program based on an example from Gurobi.

```julia
function example_qcp()
    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)
    print(model)
    println("Objective value: ", objective_value(model))
    println("x = ", value(x))
```

```
    println("y = ", value(y))
    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
 x² + y² - z² ≤ 0
 x² - z*y ≤ 0
 y ≥ 0
 z ≥ 0
Objective value: 0.3269928349138724
x = 0.3269928349138724
y = 0.2570658388068964
```

6.3 Portfolio optimization

This tutorial was 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 Shabbir Ahmed.

Required packages

This tutorial uses the following packages

```
using JuMP
import DataFrames
import Ipopt
import MultiObjectiveAlgorithms as MOA
import Plots
import Statistics
import StatsPlots
```

Formulation

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.

We will use the initial 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, that is, 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:

$$\mathrm{Var}\left[\sum_{i=1}^{3} \tilde{r}_i x_i\right] = \sum_{i=1}^{3}\sum_{j=1}^{3} x_i x_j \sigma_{ij}$$

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:

$$\mathrm{Var}\left[\sum_{i=1}^{3} \tilde{r}_i x_i\right] = x^\top Q x$$

Where Q is the covariance matrix for the random vector $\tilde{r}$.

Finally, we can write the model as:

$$\min x^\top Q x$$

$$\text{s.t. } \sum_{i=1}^{3} x_i \leq 1000$$

$$\overline{r}^\top x \geq 50$$

$$x \geq 0$$

Data

For the data in our problem, we use the stock prices given below, in monthly values from November 2000, through November 2001.

```
df = DataFrames.DataFrame(
    [
        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
    ],
    [:IBM, :WMT, :SEHI],
)
```

	IBM	WMT	SEHI
	Float64	Float64	Float64
1	93.043	51.826	1.063
2	84.585	52.823	0.938
3	111.453	56.477	1.0
4	99.525	49.805	0.938
5	95.819	50.287	1.438
6	114.708	51.521	1.7
7	111.515	51.531	2.54
8	113.211	48.664	2.39
9	104.942	55.744	3.12
10	99.827	47.916	2.98
11	91.607	49.438	1.9
12	107.937	51.336	1.75
13	115.59	55.081	1.8

Next, we compute the percentage return for the stock in each month:

```
returns = diff(Matrix(df); dims = 1) ./ Matrix(df[1:end-1, :])
```

```
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
```

The expected monthly return is:

```
r = vec(Statistics.mean(returns; dims = 1))
```

```
3-element Vector{Float64}:
 0.026002150277777348
 0.008101316405671459
 0.07371590949198982
```

and the covariance matrix is:

```
Q = Statistics.cov(returns)
```

```
3×3 Matrix{Float64}:
 0.018641     0.00359853   0.00130976
 0.00359853   0.00643694   0.00488727
 0.00130976   0.00488727   0.0686828
```

JuMP formulation

```
model = Model(Ipopt.Optimizer)
set_silent(model)
@variable(model, x[1:3] >= 0)
@objective(model, Min, x' * Q * x)
@constraint(model, sum(x) <= 1000)
@constraint(model, r' * x >= 50)
optimize!(model)
solution_summary(model)
```

```
* Solver : Ipopt

* Status
  Result count       : 1
  Termination status : LOCALLY_SOLVED
  Message from the solver:
  "Solve_Succeeded"
```

```
* Candidate solution (result #1)
  Primal status       : FEASIBLE_POINT
  Dual status         : FEASIBLE_POINT
  Objective value     : 2.26344e+04
  Dual objective value : 4.52688e+04

* Work counters
  Solve time (sec)    : 4.73809e-03
```

The optimal allocation of our assets is:

```
value.(x)
```

```
3-element Vector{Float64}:
 497.04552984986407
  -9.670578501816873e-9
 502.9544801594808
```

So we spend $497 on IBM, and $503 on SEHI. This results in a variance of:

```
scalar_variance = value(x' * Q * x)
```

```
22634.41784988414
```

and an expected return of:

```
scalar_return = value(r' * x)
```

```
49.99999950000236
```

Multi-objective portfolio optimization

The previous model returned a single solution that minimized the variance, ensuring that our expected return was at least $50. In practice, we might be willing to accept a slightly higher variance if it meant a much increased expected return. To explore this problem space, we can instead formulate our portfolio optimization problem with two objectives:

1. to minimize the variance

2. to maximize the expected return

The solution to this bi-objective problem is the efficient frontier of modern portfolio theory, and each point in the solution is a point with the best return for a fixed level of risk.

```julia
model = Model(() -> MOA.Optimizer(Ipopt.Optimizer))
set_silent(model)
```

We also need to choose a solution algorithm for MOA. For our problem, the efficient frontier will have an infinite number of solutions. Since we cannot find all of the solutions, we choose an approximation algorithm and limit the number of solution points that are returned:

```julia
set_optimizer_attribute(model, MOA.Algorithm(), MOA.EpsilonConstraint())
set_optimizer_attribute(model, MOA.SolutionLimit(), 50)
```

Now we can define the rest of the model:

```julia
@variable(model, x[1:3] >= 0)
@constraint(model, sum(x) <= 1000)
@expression(model, variance, x' * Q * x)
@expression(model, expected_return, r' * x)
# We want to minimize variance and maximize expected return, but we must pick
# a single objective sense `Min`, and negate any `Max` objectives:
@objective(model, Min, [variance, -expected_return])
optimize!(model)
solution_summary(model)
```

```
* Solver : MOA[algorithm=MultiObjectiveAlgorithms.EpsilonConstraint, optimizer=Ipopt]

* Status
  Result count       : 50
  Termination status : OPTIMAL
  Message from the solver:
  "Solve complete. Found 50 solution(s)"

* Candidate solution (result #1)
  Primal status        : FEASIBLE_POINT
  Dual status          : NO_SOLUTION
  Objective value      : [2.58170e-08,-5.36777e-05]
  Objective bound      : [5.78303e-09,-7.37159e+01]
  Dual objective value : -7.37159e+01

* Work counters
  Solve time (sec)   : 6.38610e+00
```

The algorithm found 50 different solutions. Let's plot them to see how they differ:

```julia
objective_space = Plots.hline(
    [scalar_return];
    label = "Single-objective solution",
    linecolor = :red,
)
Plots.vline!(objective_space, [scalar_variance]; label = "", linecolor = :red)
Plots.scatter!(
```

```julia
    objective_space,
    [value(variance; result = i) for i in 1:result_count(model)],
    [value(expected_return; result = i) for i in 1:result_count(model)];
    xlabel = "Variance",
    ylabel = "Expected Return",
    label = "",
    title = "Objective space",
    markercolor = "white",
    markersize = 5,
    legend = :bottomright,
)
for i in 1:result_count(model)
    y = objective_value(model; result = i)
    Plots.annotate!(objective_space, y[1], -y[2], (i, 3))
end

decision_space = StatsPlots.groupedbar(
    vcat([value.(x; result = i)' for i in 1:result_count(model)]...);
    bar_position = :stack,
    label = ["IBM" "WMT" "SEHI"],
    xlabel = "Solution #",
    ylabel = "Investment (\$)",
    title = "Decision space",
)
Plots.plot(objective_space, decision_space; layout = (2, 1), size = (600, 600))
```

Perhaps our trade-off wasn't so bad after all. Our original solution corresponded to picking a solution #17. If we buy more SEHI, we can increase the return, but the variance also increases. If we buy less SEHI, such as a solution like #5 or #6, then we can achieve the corresponding return without deploying all of our capital. We should also note that at no point should we buy WMT.

6.4   Rocket Control

This tutorial was 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.

```julia
using JuMP
import Ipopt
import Plots
```

Create JuMP model, using Ipopt as the solver

```julia
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
  Result count       : 1
  Termination status : LOCALLY_SOLVED
  Message from the solver:
  "Solve_Succeeded"

* Candidate solution (result #1)
  Primal status       : FEASIBLE_POINT
  Dual status         : FEASIBLE_POINT
  Objective value     : 1.01283e+00
  Dual objective value : 4.57625e+00

* Work counters
  Solve time (sec)   : 1.49924e+00
```

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,
)
```

6.5   Optimal control for a Space Shuttle reentry trajectory

This tutorial was 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:

$$\dot{h} = v \sin \gamma,$$

$$\dot{\phi} = \frac{v}{r} \cos \gamma \sin \psi / \cos \theta,$$

$$\dot{\theta} = \frac{v}{r} \cos \gamma \cos \psi,$$

$$\dot{v} = -\frac{D}{m} - g \sin \gamma,$$

$$\dot{\gamma} = \frac{L}{mv} \cos(\beta) + \cos \gamma \left(\frac{v}{r} - \frac{g}{v} \right),$$

$$\dot{\psi} = \frac{1}{mv \cos \gamma} L \sin(\beta) + \frac{v}{r \cos \theta} \cos \gamma \sin \psi \sin \theta,$$

$$q \leq q_U,$$

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):

$$D = \frac{1}{2} c_D S \rho v^2, \qquad a_0 = -0.20704,$$

$$L = \frac{1}{2} c_L S \rho v^2, \qquad a_1 = 0.029244,$$

$$g = \mu/r^2, \qquad \mu = 0.14076539 \times 10^{17},$$

$$r = R_e + h, \qquad b_0 = 0.07854,$$

$$\rho = \rho_0 \exp[-h/h_r], \qquad b_1 = -0.61592 \times 10^{-2},$$

$$\rho_0 = 0.002378, \qquad b_2 = 0.621408 \times 10^{-3},$$

$$h_r = 23800, \qquad q_r = 17700 \sqrt{\rho} (0.0001v)^{3.07},$$

$$c_L = a_0 + a_1 \hat{\alpha}, \qquad q_a = c_0 + c_1 \hat{\alpha} + c_2 \hat{\alpha}^2 + c_3 \hat{\alpha}^3,$$

$$c_D = b_0 + b_1 \hat{\alpha} + b_2 \hat{\alpha}^2, \qquad c_0 = 1.0672181,$$

$$\hat{\alpha} = 180\alpha/\pi, \qquad c_1 = -0.19213774 \times 10^{-1},$$

$$R_e = 20902900, \qquad c_2 = 0.21286289 \times 10^{-3},$$

$$S = 2690, \qquad c_3 = -0.10117249 \times 10^{-5}.$$

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:

$$h = 260000 \text{ ft}, \qquad v = 25600 \text{ ft/sec},$$

$$\phi = 0 \text{ deg}, \qquad \gamma = -1 \text{ deg},$$

$$\theta = 0 \text{ deg}, \qquad \psi = 90 \text{ deg}.$$

Figure 6.1: Max cross-range 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 cross-range, which is equivalent to maximizing the final latitude of the vehicle, that is, $J = \theta(t_F)$.

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 h_r = 23800.0
const R_e = 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",
)
```

```julia
# 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.

```julia
# 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
```

```julia
@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]))
```

```julia
        @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 cross-range
@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 cross-range of the solution
println(
    "Final latitude θ = ",
    round(objective_value(model) |> rad2deg; digits = 2),
    "°",
)
```

```
Final latitude θ = 34.18°
```

Plotting the results

Let's plot the results to visualize the optimal trajectory:

```julia
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)")

plot(
```

```
    plt_altitude,
    plt_velocity,
    plt_longitude,
    plt_flight_path,
    plt_latitude,
    plt_azimuth;
    layout = grid(3, 2),
    linewidth = 2,
    size = (700, 700),
)
```

Altitude (100,000 ft)

Velocity (1000 ft/sec)

Longitude (deg)

Flight Path (deg)

Latitude (deg)

Azimuth (deg)
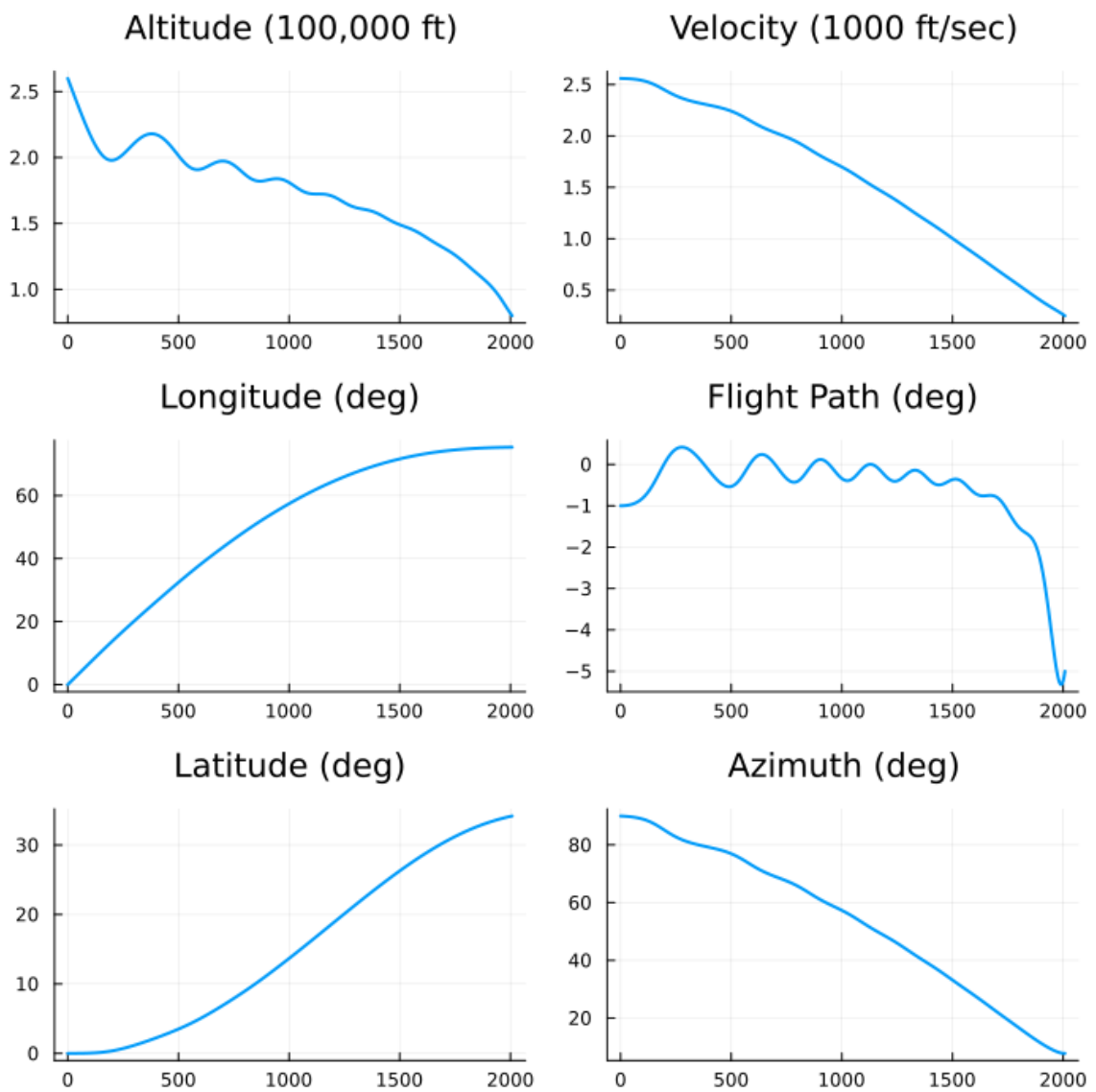
```
function q(h, v, a)
    ρ(h) = ρ₀ * exp(-h / hᵣ)
    qᵣ(h, v) = 17700 * √ρ(h) * (0.0001 * v)^3.07
    qₐ(a) = c₀ + c₁ * rad2deg(a) + c₂ * rad2deg(a)^2 + c₃ * rad2deg(a)^3
    # Aerodynamic heating on the vehicle wing leading edge
    return qₐ(a) * qᵣ(h, v)
end

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)",
)

plot(
    plt_attack_angle,
    plt_bank_angle,
    plt_heating;
    layout = grid(3, 1),
    linewidth = 2,
    size = (700, 700),
)
```

Angle of Attack (deg)

Bank Angle (deg)

Heating (BTU/ft/ft/sec)

```
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)",
)
```

Space Shuttle Reentry Trajectory

6.6 Tips and tricks

This example collates some tips and tricks you can use when formulating nonlinear programs. It uses the following packages:

```julia
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):

```julia
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 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 element
`i` is a function that returns the equivalent of `foo(x...)[i]`.

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
    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.

6.7 User-defined Hessians

In this tutorial, we explain how to write a user-defined function (see User-defined Functions) with a Hessian matrix explicitly provided by the user.

For a more advanced example, see Nested optimization problems.

This tutorial uses the following packages:

```
using JuMP
import Ipopt
```

Rosenbrock example

As a simple example, we consider the Rosenbrock function:

```
rosenbrock(x...) = (1 - x[1])^2 + 100 * (x[2] - x[1]^2)^2
```

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

which has the gradient vector:

```
function ∇rosenbrock(g::AbstractVector, x...)
    g[1] = 400 * x[1]^3 - 400 * x[1] * x[2] + 2 * x[1] - 2
    g[2] = 200 * (x[2] - x[1]^2)
    return
end
```

```
∇rosenbrock (generic function with 1 method)
```

and the Hessian matrix:

```
function ∇²rosenbrock(H::AbstractMatrix, x...)
    H[1, 1] = 1200 * x[1]^2 - 400 * x[2] + 2
    # H[1, 2] = -400 * x[1] <-- not needed because Hessian is symmetric
    H[2, 1] = -400 * x[1]
    H[2, 2] = 200.0
    return
end
```

```
∇²rosenbrock (generic function with 1 method)
```

You may assume the Hessian matrix H is initialized with zeros, and because it is symmetric you need only to fill in the non-zero of the lower-triangular terms.

The matrix type passed in as H depends on the automatic differentiation system, so make sure the first argument to the Hessian function supports an AbstractMatrix (it may be something other than Matrix{Float64}). However, you may assume only that H supports size(H) and setindex!.

Now that we have the function, its gradient, and its Hessian, we can construct a JuMP model, register the function, and use it in a @NL macro:

```
model = Model(Ipopt.Optimizer)
@variable(model, x[1:2])
register(model, :rosenbrock, 2, rosenbrock, ∇rosenbrock, ∇²rosenbrock)
@NLobjective(model, Min, rosenbrock(x[1], x[2]))
optimize!(model)
solution_summary(model; verbose = true)
```

```
* Solver : Ipopt

* Status
  Result count       : 1
  Termination status : LOCALLY_SOLVED
  Message from the solver:
  "Solve_Succeeded"

* Candidate solution (result #1)
  Primal status      : FEASIBLE_POINT
  Dual status        : FEASIBLE_POINT
  Objective value    : 1.20425e-27
  Dual objective value : 0.00000e+00
  Primal solution :
    x[1] : 1.00000e+00
    x[2] : 1.00000e+00
  Dual solution :
```

```
* Work counters
  Solve time (sec)    : 5.18520e-02
```

6.8 Nested optimization problems

In this tutorial we explain how to write a nested optimization problem, where an outer problem uses the results from the optimization of an inner subproblem.

To model the problem, we define a user-defined function to handle the decomposition of the inner problem inside the outer one. Finally, we show how to improve the performance by using a cache that avoids resolving the inner problem.

For a simpler example of writing a user-defined function, see the User-defined Hessians tutorial.

This tutorial uses the following packages:

```
using JuMP
import Ipopt
```

In the rest of this tutorial, our goal is to solve the bilevel optimization problem:

$$
\begin{aligned}
\min_{x,z} \quad & x_1^2 + x_2^2 + z \\
s.t. \quad & z = \max_{y} \quad x_1^2 y_1 + x_2^2 y_2 - x_1 y_1^4 - 2 x_2 y_2^4 \\
& \qquad\quad s.t. \quad (y_1 - 10)^2 + (y_2 - 10)^2 \leq 25 \\
& x \geq 0.
\end{aligned}
$$

This bilevel optimization problem is composed of two nested optimization problems. An upper level, involving variables x, and a lower level, involving variables y. From the perspective of the lower-level problem, the values of x are fixed parameters, and so the model optimizes y given those fixed parameters. Simultaneously, the upper-level problem optimizes x and z given the response of y.

Decomposition

There are a few ways to solve this problem, but we are going to use a nonlinear decomposition method. The first step is to write a function to compute the lower-level problem:

$$
\begin{aligned}
V(x_1, x_2) = \max_{y} \quad & x_1^2 y_1 + x_2^2 y_2 - x_1 y_1^4 - 2 x_2 y_2^4 \\
s.t. \quad & (y_1 - 10)^2 + (y_2 - 10)^2 \leq 25
\end{aligned}
$$

```
function solve_lower_level(x...)
    model = Model(Ipopt.Optimizer)
    set_silent(model)
    @variable(model, y[1:2])
    @NLobjective(
        model,
        Max,
        x[1]^2 * y[1] + x[2]^2 * y[2] - x[1] * y[1]^4 - 2 * x[2] * y[2]^4,
```

```
    )
    @constraint(model, (y[1] - 10)^2 + (y[2] - 10)^2 <= 25)
    optimize!(model)
    @assert termination_status(model) == LOCALLY_SOLVED
    return objective_value(model), value.(y)
end
```

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

The next function takes a value of x and returns the optimal lower-level objective-value and the optimal response y. The reason why we need both the objective and the optimal y will be made clear shortly, but for now let us define:

```
function V(x...)
    f, _ = solve_lower_level(x...)
    return f
end
```

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

Then, we can substitute V into our full problem to create:

$$\min_{x} \quad x_1^2 + x_2^2 + V(x_1, x_2)$$
$$s.t. \quad x \geq 0.$$

This looks like a nonlinear optimization problem with a user-defined function V! However, because V solves an optimization problem internally, we can't use automatic differentiation to compute the first and second derivatives. Instead, we can use JuMP's ability to pass callback functions for the gradient and Hessian instead.

First up, we need to define the gradient of V with respect to x. In general, this may be difficult to compute, but because x appears only in the objective, we can just differentiate the objective function with respect to x, giving:

```
function ∇V(g::AbstractVector, x...)
    _, y = solve_lower_level(x...)
    g[1] = 2 * x[1] * y[1] - y[1]^4
    g[2] = 2 * x[2] * y[2] - 2 * y[2]^4
    return
end
```

```
∇V (generic function with 1 method)
```

Second, we need to define the Hessian of V with respect to x. This is a symmetric matrix, but in our example only the diagonal elements are non-zero:

```julia
function ∇²V(H::AbstractMatrix, x...)
    _, y = solve_lower_level(x...)
    H[1, 1] = 2 * y[1]
    H[2, 2] = 2 * y[2]
    return
end
```

```
∇²V (generic function with 1 method)
```

> **Info**
>
> Providing an explicit Hessian function is optional if first derivatives are already available.

We now have enough to define our bilevel optimization problem:

```julia
model = Model(Ipopt.Optimizer)
@variable(model, x[1:2] >= 0)
register(model, :V, 2, V, ∇V, ∇²V)
@NLobjective(model, Min, x[1]^2 + x[2]^2 + V(x[1], x[2]))
optimize!(model)
solution_summary(model)
```

```
* Solver : Ipopt

* Status
  Result count       : 1
  Termination status : LOCALLY_SOLVED
  Message from the solver:
  "Solve_Succeeded"

* Candidate solution (result #1)
  Primal status        : FEASIBLE_POINT
  Dual status          : FEASIBLE_POINT
  Objective value      : -4.18983e+05
  Dual objective value : 0.00000e+00

* Work counters
  Solve time (sec)   : 9.94209e-01
```

The optimal objective value is:

```julia
objective_value(model)
```

```
-418983.4868064086
```

and the optimal upper-level decision variables x are:

```
value.(x)
```

```
2-element Vector{Float64}:
 154.97862349841566
 180.00961418355053
```

To compute the optimal lower-level decision variables, we need to call `solve_lower_level` with the optimal upper-level decision variables:

```
_, y = solve_lower_level(value.(x)...)
y
```

```
2-element Vector{Float64}:
 7.072593960195712
 5.946569893523139
```

Improving performance

Our solution approach works, but it has a performance problem: every time we need to compute the value, gradient, or Hessian of V, we have to re-solve the lower-level optimization problem. This is wasteful, because we will often call the gradient and Hessian at the same point, and so solving the problem twice with the same input repeats work unnecessarily.

We can work around this by using a cache:

```
mutable struct Cache
    x::Any
    f::Float64
    y::Vector{Float64}
end
```

with a function to update the cache if needed:

```
function _update_if_needed(cache::Cache, x...)
    if cache.x !== x
        cache.f, cache.y = solve_lower_level(x...)
        cache.x = x
    end
    return
end
```

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

Then, we define cached versions of out three functions which call _updated_if_needed and return values from the cache.

```julia
function cached_f(cache::Cache, x...)
    _update_if_needed(cache, x...)
    return cache.f
end

function cached_∇f(cache::Cache, g::AbstractVector, x...)
    _update_if_needed(cache, x...)
    g[1] = 2 * x[1] * cache.y[1] - cache.y[1]^4
    g[2] = 2 * x[2] * cache.y[2] - 2 * cache.y[2]^4
    return
end

function cached_∇²f(cache::Cache, H::AbstractMatrix, x...)
    _update_if_needed(cache, x...)
    H[1, 1] = 2 * cache.y[1]
    H[2, 2] = 2 * cache.y[2]
    return
end
```

```
cached_∇²f (generic function with 1 method)
```

Now we're ready to setup and solve the upper level optimization problem:

```julia
model = Model(Ipopt.Optimizer)
@variable(model, x[1:2] >= 0)
cache = Cache(Float64[], NaN, Float64[])
register(
    model,
    :V,
    2,
    (x...) -> cached_f(cache, x...),
    (g, x...) -> cached_∇f(cache, g, x...),
    (H, x...) -> cached_∇²f(cache, H, x...),
)
@NLobjective(model, Min, x[1]^2 + x[2]^2 + V(x[1], x[2]))
optimize!(model)
solution_summary(model)
```

```
* Solver : Ipopt

* Status
  Result count       : 1
  Termination status : LOCALLY_SOLVED
  Message from the solver:
  "Solve_Succeeded"

* Candidate solution (result #1)
```

```
  Primal status       : FEASIBLE_POINT
  Dual status         : FEASIBLE_POINT
  Objective value     : -4.18983e+05
  Dual objective value : 0.00000e+00

* Work counters
  Solve time (sec)    : 3.75050e-01
```

an we can check we get the same objective value:

```
objective_value(model)
```

```
-418983.4868064086
```

and upper-level decision variable x:

```
value.(x)
```

```
2-element Vector{Float64}:
 154.97862349841566
 180.00961418355053
```

6.9 Computing Hessians

The purpose of this tutorial is to demonstrate how to compute the Hessian of the Lagrangian of a nonlinear program.

> **Warning**
>
> This is an advanced tutorial that interacts with the low-level nonlinear interface of MathOptInterface.
>
> By default, JuMP exports the MOI symbol as an alias for the MathOptInterface.jl package. We recommend making this more explicit in your code by adding the following lines:
>
> ```
> import MathOptInterface as MOI
> ```

Given a nonlinear program:

$$\min_{x\in\mathbb{R}^n} \qquad f(x) \qquad\qquad (6.4)$$

$$\text{s.t.} \qquad l \le g_i(x) \le u \qquad\qquad (6.5)$$

the Hessian of the Lagrangian is computed as:

$$H(x, \sigma, \mu) = \sigma \nabla^2 f(x) + \sum_{i=1}^{m} \mu_i \nabla^2 g_i(x)$$

where x is a primal point, σ is a scalar (typically 1), and μ is a vector of weights corresponding to the Lagrangian dual of the constraints.

This tutorial uses the following packages:

```
using JuMP
import Ipopt
import LinearAlgebra
import Random
import SparseArrays
```

The basic model

To demonstrate how to interact with the lower-level nonlinear interface, we need an example model. The exact model isn't important; we use the model from The Rosenbrock function tutorial, with some additional constraints to demonstrate various features of the lower-level interface.

```
model = Model(Ipopt.Optimizer)
set_silent(model)
@variable(model, x[i = 1:2], start = -i)
@constraint(model, g_1, x[1]^2 <= 1)
@NLconstraint(model, g_2, (x[1] + x[2])^2 <= 2)
@NLobjective(model, Min, (1 - x[1])^2 + 100 * (x[2] - x[1]^2)^2)
optimize!(model)
```

The analytic solution

With a little work, it is possible to analytically derive the correct hessian:

```
function analytic_hessian(x, σ, μ)
    g_1_H = [2.0 0.0; 0.0 0.0]
    g_2_H = [2.0 2.0; 2.0 2.0]
    f_H = zeros(2, 2)
    f_H[1, 1] = 2.0 + 1200.0 * x[1]^2 - 400.0 * x[2]
    f_H[1, 2] = f_H[2, 1] = -400.0 * x[1]
    f_H[2, 2] = 200.0
    return σ * f_H + μ' * [g_1_H, g_2_H]
end
```

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

Here are various points:

```
analytic_hessian([1, 1], 0, [0, 0])
```

```
2×2 Matrix{Float64}:
 0.0  0.0
 0.0  0.0
```

```
analytic_hessian([1, 1], 0, [1, 0])
```

```
2×2 Matrix{Float64}:
 2.0  0.0
 0.0  0.0
```

```
analytic_hessian([1, 1], 0, [0, 1])
```

```
2×2 Matrix{Float64}:
 2.0  2.0
 2.0  2.0
```

```
analytic_hessian([1, 1], 1, [0, 0])
```

```
2×2 Matrix{Float64}:
  802.0  -400.0
 -400.0   200.0
```

Initializing the NLPEvaluator

JuMP stores all information relating to the nonlinear portions of the model in a NLPEvaluator struct:

```
d = NLPEvaluator(model)
```

```
Nonlinear.Evaluator with available features:
  * :Grad
  * :Jac
  * :JacVec
  * :Hess
  * :HessVec
  * :ExprGraph
```

Before computing anything with the NLPEvaluator, we need to initialize it. Use MOI.features_available to see what we can query:

```
MOI.features_available(d)
```

```
6-element Vector{Symbol}:
 :Grad
 :Jac
 :JacVec
 :Hess
 :HessVec
 :ExprGraph
```

Consult the MOI documentation for specifics. But to obtain the Hessian matrix, we need to initialize :Hess:

```
MOI.initialize(d, [:Hess])
```

MOI represents the Hessian as a sparse matrix. Get the sparsity pattern as follows:

```
hessian_sparsity = MOI.hessian_lagrangian_structure(d)
```

```
6-element Vector{Tuple{Int64, Int64}}:
 (1, 1)
 (2, 2)
 (2, 1)
 (1, 1)
 (2, 2)
 (2, 1)
```

The sparsity pattern has a few properties of interest:

- Each element (i, j) indicates a structural non-zero in row i and column j

- The list may contain duplicates, in which case we should add the values together

- The list does not need to be sorted

- The list may contain any mix of lower- or upper-triangular indices

This format matches Julia's sparse-triplet form of a SparseArray, so we can convert from the sparse Hessian representation to a Julia SparseArray as follows:

```
I = [i for (i, _) in hessian_sparsity]
J = [j for (_, j) in hessian_sparsity]
V = zeros(length(hessian_sparsity))
n = num_variables(model)
H = SparseArrays.sparse(I, J, V, n, n)
```

```
2×2 SparseArrays.SparseMatrixCSC{Float64, Int64} with 3 stored entries:
 0.0   ⋅
 0.0  0.0
```

Of course, knowing where the zeros are isn't very interesting. We really want to compute the value of the Hessian matrix at a point.

```
num_g = num_nonlinear_constraints(model)
MOI.eval_hessian_lagrangian(d, V, ones(n), 1.0, ones(num_g))
H = SparseArrays.sparse(I, J, V, n, n)
```

```
2×2 SparseArrays.SparseMatrixCSC{Float64, Int64} with 3 stored entries:
   804.0     ⋅
  -398.0  202.0
```

In practice, we often want to compute the value of the hessian at the optimal solution.

First, we compute the primal solution. To do so, we need a vector of the variables in the order that they were passed to the solver:

```
x = all_variables(model)
```

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

Here x[1] is the variable that corresponds to column 1, and so on. Here's the optimal primal solution:

```
x_optimal = value.(x)
```

```
2-element Vector{Float64}:
 0.7903587551555908
 0.6238546250475736
```

Next, we need the optimal dual solution associated with the nonlinear constraints:

```
y_optimal = dual.(all_nonlinear_constraints(model))
```

```
1-element Vector{Float64}:
 -0.057440893636909414
```

Now we can compute the Hessian at the optimal primal-dual point:

```
MOI.eval_hessian_lagrangian(d, V, x_optimal, 1.0, y_optimal)
H = SparseArrays.sparse(I, J, V, n, n)
```

```
2×2 SparseArrays.SparseMatrixCSC{Float64, Int64} with 3 stored entries:
  501.944      ⋅
 -316.258   199.885
```

However, this Hessian isn't quite right because it isn't symmetric. We can fix this by filling in the appropriate off-diagonal terms:

```
function fill_off_diagonal(H)
    ret = H + H'
    row_vals = SparseArrays.rowvals(ret)
    non_zeros = SparseArrays.nonzeros(ret)
    for col in 1:size(ret, 2)
        for i in SparseArrays.nzrange(ret, col)
            if col == row_vals[i]
                non_zeros[i] /= 2
            end
        end
    end
    return ret
end

fill_off_diagonal(H)
```

```
2×2 SparseArrays.SparseMatrixCSC{Float64, Int64} with 4 stored entries:
  501.944   -316.258
 -316.258    199.885
```

Moreover, this Hessian only accounts for the objective and constraints entered using @NLobjective and @NLconstraint. If we want to take quadratic objectives and constraints written using @objective or @constraint into account, we'll need to handle them separately.

> **Tip**
>
> If you don't want to do this, you can replace calls to @objective and @constraint with @NLobjective and @NLconstraint.

Hessians from QuadExpr functions

To compute the hessian from a quadratic expression, let's see how JuMP represents a quadratic constraint:

```
f = constraint_object(g_1).func
```

$$x_1^2$$

f is a quadratic expression of the form:

```
f(x) = Σqᵢⱼ * xᵢ * xⱼ + Σaᵢ xᵢ + c
```

So $\nabla^2 f(x)$ is the matrix formed by $[q_{ij}]_{ij}$ if i != j and $2[q_{ij}]_{ij}$ if i = j.

```
variables_to_column = Dict(x[i] => i for i in 1:n)

function add_to_hessian(H, f::QuadExpr, μ)
    for (vars, coef) in f.terms
        i = variables_to_column[vars.a]
        j = variables_to_column[vars.b]
        H[i, j] += μ * coef
    end
    return
end
```

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

If the function f is not a QuadExpr, do nothing because it is an AffExpr or a VariableRef. In both cases, the second derivative is zero.

```
add_to_hessian(H, f::Any, μ) = nothing
```

```
add_to_hessian (generic function with 2 methods)
```

Then we iterate over all constraints in the model and add their Hessian components:

```
for (F, S) in list_of_constraint_types(model)
    for cref in all_constraints(model, F, S)
        f = constraint_object(cref).func
        add_to_hessian(H, f, dual(cref))
    end
end

H
```

```
2×2 SparseArrays.SparseMatrixCSC{Float64, Int64} with 3 stored entries:
  501.944      ·
 -316.258  199.885
```

Finally, we need to take into account the objective function:

```
add_to_hessian(H, objective_function(model), 1.0)

fill_off_diagonal(H)
```

```
2×2 SparseArrays.SparseMatrixCSC{Float64, Int64} with 4 stored entries:
   501.944   -316.258
  -316.258    199.885
```

Putting everything together:

```julia
function compute_optimal_hessian(model)
    d = NLPEvaluator(model)
    MOI.initialize(d, [:Hess])
    hessian_sparsity = MOI.hessian_lagrangian_structure(d)
    I = [i for (i, _) in hessian_sparsity]
    J = [j for (_, j) in hessian_sparsity]
    V = zeros(length(hessian_sparsity))
    x = all_variables(model)
    x_optimal = value.(x)
    y_optimal = dual.(all_nonlinear_constraints(model))
    MOI.eval_hessian_lagrangian(d, V, x_optimal, 1.0, y_optimal)
    n = num_variables(model)
    H = SparseArrays.sparse(I, J, V, n, n)
    vmap = Dict(x[i] => i for i in 1:n)
    add_to_hessian(H, f::Any, μ) = nothing
    function add_to_hessian(H, f::QuadExpr, μ)
        for (vars, coef) in f.terms
            if vars.a != vars.b
                H[vmap[vars.a], vmap[vars.b]] += μ * coef
            else
                H[vmap[vars.a], vmap[vars.b]] += 2 * μ * coef
            end
        end
    end
    for (F, S) in list_of_constraint_types(model)
        for cref in all_constraints(model, F, S)
            add_to_hessian(H, constraint_object(cref).func, dual(cref))
        end
    end
    add_to_hessian(H, objective_function(model), 1.0)
    return Matrix(fill_off_diagonal(H))
end

H_star = compute_optimal_hessian(model)
```

```
2×2 Matrix{Float64}:
   501.944   -316.258
  -316.258    199.885
```

If we compare our solution against the analytical solution:

```
analytic_hessian(value.(x), 1.0, dual.([g_1, g_2]))
```

```
2×2 Matrix{Float64}:
   501.944  -316.258
  -316.258   199.885
```

If we look at the eigenvalues of the Hessian:

```
LinearAlgebra.eigvals(H_star)
```

```
2-element Vector{Float64}:
   0.44439959255495864
 701.3843408744132
```

we see that they are all positive. Therefore, the Hessian is positive definite, and so the solution found by Ipopt is a local minimizer.

Chapter 7

Conic programs

7.1 Introduction

Conic programs are a class of convex nonlinear optimization problems which use cones to represent the nonlinearities. They have the form:

$$\min_{x \in \mathbb{R}^n} f_0(x) \tag{7.1}$$

$$\text{s.t.} \quad f_j(x) \in \mathcal{S}_j \ \ j = 1 \ldots m \tag{7.2}$$

Mixed-integer conic programs (MICPs) are extensions of conic programs in which some (or all) of the decision variables take discrete values.

How to choose a solver

JuMP supports a range of conic solvers, although support differs on what types of cones each solver supports. In the list of Supported solvers, "SOCP" denotes solvers supporting second-order cones and "SDP" denotes solvers supporting semidefinite cones. In addition, solvers such as SCS and Mosek have support for the exponential cone. Moreover, due to the bridging system in MathOptInterface, many of these solvers support a much wider range of exotic cones than they natively support. Solvers supporting discrete variables start with "(MI)" in the list of Supported solvers.

How these tutorials are structured

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

- The following tutorials are worked examples that present a problem in words, then formulate it in mathematics, and then solve it in JuMP. This usually involves some sort of visualization of the solution. Start here if you are new to JuMP.

    - Experiment design
    - Logistic regression

- The Tips and tricks tutorial contains a number of helpful reformulations and tricks you can use when modeling conic programs. Look here if you are stuck trying to formulate a problem as a conic program.

- The remaining tutorials are less verbose and styled in the form of short code examples. These tutorials have less explanation, but may contain useful code snippets, particularly if they are similar to a problem you are trying to solve.

7.2 Primal and dual warm-starts

Some conic solvers have the ability to set warm-starts for the primal and dual solution. This can improve performance, particularly if you are repeatedly solving a sequence of related problems.

> **Tip**
>
> See set_start_values for a generic implementation of this function that was added to JuMP after this tutorial was written.

In this tutorial, we demonstrate how to write a function that sets the primal and dual starts as the optimal solution stored in a model. It is intended to be a starting point for which you can modify if you want to do something similar in your own code.

> **Warning**
>
> This tutorial does not set start values for nonlinear models.

This tutorial uses the following packages:

```
using JuMP
import SCS
```

The main component of this tutorial is the following function. The most important observation is that we cache all of the solution values first, and then we modify the model second. (Alternating between querying a value and modifying the model is not allowed in JuMP.)

```
function set_optimal_start_values(model::Model)
    # Store a mapping of the variable primal solution
    variable_primal = Dict(x => value(x) for x in all_variables(model))
    # In the following, we loop through every constraint and store a mapping
    # from the constraint index to a tuple containing the primal and dual
    # solutions.
    constraint_solution = Dict()
    for (F, S) in list_of_constraint_types(model)
        # We add a try-catch here because some constraint types might not
        # support getting the primal or dual solution.
        try
            for ci in all_constraints(model, F, S)
                constraint_solution[ci] = (value(ci), dual(ci))
            end
        catch
            @info("Something went wrong getting $F-in-$S. Skipping")
        end
    end
    # Now we can loop through our cached solutions and set the starting values.
    for (x, primal_start) in variable_primal
        set_start_value(x, primal_start)
```

```
    end
    for (ci, (primal_start, dual_start)) in constraint_solution
        set_start_value(ci, primal_start)
        set_dual_start_value(ci, dual_start)
    end
    return
end
```

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

To test our function, we use the following linear program:

```
model = Model(SCS.Optimizer)
@variable(model, x[1:3] >= 0)
@constraint(model, sum(x) <= 1)
@objective(model, Max, sum(i * x[i] for i in 1:3))
optimize!(model)
```

By looking at the log (not shown in Documenter due to a bug), we can see that SCS took 100 iterations to find the optimal solution. Now we set the optimal solution as our starting point:

```
set_optimal_start_values(model)
```

and we re-optimize:

```
optimize!(model)
```

Now the optimization terminates after 0 iterations because our starting point is already optimal.

Note that some solvers do not support setting some parts of the starting solution, for example, they may support only `set_start_value` for variables. If you encounter an `UnsupportedSupported` attribute error for `MOI.VariablePrimalStart`, `MOI.ConstraintPrimalStart`, or `MOI.ConstraintDualStart`, comment out the corresponding part of the set_optimal_start_values function.

7.3 Tips and Tricks

This tutorial was 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
```

> **Info**
>
> This tutorial uses sets from MathOptInterface. By default, JuMP exports the MOI symbol as an alias for the MathOptInterface.jl package. We recommend making this more explicit in your code by adding the following lines:
>
> ```
> import MathOptInterface as MOI
> ```

> **Tip**
>
> A good resource for learning more about functions which can be modeled using cones is the MOSEK Modeling Cookbook.

Background theory

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 $x, y \in C$.

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:

$$
\begin{aligned}
\min_{x \in \mathbb{R}^n} \quad & a_0^T x + b_0 \\
\text{s.t.} \quad & A_i x + b_i \in \mathcal{C}_i \quad i = 1 \ldots m
\end{aligned}
$$

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 SecondOrderCone (or Lorentz Cone) of dimension n is a cone of the form:

$$
K_{soc} = \{(t, x) \in \mathbb{R}^n : t \geq ||x||_2\}
$$

It is most commonly used to represent the L2-norm of the vector x:

```
model = Model(SCS.Optimizer)
set_silent(model)
@variable(model, x[1:3])
@variable(model, t)
@constraint(model, sum(x) == 1)
@constraint(model, [t; x] in SecondOrderCone())
@objective(model, Min, t)
optimize!(model)
value(t), value.(x)
```

```
(0.5773503148525505, [0.3333333333638094, 0.33333333336381227, 0.33333333336381227])
```

Rotated Second-Order Cone

A Second-Order Cone rotated by $\pi/4$ in the (x_1, x_2) plane is called a RotatedSecondOrderCone. It is a cone of the form:

$$K_{rsoc} = \{(t, u, x) \in \mathbb{R}^n : 2tu \geq ||x||_2^2, t, u \geq 0\}$$

When u = 0.5, it represents the sum of squares of a vector x:

```
data = [1.0, 2.0, 3.0, 4.0]
target = [0.45, 1.04, 1.51, 1.97]
model = Model(SCS.Optimizer)
set_silent(model)
@variable(model, θ)
@variable(model, t)
@expression(model, residuals, θ * data .- target)
@constraint(model, [t; 0.5; residuals] in RotatedSecondOrderCone())
@objective(model, Min, t)
optimize!(model)
value(θ), value(t)
```

```
(0.4980000000000055, 0.004979422159620655)
```

Exponential Cone

The MOI.ExponentialCone is a set of the form:

$$K_{exp} = \{(x, y, z) \in \mathbb{R}^3 : y \exp(x/y) \leq z, y > 0\}$$

It can be used to model problems involving log and exp.

Exponential

To model $\exp(x) \leq z$, use (x, 1, z):

```
model = Model(SCS.Optimizer)
set_silent(model)
@variable(model, x == 1.5)
@variable(model, z)
@objective(model, Min, z)
@constraint(model, [x, 1, z] in MOI.ExponentialCone())
optimize!(model)
value(z), exp(1.5)
```

```
(4.481654619603649, 4.4816890703380645)
```

Logarithm

To model $x \leq \log(z)$, use (x, 1, z):

```
model = Model(SCS.Optimizer)
set_silent(model)
@variable(model, x)
@variable(model, z == 1.5)
@objective(model, Max, x)
@constraint(model, [x, 1, z] in MOI.ExponentialCone())
optimize!(model)
value(x), log(1.5)
```

```
(0.4055956586655662, 0.4054651081081644)
```

Log-sum-exp

To model $t \geq \log\left(\sum e^{x_i}\right)$, use:

```
N = 3
x0 = rand(N)
model = Model(SCS.Optimizer)
set_silent(model)
@variable(model, x[i = 1:N] == x0[i])
@variable(model, t)
@objective(model, Min, t)
@variable(model, u[1:N])
@constraint(model, sum(u) <= 1)
@constraint(model, [i = 1:N], [x[i] - t, 1, u[i]] in MOI.ExponentialCone())
optimize!(model)
value(t), log(sum(exp.(x0)))
```

```
(1.7438822741060218, 1.743968586067829)
```

Entropy

The entropy maximization problem consists of maximizing the entropy function, $H(x) = -x \log x$ subject to linear inequality constraints.

$$
\begin{aligned}
\max \quad & -\sum_{i=1}^{n} x_i \log x_i \\
\text{s.t.} \quad & \mathbf{1}^\top x = 1 \\
& Ax \leq b
\end{aligned}
$$

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,

$$
\begin{aligned}
\max \quad & 1^T t \\
\text{s.t.} \quad & Ax \leq b \\
& 1^T x = 1 \\
& (t_i, x_i, 1) \in K_{exp} \quad \forall i = 1 \ldots n
\end{aligned}
$$

```
m, n = 10, 15
A, b = randn(m, n), 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, [i = 1:n], [t[i], x[i], 1] in MOI.ExponentialCone())
optimize!(model)
objective_value(model)
```

```
2.7082832280873395
```

The `MOI.ExponentialCone` has a dual, the `MOI.DualExponentialCone`, that offers an alternative formulation that can be more efficient for some formulations.

There is also the `MOI.RelativeEntropyCone` for explicitly encoding the relative entropy function

```
model = Model(SCS.Optimizer)
set_silent(model)
@variable(model, t)
@variable(model, x[1:n])
@objective(model, Max, -t)
@constraint(model, sum(x) == 1)
@constraint(model, A * x .<= b)
@constraint(model, [t; ones(n); x] in MOI.RelativeEntropyCone(2n + 1))
optimize!(model)
objective_value(model)
```

```
2.7081407172761525
```

PowerCone

The `MOI.PowerCone` is a three-dimensional set parameterized by a scalar value α. It has the form:

$$K_p = \{(x, y, z) \in \mathbb{R}^3 : x^\alpha y^{1-\alpha} \geq |z|, x \geq 0, y \geq 0\}$$

The power cone permits a number of reformulations. For example, when $p > 1$, we can model $t \geq x^p$ using the power cone $(t, 1, x)$ with $\alpha = 1/p$. Thus, to model $t \geq x^3$ with $x \geq 0$

```
model = Model(SCS.Optimizer)
set_silent(model)
@variable(model, t)
@variable(model, x >= 1.5)
@constraint(model, [t, 1, x] in MOI.PowerCone(1 / 3))
@objective(model, Min, t)
optimize!(model)
value(t), value(x)
```

```
(3.374754835774048, 1.4999552263153844)
```

The `MOI.PowerCone` has a dual, the `MOI.DualPowerCone`, that offers an alternative formulation that can be more efficient for some formulations.

P-Norm

The p-norm $||x||_p = \left(\sum_i |x_i|^p \right)^{\frac{1}{p}}$ can be modeled using `MOI.PowerCone`s. See the Mosek Modeling Cookbook for the derivation.

```
function p_norm(x::Vector, p)
    N = length(x)
    model = Model(SCS.Optimizer)
```

```
    set_silent(model)
    @variable(model, r[1:N])
    @variable(model, t)
    @constraint(model, [i = 1:N], [r[i], t, x[i]] in MOI.PowerCone(1 / p))
    @constraint(model, sum(r) == t)
    @objective(model, Min, t)
    optimize!(model)
    return value(t)
end

x = rand(5);
LinearAlgebra.norm(x, 4), p_norm(x, 4)
```

```
(1.1219036317582844, 1.1218941398170095)
```

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:

$$\lambda_1 = \min t$$
$$\text{s.t. } tI - A \succeq 0$$

```
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)
```

```
8.000003377698672
```

GeometricMeanCone

The MOI.GeometricMeanCone is a cone of the form:

$$K_{geo} = \{(t, x) \in \mathbb{R}^n : x \geq 0, t \leq \sqrt[n-1]{x_1 x_2 \cdots x_{n-1}}\}$$

```
model = Model(SCS.Optimizer)
set_silent(model)
@variable(model, x[1:4])
@variable(model, t)
@constraint(model, sum(x) == 1)
@constraint(model, [t; x] in MOI.GeometricMeanCone(5))
optimize!(model)
value(t), value.(x)
```

```
(0.0, [0.2500000011030502, 0.2500000011030505, 0.2500000011030505, 0.2500000011030505])
```

Other Cones and Functions

For other cones supported by JuMP, check out the MathOptInterface Manual.

7.4 Simple semidefinite programming examples

This tutorial is a collection of examples of small conic programs from the field of semidefinite programming (SDP).

This tutorial makes use of the following packages:

```
using JuMP
import LinearAlgebra
import Plots
import Random
import SCS
import Test
```

Maximum cut via SDP

The maximum cut problem is a classical example in graph theory, where we seek to partition a graph into two complementary sets, such that the weight of edges between the two sets is maximized. This problem is NP-hard, but it is possible to obtain an approximate solution using the semidefinite programming relaxation:

$$
\begin{aligned}
\max \quad & 0.25 L \bullet X \\
\text{s.t.} \quad & \text{diag}(X) = e \\
& X \succeq 0
\end{aligned}
$$

where L is the weighted graph Laplacian and e is a vector of ones. For more details, see:

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.

```julia
"""
    svd_cholesky(X::AbstractMatrix, rtol)

Return the matrix `U` of the Cholesky decomposition of `X` as `U' * U`.
Note that we do not use the `LinearAlgebra.cholesky` function because it
requires the matrix to be positive definite while `X` may be only
positive *semi*definite.

We use the convention `U' * U` instead of `U * U'` to be consistent with
`LinearAlgebra.cholesky`.
"""
function svd_cholesky(X::AbstractMatrix)
    F = LinearAlgebra.svd(X)
    # We now have `X ≈ `F.U * D² * F.U'` where:
    D = LinearAlgebra.Diagonal(sqrt.(F.S))
    # So `X ≈ U' * U` where `U` is:
    return (F.U * D)'
end

function solve_max_cut_sdp(weights)
    N = size(weights, 1)
    # Calculate the (weighted) Laplacian of the graph: L = D - W.
    L = LinearAlgebra.diagm(0 => weights * ones(N)) - weights
    model = Model(SCS.Optimizer)
    set_silent(model)
    @variable(model, X[1:N, 1:N], PSD)
    for i in 1:N
        set_start_value(X[i, i], 1.0)
    end
    @objective(model, Max, 0.25 * LinearAlgebra.dot(L, X))
    @constraint(model, LinearAlgebra.diag(X) .== 1)
    optimize!(model)
    V = svd_cholesky(value(X))
    Random.seed!(N)
    r = rand(N)
    r /= LinearAlgebra.norm(r)
    cut = [LinearAlgebra.dot(r, V[:, i]) > 0 for i in 1:N]
    S = findall(cut)
    T = findall(.!cut)
    println("Solution:")
    println(" (S, T) = ({", join(S, ", "), "}, {", join(T, ", "), "})")
    return S, T
end
```

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

Given the graph

```
[1] --- 5 --- [2]
```

The solution is (S, T) = ({1}, {2})

```
S, T = solve_max_cut_sdp([0 5; 5 0])
```

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

Given the graph

```
[1] --- 5 --- [2]
 |  \           |
 |    \         |
 7      6       1
 |        \     |
 |          \   |
[3] --- 1 --- [4]
```

The solution is (S, T) = ({1}, {2, 3, 4})

```
S, T = solve_max_cut_sdp([0 5 7 6; 5 0 0 1; 7 0 0 1; 6 1 1 0])
```

```
([1], [2, 3, 4])
```

Given the graph

```
[1] --- 1 --- [2]
 |               |
 |               |
 5               9
 |               |
 |               |
[3] --- 2 --- [4]
```

The solution is (S, T) = ({1, 4}, {2, 3})

```
S, T = solve_max_cut_sdp([0 1 5 0; 1 0 0 9; 5 0 0 2; 0 9 2 0])
```

```
([1, 4], [2, 3])
```

K-means clustering via SDP

Given a set of points $a_1, \ldots, a_m$ in $\mathbb{R}^n$, allocate them to k clusters.

For more details, see:

Peng, J., & Wei, Y. (2007). Approximating k-means-type clustering via semidefinite programming. SIAM Journal on Optimization, 18(1), 186-205.

```julia
function example_k_means_clustering()
    a = [[2.0, 2.0], [2.5, 2.1], [7.0, 7.0], [2.2, 2.3], [6.8, 7.0], [7.2, 7.5]]
    m = length(a)
    num_clusters = 2
    W = zeros(m, m)
    for i in 1:m, j in i+1:m
        W[i, j] = W[j, i] = exp(-LinearAlgebra.norm(a[i] - a[j]) / 1.0)
    end
    model = Model(SCS.Optimizer)
    set_silent(model)
    @variable(model, Z[1:m, 1:m] >= 0, PSD)
    @objective(model, Min, LinearAlgebra.tr(W * (LinearAlgebra.I - Z)))
    @constraint(model, [i = 1:m], sum(Z[i, :]) .== 1)
    @constraint(model, LinearAlgebra.tr(Z) == num_clusters)
    optimize!(model)
    Z_val = value.(Z)
    current_cluster, visited = 0, Set{Int}()
    for i in 1:m
        if !(i in visited)
            current_cluster += 1
            println("Cluster $current_cluster")
            for j in i:m
                if isapprox(Z_val[i, i], Z_val[i, j]; atol = 1e-3)
                    println(a[j])
                    push!(visited, j)
                end
            end
        end
    end
    return
end

example_k_means_clustering()
```

```
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]
```

The correlation problem

Given three random variables A, B, and C, and given bounds on two of the three correlation coefficients:

$$-0.2 \le AB \le -0.1$$
$$0.4 \le BC \le 0.5$$

our problem is to determine upper and lower bounds on other correlation coefficient AC.

We solve an SDP to make use of the following positive semidefinite property of the correlation matrix:

$$\begin{bmatrix} 1 & AB & AC \\ AB & 1 & BC \\ AC & BC & 1 \end{bmatrix} \succeq 0$$

```julia
function example_correlation_problem()
    model = Model(SCS.Optimizer)
    set_silent(model)
    @variable(model, X[1:3, 1:3], PSD)
    S = ["A", "B", "C"]
    ρ = Containers.DenseAxisArray(X, S, S)
    @constraint(model, [i in S], ρ[i, i] == 1)
    @constraint(model, -0.2 <= ρ["A", "B"] <= -0.1)
    @constraint(model, 0.4 <= ρ["B", "C"] <= 0.5)
    @objective(model, Max, ρ["A", "C"])
    optimize!(model)
    println("An upper bound for ρ_AC is $(value(ρ["A", "C"]))")
    @objective(model, Min, ρ["A", "C"])
    optimize!(model)
    println("A lower bound for ρ_AC is $(value(ρ["A", "C"]))")
    return
end

example_correlation_problem()
```

```
An upper bound for ρ_AC is 0.87192203028973
A lower bound for ρ_AC is -0.9779989594185395
```

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

```
[1]
  \
   1
    \
    [0] —- 1 -- [2]
    /
   1
  /
[3]
```

cannot be exactly embedded into a Euclidean space of any dimension, where distances are computed by length of the shortest path between vertices. A distance-preserving embedding would require the three leaf nodes to form an equilateral triangle of side length 2, with the centre node (0) mapped to an equidistant point at distance 1; this is impossible since the triangle inequality in Euclidean space implies all points would need to be simultaneously collinear.

Here we will formulate and solve an SDP to compute the best possible embedding, that is, the embedding f assigning each vertex v to a vector $f(v)$ that minimizes the distortion c such that

$$D[a,b] \leq ||f(a) - f(b)|| \leq c\, D[a,b]$$

for all edges (a,b) in the graph, where $D[a,b]$ is the distance in the graph metric space.

Any embedding f can be characterized by a Gram matrix Q, which is PSD and such that

$$||f(a) - f(b)||^2 = Q[a,a] + Q[b,b] - 2Q[a,b]$$

The matrix entry $Q[a,b]$ represents the inner product of $f(a)$ with $f(b)$.

We therefore impose the constraint

$$D[a,b]^2 \leq Q[a,a] + Q[b,b] - 2Q[a,b] \leq c^2\, D[a,b]^2$$

for all edges (a,b) in the graph and minimize c^2, which gives us the SDP formulation below. Since we may choose any point to be the origin, we fix the first vertex at 0.

For more details, see:

J. Matoušek (2002), Lectures on discrete geometry, Springer, pp. 378-379

N. Linial (2002), Finite metric spaces–combinatorics, geometry and algorithms, Proceedings of the ICM, Vol. 3, 573-586

```julia
function example_minimum_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, 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
    fix(Q[1, 1], 0)
    @objective(model, Min, c²)
    optimize!(model)
```

```julia
    Test.@test termination_status(model) == OPTIMAL
    Test.@test primal_status(model) == FEASIBLE_POINT
    Test.@test objective_value(model) ≈ 4 / 3 atol = 1e-4
    # Recover the minimal distorted embedding:
    X = [zeros(3) sqrt(value.(Q)[2:end, 2:end])]
    return Plots.plot(
        X[1, :],
        X[2, :],
        X[3, :];
        seriestype = :mesh3d,
        connections = ([0, 0, 0, 1], [1, 2, 3, 2], [2, 3, 1, 3]),
        legend = false,
        fillalpha = 0.1,
        lw = 3,
        ratio = :equal,
        xlim = (-1.1, 1.1),
        ylim = (-1.1, 1.1),
        zlim = (-1.5, 1.0),
        zticks = -1:1,
        camera = (60, 30),
    )
end

example_minimum_distortion()
```
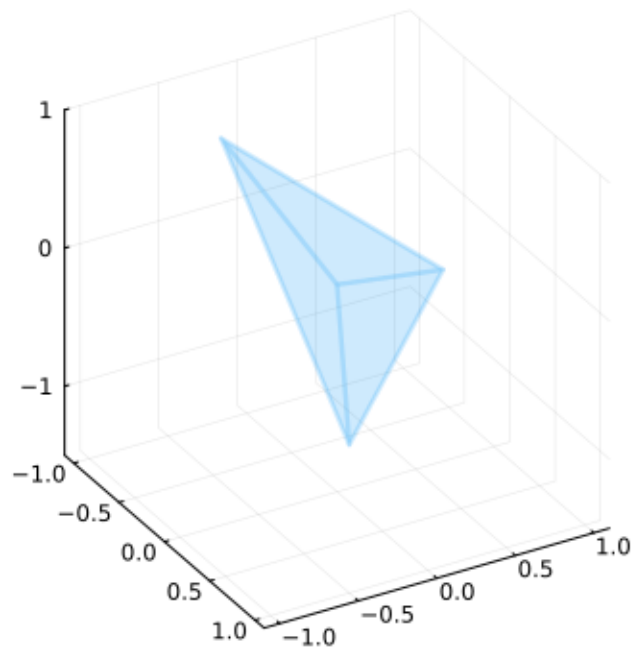
Robust uncertainty sets

This example computes the Value at Risk for a data-driven uncertainty set. Closed-form expressions for the optimal value are available. For more details, see:

Bertsimas, D., Gupta, V., & Kallus, N. (2018). Data-driven robust optimization. Mathematical Programming, 167, 235-292.

```julia
function example_robust_uncertainty_sets()
    R, d, , ε = 1, 3, 0.05, 0.05
    N = ceil((2 + 2 * log(2 / ))^2) + 1
    c, μhat, M = randn(d), rand(d), 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] >= 0, PSDCone())
    @objective(model, Max, c' * u)
    optimize!(model)
    exact =
        μhat' * c +
        Γ1( / 2, N) * LinearAlgebra.norm(c) +
        sqrt((1 - ε) / ε) *
        sqrt(c' * (Σhat + Γ2( / 2, N) * LinearAlgebra.I) * c)
    Test.@test objective_value(model) ≈ exact atol = 1e-2
    return
end

example_robust_uncertainty_sets()
```

7.5 Arbitrary precision arithmetic

The purpose of this tutorial is to explain how to use a solver which supports arithmetic using a number type other than `Float64`.

This tutorial uses the following packages:

```julia
using JuMP
import CDDLib
import Clarabel
```

Higher-precision arithmetic

To create a model with a number type other than Float64, use GenericModel{T} with an optimizer which supports the same number type:

```
model = GenericModel{BigFloat}(Clarabel.Optimizer{BigFloat})
```

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

> **Tip**
>
> Model() is short-hand for GenericModel{Float64}().

The syntax for adding decision variables is the same as a normal JuMP model, except that values are converted to BigFloat:

```
@variable(model, -1 <= x[1:2] <= sqrt(big"2"))
```

```
2-element Vector{GenericVariableRef{BigFloat}}:
 x[1]
 x[2]
```

Note that each x is now a GenericVariableRef{BigFloat}, which means that the value of x in a solutionn will be a BigFloat.

The lower and upper bounds of the decision variables are also BigFloat:

```
lower_bound(x[1])
```

```
-1.0
```

```
typeof(lower_bound(x[1]))
```

```
BigFloat
```

```
upper_bound(x[2])
```

```
1.4142135623730950488016887242096980785696718753769480731766797379907324784621\02
```

```
typeof(upper_bound(x[2]))
```

```
BigFloat
```

The syntax for adding constraints is the same as a normal JuMP model, except that coefficients are converted to `BigFloat`:

```
@constraint(model, c, x[1] == big"2" * x[2])
```

$c : x_1 - 2.0x_2 = 0.0$

The function is a `GenericAffExpr` with `BigFloat` for the coefficient and variable types;

```
constraint = constraint_object(c)
typeof(constraint.func)
```

```
GenericAffExpr{BigFloat, GenericVariableRef{BigFloat}}
```

and the set is a `MOI.EqualTo{BigFloat}`:

```
typeof(constraint.set)
```

```
MathOptInterface.EqualTo{BigFloat}
```

The syntax for adding and objective is the same as a normal JuMP model, except that coefficients are converted to `BigFloat`:

```
@objective(model, Min, 3x[1]^2 + 2x[2]^2 - x[1] - big"4" * x[2])
```

$$3.0x_1^2 + 2.0x_2^2 - x_1 - 4.0x_2$$

```
typeof(objective_function(model))
```

```
GenericQuadExpr{BigFloat, GenericVariableRef{BigFloat}}
```

Here's the model we have built:

```
print(model)
```

```
Min 3.0 x[1]² + 2.0 x[2]² - x[1] - 4.0 x[2]
Subject to
 c : x[1] - 2.0 x[2] = 0.0
 x[1] ≥ -1.0
 x[2] ≥ -1.0
 x[1] ≤ 1.4142135623730950488016887242096980785696718753769480731766797373799073247846210_2
 x[2] ≤ 1.4142135623730950488016887242096980785696718753769480731766797373799073247846210_2
```

Let's solve and inspect the solution:

```
optimize!(model)
solution_summary(model)
```

```
* Solver : Clarabel

* Status
  Result count       : 1
  Termination status : OPTIMAL
  Message from the solver:
  "SOLVED"

* Candidate solution (result #1)
  Primal status       : FEASIBLE_POINT
  Dual status         : FEASIBLE_POINT
  Objective value     : -6.42857e-01
  Dual objective value : -6.42857e-01

* Work counters
  Solve time (sec)   : 1.85134e-03
  Barrier iterations : 5
```

The value of each decision variable is a BigFloat:

```
value.(x)
```

```
2-element Vector{BigFloat}:
 0.4285714246558161076147072906813123533593766450416896337912086518811186790735016
 0.2142857123279078924828007272730108809297577877991360649674411645247653239673974
```

as well as other solution attributes like the objective value:

```
objective_value(model)
```

```
-0.6428571428571422964607590389935242587959291815638830868454759876473734138856053
```

and dual solution:

```
dual(c)
```

```
1.5714285719771408453439780690150921905482509197879450650220590710525570478880015
```

This problem has an analytic solution of `x = [3//7, 3//14]`. Currently, our solution has an error of approximately `1e-9`:

```
value.(x) .- [3 // 7, 3 // 14]
```

```
2-element Vector{BigFloat}:
 -3.915612463813864137890116218069194783529738937637362776690309892355071064813317e-09
 -1.957806393231484987012703404784527926486578220746844549760948961746888943071481e-09
```

But by reducing the tolerances, we can obtain a more accurate solution:

```
set_attribute(model, "tol_gap_abs", 1e-32)
set_attribute(model, "tol_gap_rel", 1e-32)
optimize!(model)
value.(x) .- [3 // 7, 3 // 14]
```

```
2-element Vector{BigFloat}:
 -4.120732596246374574619292889406407106157605546563218305172773512099467866195165e-32
 -7.146646610782677659152301436088423235900252780211057986251367981130623553333357e-32
```

Rational arithmetic

In addition to higher-precision floating point number types like `BigFloat`, JuMP also supports solvers with exact rational arithmetic. One example is CDDLib.jl, which supports the `Rational{BigInt}` number type:

```
model = GenericModel{Rational{BigInt}}(CDDLib.Optimizer{Rational{BigInt}})
```

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

As before, we can create variables using rational bounds:

```
@variable(model, 1 // 7 <= x[1:2] <= 2 // 3)
```

```
2-element Vector{GenericVariableRef{Rational{BigInt}}}:
 x[1]
 x[2]
```

```
lower_bound(x[1])
```

```
1//7
```

```
typeof(lower_bound(x[1]))
```

```
Rational{BigInt}
```

As well as constraints:

```
@constraint(model, c1, (2 // 1) * x[1] + x[2] <= 1)
```

c1 : $2//1x_1 + x_2 \leq 1//1$

```
@constraint(model, c2, x[1] + 3x[2] <= 9 // 4)
```

c2 : $x_1 + 3//1x_2 \leq 9//4$

and objective functions:

```
@objective(model, Max, sum(x))
```

$$x_1 + x_2$$

Here's the model we have built:

```
print(model)
```

```
Max x[1] + x[2]
Subject to
 c1 : 2//1 x[1] + x[2] ≤ 1//1
 c2 : x[1] + 3//1 x[2] ≤ 9//4
 x[1] ≥ 1//7
 x[2] ≥ 1//7
 x[1] ≤ 2//3
 x[2] ≤ 2//3
```

Let's solve and inspect the solution:

```
optimize!(model)
solution_summary(model)
```

```
* Solver : CDD

* Status
  Result count       : 1
  Termination status : OPTIMAL
  Message from the solver:
  "Optimal"

* Candidate solution (result #1)
  Primal status      : FEASIBLE_POINT
  Dual status        : NO_SOLUTION
  Objective value    : 5//6

* Work counters
```

The optimal values are given in exact rational arithmetic:

```
value.(x)
```

```
2-element Vector{Rational{BigInt}}:
 1//6
 2//3
```

```
objective_value(model)
```

```
5//6
```

```
value(c2)
```

```
13//6
```

7.6 Logistic regression

This tutorial was 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

$$l(\theta) = \sum_{i=1}^{n} \log\left(\frac{1}{1 + \exp(-y_i \theta^\top x_i)}\right)$$

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

$$\min_{t,r,\theta} \sum_{i=1}^{n} t_i + \lambda r$$
$$\text{subject to} \quad t_i \geq \log(1 + \exp(-y_i \theta^\top x_i))$$
$$r \geq \|\theta\|$$

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

$$\begin{cases} (u_i - t_i, 1, z_{i1}) \in K_{exp} \\ (-t_i, 1, z_{i2}) \in K_{exp} \\ \quad z_{i1} + z_{i2} \leq 1 \end{cases}$$

In this setting, the conic version of the logistic regression problems writes out

$$\min_{t,z,r,\theta} \sum_{i=1}^{n} t_i + \lambda r$$
$$\text{subject to} \quad (u_i - t_i, 1, z_{i1}) \in K_{exp}$$
$$(-t_i, 1, z_{i2}) \in K_{exp}$$
$$z_{i1} + z_{i2} \leq 1$$
$$u_i = -y_i x_i^\top \theta$$
$$r \geq \|\theta\|$$

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.

```julia
using JuMP
import Random
import SCS
```

> **Info**
>
> This tutorial uses sets from MathOptInterface. By default, JuMP exports the MOI symbol as an alias for the MathOptInterface.jl package. We recommend making this more explicit in your code by adding the following lines:
>
> ```julia
> import MathOptInterface as MOI
> ```

```julia
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.

```julia
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
```

```julia
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.

```julia
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
```

```julia
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.

```julia
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 SecondOrderCone())
    # 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.

```julia
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)
```

```julia
θ♯ = JuMP.value.(model[:θ])
```

```
11-element Vector{Float64}:
  0.0015739330765939647
  0.6238309104953497
 -0.3606804744097859
  0.1671169113646527
  0.24900921085425312
 -0.49972910059326164
 -0.4648274703366815
  0.4218951914724902
 -0.14975368883728812
```

```
  0.027573776938405868
 -0.1251381661716098
```

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 formulate the sparse logistic regression problem with the help of a `MOI.NormOneCone` set.

```julia
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)
```

```julia
θ♯ = JuMP.value.(sparse_model[:θ])
println(
    "Number of non-zero components: ",
    count_nonzero(θ♯),
    " (out of ",
    p,
    " features)",
)
```

```
Number of non-zero components: 7 (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:

$$\min_{\theta} \ \sum_{i=1}^{n} \log(1 + \exp(-y_i \theta^\top x_i)) + \lambda \|\theta\|_2^2$$
$$\text{subject to} \quad \|\theta\|_0 <= k$$

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.

7.7   Experiment design

This tutorial was originally contributed by Arpit Bhatia and 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
```

> **Info**
>
> This tutorial uses sets from MathOptInterface. By default, JuMP exports the MOI symbol as an alias for the MathOptInterface.jl package. We recommend making this more explicit in your code by adding the following lines:
>
> ```
> import MathOptInterface as MOI
> ```

We set a seed so the random numbers are repeatable:

```
Random.seed!(1234)
```

```
MersenneTwister(1234)
```

The 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, that is, $m_1, \ldots, m_p$ 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:

$$\min \left(\text{w.r.t.} \mathbf{S}_+^n \right) E = \left(\sum_{j=1}^{p} m_j v_j v_j^T \right)^{-1}$$

$$\text{subject to} \, m_i \geq 0$$

$$\sum_{i=1}^{p} m_i = m$$

$$m_i \in \mathbb{Z}, \quad i = 1, \ldots, p$$

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:

$$\min \left(\text{w.r.t.} \mathbf{S}_+^n \right) E = (1/m) \left(\sum_{i=1}^{p} \lambda_i v_i v_i^T \right)^{-1}$$

$$\text{subject to:} \quad \lambda \succeq 0$$

$$\mathbf{1}^T \lambda = 1$$

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
n_max = 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:

$$\mathbf{E}\|e\|_2^2 = \mathbf{E}\operatorname{tr}\left(ee^T\right) = \operatorname{tr}E$$

The A-optimal experiment design problem in SDP form is

$$\min \mathbf{1}^T u$$

$$\text{subject to } \begin{bmatrix} \displaystyle\sum_{i=1}^{p} \lambda_i v_i v_i^T & e_k \\ & e_k^T u_k \end{bmatrix} \succeq 0, \quad k = 1, \ldots, n$$

$$\lambda \succeq 0$$

$$\mathbf{1}^T \lambda = 1$$

```
aOpt = Model(SCS.Optimizer)
set_silent(aOpt)
@variable(aOpt, np[1:p], lower_bound = 0, upper_bound = n_max)
@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)
```

```
5.0412636484980755
```

```
value.(np)
```

```
8-element Vector{Float64}:
 1.747945718805369
 1.1153130074864266
 1.288808299639784e-7
 1.661952499684036
 2.9999994121581453
 0.8414258334554227
 1.3825595386035296
 2.25080346403116
```

E-optimal design

In E-optimal design, we minimize the norm of the error covariance matrix, that is, 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:

$$\min t$$
$$\text{subject to} \sum_{i=1}^{p} \lambda_i v_i v_i^T \succeq tI$$
$$\lambda \succeq 0$$
$$\mathbf{1}^T \lambda = 1$$

```
eOpt = Model(SCS.Optimizer)
set_silent(eOpt)
@variable(eOpt, 0 <= np[1:p] <= n_max)
@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)
```

```
0.44896306556366
```

```
value.(np)
```

```
8-element Vector{Float64}:
  2.9999591973576862
  0.6746121070433458
 -4.077828668073517e-5
  1.0453736634429782
  2.999976477144745
  1.7873172744476913
  0.3017495901336785
  2.1911100940002943
```

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:

$$\min \log \det \left(\sum_{i=1}^{p} \lambda_i v_i v_i^T \right)^{-1}$$
$$\text{subject to} \lambda \succeq 0$$
$$\mathbf{1}^T \lambda = 1$$

```
dOpt = Model(SCS.Optimizer)
set_silent(dOpt)
@variable(dOpt, np[1:p], lower_bound = 0, upper_bound = n_max)
@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)
```

```
0.19012913442822837
```

```
value.(np)
```

```
8-element Vector{Float64}:
 8.953314019826512e-5
 2.5660635420070887
 0.00014372289278898837
 0.26257121910114045
 2.9421084590824993
 2.392124018059283
 2.8368056348962
 0.9999027705763397
```

7.8   Minimal ellipses

This example comes from section 8.4.1 of the book Convex Optimization by Boyd and Vandenberghe (2004).

Formulation

Given a set of m ellipses of the form:

$$E(A, b, c) = \{x : x^\top A x + 2b^\top x + c \leq 0\},$$

the minimal ellipse problem finds an ellipse with the minimum area that encloses the given ellipses.

It is convenient to parameterize the minimal enclosing ellipse as

$$\{x : ||Px + q|| \leq 1\}.$$

Then the optimal P and q are given by the convex semidefinite program;

$$
\begin{aligned}
\text{maximize} \quad & \log(\det(P)) \\
\text{subject to} \quad & \tau_i \geq 0, && i = 1, \ldots, m \\
& \begin{bmatrix} P^2 - \tau_i A_i & Pq - \tau_i b_i & 0 \\ (Pq - \tau_i b_i)^\top & -1 - \tau_i c_i & (Pq)^\top \\ 0 & (Pq) & -P^2 \end{bmatrix} \preceq 0 \text{ (PSD)} && i = 1, \ldots, m
\end{aligned}
$$

with helper variables τ.

Required packages

This tutorial uses the following packages:

```julia
using JuMP
import LinearAlgebra
import Plots
import SCS
```

Data

First, define the m input ellipses (here $m = 6$), parameterized as $x^T A_i x + 2b_i^T x + c \leq 0$:

```
struct Ellipse
    A::Matrix{Float64}
    b::Vector{Float64}
    c::Float64
    function Ellipse(A::Matrix{Float64}, b::Vector{Float64}, c::Float64)
        @assert isreal(A) && LinearAlgebra.issymmetric(A)
        return new(A, b, c)
    end
end

ellipses = [
    Ellipse([1.2576 -0.3873; -0.3873 0.3467], [0.2722, 0.1969], 0.1831),
    Ellipse([1.4125 -2.1777; -2.1777 6.7775], [-1.228, -0.0521], 0.3295),
    Ellipse([1.7018 0.8141; 0.8141 1.7538], [-0.4049, 1.5713], 0.2077),
    Ellipse([0.9742 -0.7202; -0.7202 1.5444], [0.0265, 0.5623], 0.2362),
    Ellipse([0.6798 -0.1424; -0.1424 0.6871], [-0.4301, -1.0157], 0.3284),
    Ellipse([0.1796 -0.1423; -0.1423 2.6181], [-0.3286, 0.557], 0.4931),
];
```

We visualise the ellipses using the Plots package:

```
function plot_ellipse(plot, ellipse::Ellipse)
    A, b, c = ellipse.A, ellipse.b, ellipse.c
    θ = range(0, 2pi + 0.05; step = 0.05)
    # Some linear algebra to convert θ into (x,y) coordinates.
    x_y = √A \ (√(b' * (A \ b) - c) .* hcat(cos.(θ), sin.(θ)) .- (√A \ b)')'
    Plots.plot!(plot, x_y[1, :], x_y[2, :]; label = nothing, c = :navy)
    return
end

plot = Plots.plot(; size = (600, 600))
for ellipse in ellipses
    plot_ellipse(plot, ellipse)
end
plot
```
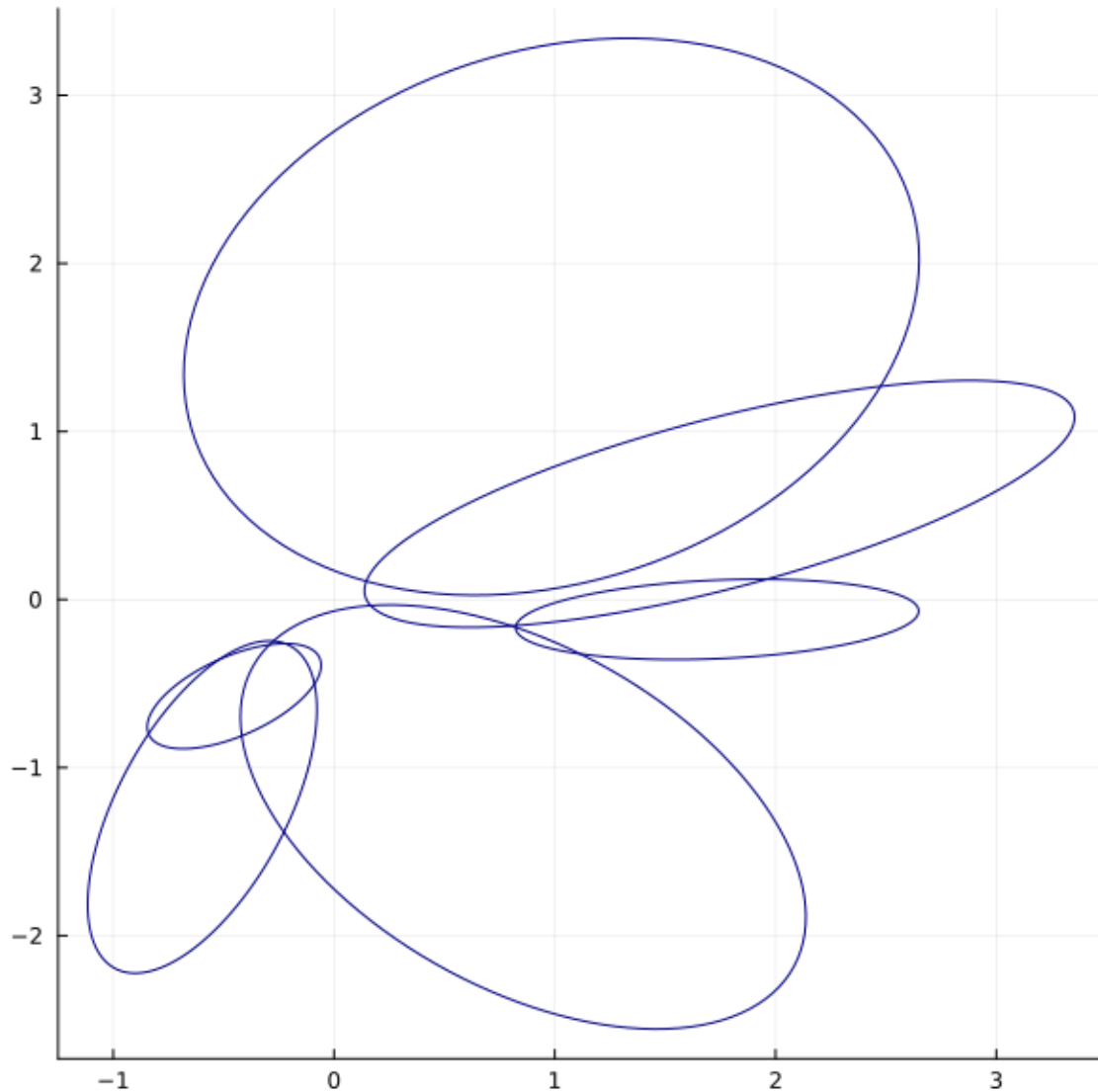
Build the model

Now let's build the model, using the change-of-variables P² = P^2 and P_q = Pq. We'll recover the true value of P and q after the solve.

```
model = Model(SCS.Optimizer)
# We need to use a tighter tolerance for this example, otherwise the bounding
# ellipse won't actually be bounding...
set_attribute(model, "eps_rel", 1e-6)
set_silent(model)
m, n = length(ellipses), size(first(ellipses).A, 1)
@variable(model, τ[1:m] >= 0)
@variable(model, P²[1:n, 1:n], PSD)
@variable(model, P_q[1:n])
```

```julia
for (i, ellipse) in enumerate(ellipses)
    A, b, c = ellipse.A, ellipse.b, ellipse.c
    X = [
        #! format: off
        (P² - τ[i] * A)    (P_q - τ[i] * b) zeros(n, n)
        (P_q - τ[i] * b)' (-1 - τ[i] * c)  P_q'
        zeros(n, n)         P_q                 -P²
        #! format: on
    ]
    @constraint(model, LinearAlgebra.Symmetric(X) <= 0, PSDCone())
end
```

We cannot directly represent the objective $\log(\det(P))$, so we introduce the conic reformulation:

```julia
@variable(model, log_det_P)
@constraint(model, [log_det_P; 1; vec(P²)] in MOI.LogDetConeSquare(n))
@objective(model, Max, log_det_P)
```

$$log\_det\_P$$

Now, solve the program:

```julia
optimize!(model)
solution_summary(model)
```

```
* Solver : SCS

* Status
  Result count       : 1
  Termination status : OPTIMAL
  Message from the solver:
  "solved"

* Candidate solution (result #1)
  Primal status      : FEASIBLE_POINT
  Dual status        : FEASIBLE_POINT
  Objective value    : -4.04358e+00
  Dual objective value : -4.04364e+00

* Work counters
  Solve time (sec)   : 3.47750e-01
```

Results

After solving the model to optimality we can recover the solution in terms of P and q:

```julia
P = sqrt(value.(P²))
q = P \ value.(P_q)
```

```
2-element Vector{Float64}:
 -0.3964217694071521
 -0.02139417888886267
```

Finally, overlaying the solution in the plot we see the minimal area enclosing ellipsoid:

```
Plots.plot!(
    plot,
    [tuple(P \ [cos(θ) - q[1], sin(θ) - q[2]]...) for θ in 0:0.05:(2pi+0.05)];
    c = :crimson,
    label = nothing,
)
```
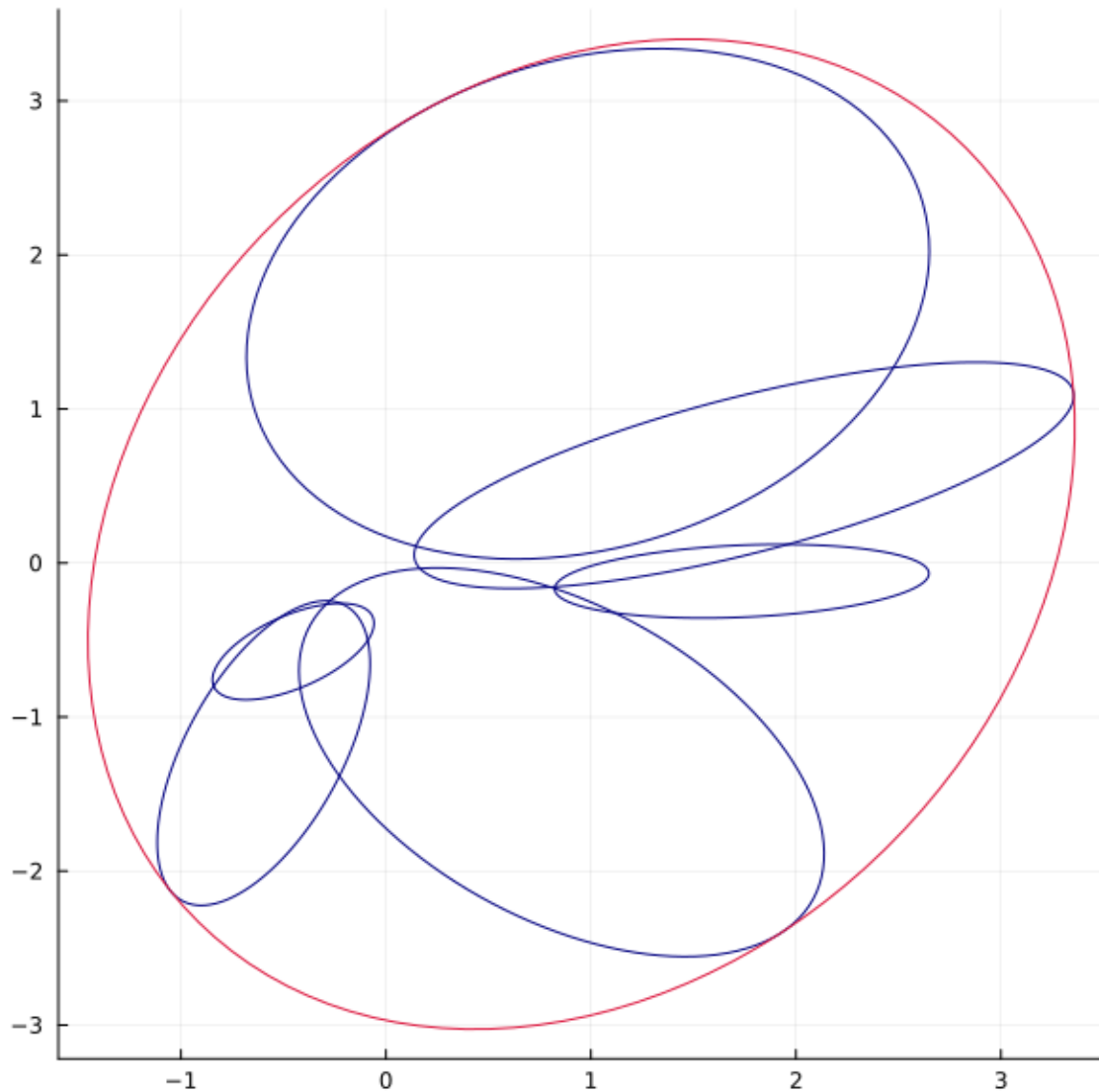
7.9 Ellipsoid approximation

This tutorial considers the problem of computing extremal ellipsoids: finding ellipsoids that best approximate a given set. As an extension, we show how to use JuMP to inspect the bridges that were used, and how to explore alternative formulations.

The model comes from Section 4.9 "Applications VII: extremal ellipsoids" of the book Lectures on Modern Convex Optimization by Ben-Tal and Nemirovski (2001).

For a related example, see also the Minimal ellipses tutorial.

Problem formulation

Suppose that we are given a set $\mathcal{S}$ consisting of m points in n-dimensional space:

$$\mathcal{S} = \{x_1, \ldots, x_m\} \subset \mathbb{R}^n$$

Our goal is to determine an optimal vector $c \in \mathbb{R}^n$ and an optimal $n \times n$ real symmetric matrix D such that the ellipse:

$$E(D, c) = \{x : (x - c)^\top D(x - c) \leq 1\},$$

contains $\mathcal{S}$ and has the smallest possible volume.

The optimal D and c are given by the optimization problem:

$$
\begin{aligned}
\max \quad & t \\
\text{s.t.} \quad & Z \succeq 0 \\
& \begin{bmatrix} s & z^\top \\ z & Z \end{bmatrix} \succeq 0 \\
& x_i^\top Z x_i - 2 x_i^\top z + s \leq 1 \qquad i = 1, \ldots, m \\
& t \leq \sqrt[n]{\det(Z)},
\end{aligned}
$$

where $D = Z_*$ and $c = Z_*^{-1} z_*$.

Required packages

This tutorial uses the following packages:

```
using JuMP
import LinearAlgebra
import Plots
import Random
import SCS
```

Data

We first need to generate some points to work with.

```
function generate_point_cloud(
    m;              # number of 2-dimensional points
    a = 10,         # scaling in x direction
    b = 2,          # scaling in y direction
    rho = π / 6,   # rotation of points around origin
    random_seed = 1,
)
    rng = Random.MersenneTwister(random_seed)
    P = randn(rng, Float64, m, 2)
    Phi = [a*cos(rho) a*sin(rho); -b*sin(rho) b*cos(rho)]
    S = P * Phi
    return S
end
```

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

For the sake of this example, let's take $m = 600$:

```
S = generate_point_cloud(600);
```

We will visualise the points (and ellipse) using the Plots package:

```
r = 1.1 * maximum(abs.(S))
plot = Plots.scatter(
    S[:, 1],
    S[:, 2];
    xlim = (-r, r),
    ylim = (-r, r),
    label = nothing,
    c = :green,
    shape = :x,
    size = (600, 600),
)
```
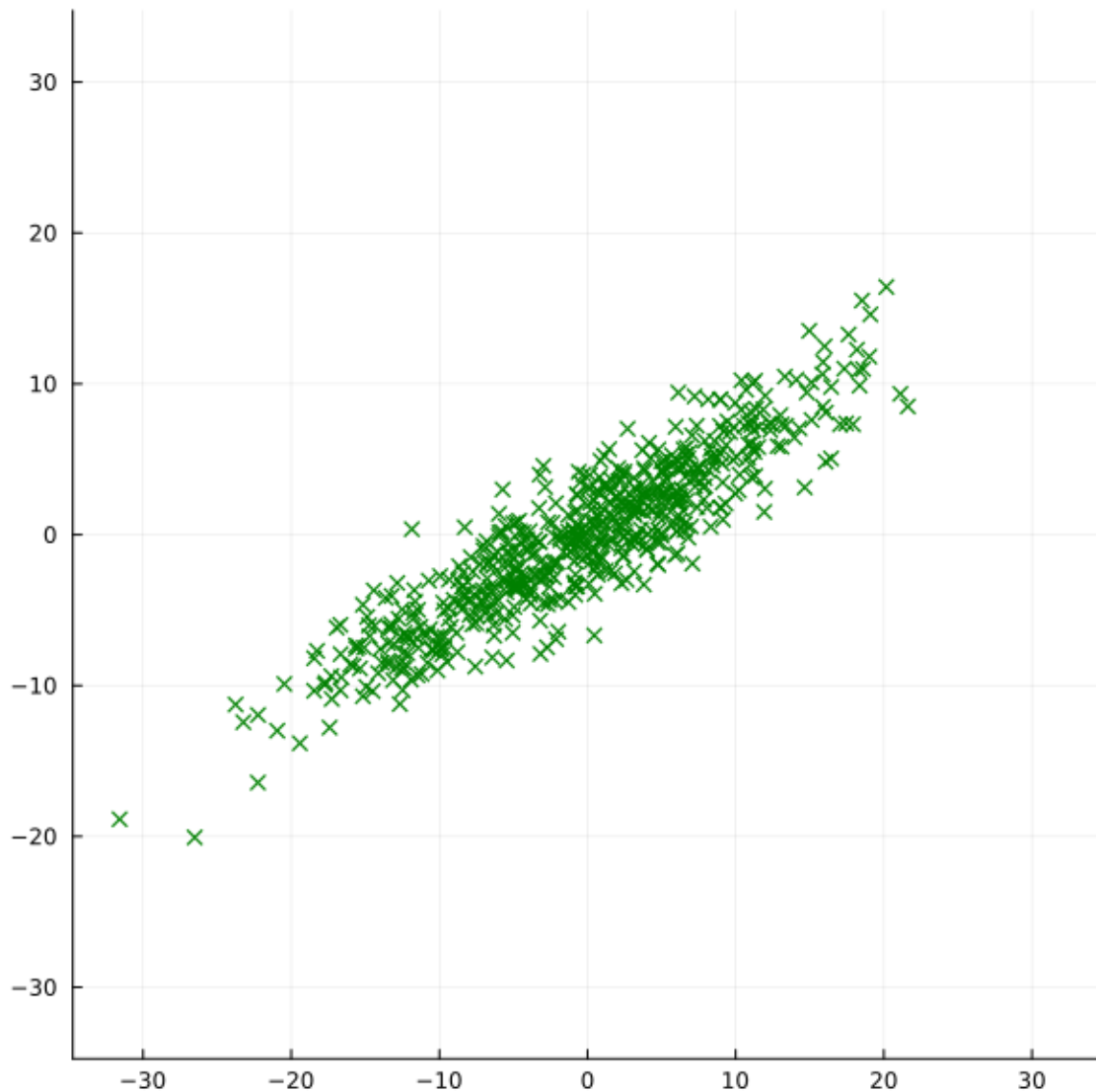
JuMP formulation

Now let's build and the JuMP model. We'll compute D and c after the solve.

```
model = Model(SCS.Optimizer)
# We need to use a tighter tolerance for this example, otherwise the bounding
# ellipse won't actually be bounding...
set_attribute(model, "eps_rel", 1e-6)
set_silent(model)
m, n = size(S)
@variable(model, z[1:n])
@variable(model, Z[1:n, 1:n], PSD)
@variable(model, s)
@variable(model, t)
@constraint(model, [s z'; z Z] >= 0, PSDCone())
```

```
@constraint(
    model,
    [i in 1:m],
    S[i, :]' * Z * S[i, :] - 2 * S[i, :]' * z + s <= 1,
)
@constraint(model, [t; vec(Z)] in MOI.RootDetConeSquare(n))
@objective(model, Max, t)
optimize!(model)
solution_summary(model)
```

```
* Solver : SCS

* Status
  Result count       : 1
  Termination status : OPTIMAL
  Message from the solver:
  "solved"

* Candidate solution (result #1)
  Primal status      : FEASIBLE_POINT
  Dual status        : FEASIBLE_POINT
  Objective value    : 4.50777e-03
  Dual objective value : 4.47966e-03

* Work counters
  Solve time (sec)   : 9.60489e-01
```

Results

After solving the model to optimality we can recover the solution in terms of D and c:

```
D = value.(Z)
```

```
2×2 Matrix{Float64}:
  0.00707822  -0.0102173
 -0.0102173    0.0175624
```

```
c = D \ value.(z)
```

```
2-element Vector{Float64}:
 -3.2480287747908188
 -1.8428256866102324
```

Finally, overlaying the solution in the plot we see the minimal volume approximating ellipsoid:

```
P = sqrt(D)
q = -P * c
data = [tuple(P \ [cos(θ) - q[1], sin(θ) - q[2]]...) for θ in 0:0.05:(2pi+0.05)]
Plots.plot!(plot, data; c = :crimson, label = nothing)
```
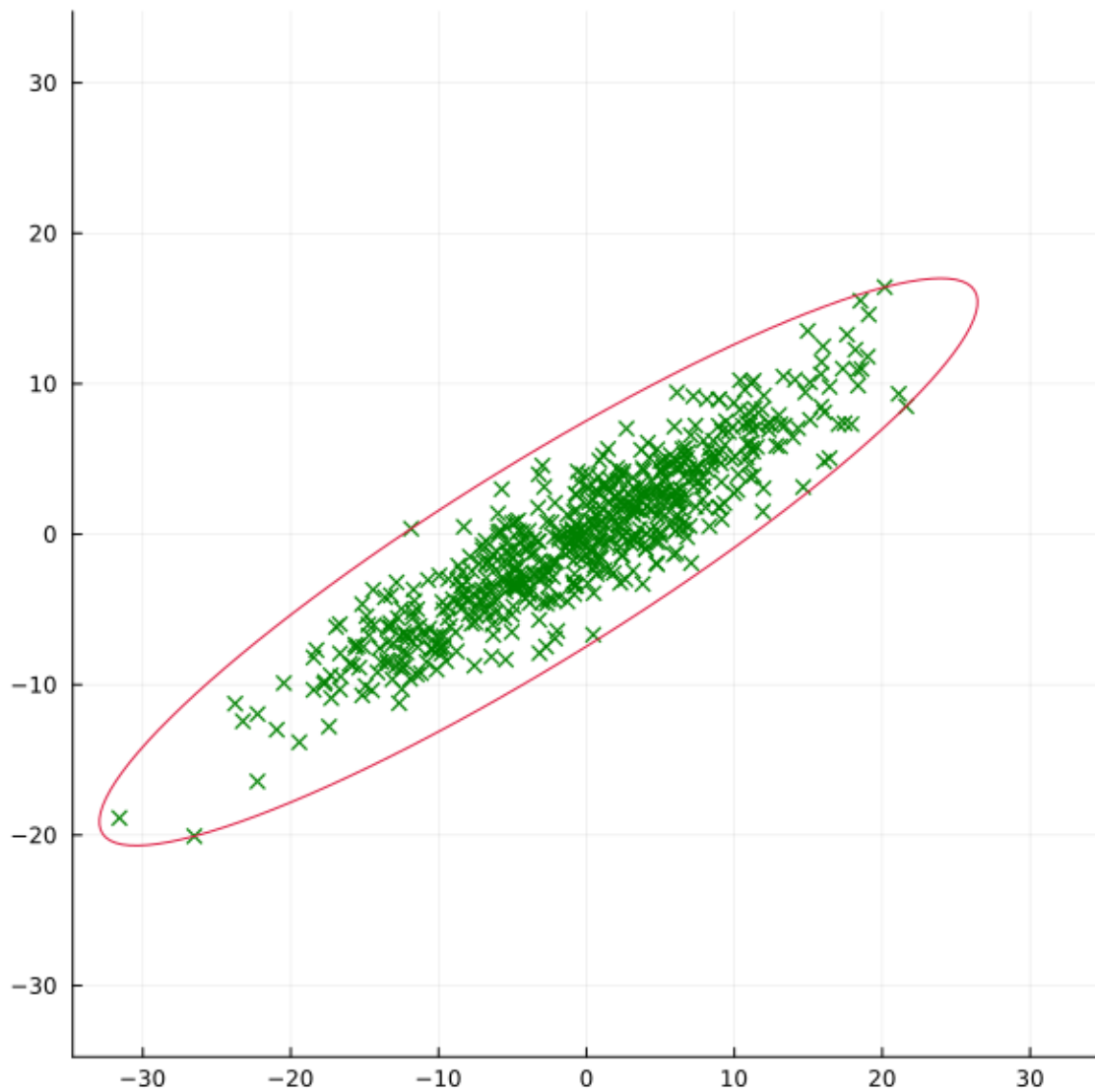

Alternative formulations

The formulation of `model` uses `MOI.RootDetConeSquare`. However, because SCS does not natively support this cone, JuMP automatically reformulates the problem into an equivalent problem that SCS does support. You can see the reformulation that JuMP chose using `print_active_bridges`:

```
print_active_bridges(model)
```

```
* Unsupported objective: MOI.VariableIndex
|  bridged by:
|   MOIB.Objective.FunctionizeBridge{Float64}
|  may introduce:
|   * Supported objective: MOI.ScalarAffineFunction{Float64}
* Unsupported constraint: MOI.ScalarAffineFunction{Float64}-in-MOI.LessThan{Float64}
|  bridged by:
|   MOIB.Constraint.LessToGreaterBridge{Float64, MOI.ScalarAffineFunction{Float64},
↪  MOI.ScalarAffineFunction{Float64}}
|  may introduce:
|   * Unsupported constraint: MOI.ScalarAffineFunction{Float64}-in-MOI.GreaterThan{Float64}
|   |  bridged by:
|   |   MOIB.Constraint.VectorizeBridge{Float64, MOI.VectorAffineFunction{Float64},
↪  MOI.Nonnegatives, MOI.ScalarAffineFunction{Float64}}
|   |  may introduce:
|   |   * Supported constraint: MOI.VectorAffineFunction{Float64}-in-MOI.Nonnegatives
* Unsupported constraint: MOI.VectorAffineFunction{Float64}-in-MOI.PositiveSemidefiniteConeSquare
|  bridged by:
|   MOIB.Constraint.SquareBridge{Float64, MOI.VectorAffineFunction{Float64},
↪  MOI.ScalarAffineFunction{Float64}, MOI.PositiveSemidefiniteConeTriangle,
↪  MOI.PositiveSemidefiniteConeSquare}
|  may introduce:
|   * Unsupported constraint:
↪  MOI.VectorAffineFunction{Float64}-in-MOI.PositiveSemidefiniteConeTriangle
|   |  bridged by:
|   |   SCS.ScaledPSDConeBridge{Float64, MOI.VectorAffineFunction{Float64}}
|   |  may introduce:
|   |   * Supported constraint: MOI.VectorAffineFunction{Float64}-in-SCS.ScaledPSDCone
|   * Unsupported constraint: MOI.ScalarAffineFunction{Float64}-in-MOI.EqualTo{Float64}
|   |  bridged by:
|   |   MOIB.Constraint.VectorizeBridge{Float64, MOI.VectorAffineFunction{Float64}, MOI.Zeros,
↪  MOI.ScalarAffineFunction{Float64}}
|   |  may introduce:
|   |   * Supported constraint: MOI.VectorAffineFunction{Float64}-in-MOI.Zeros
* Unsupported constraint: MOI.VectorOfVariables-in-MOI.PositiveSemidefiniteConeTriangle
|  bridged by:
|   SCS.ScaledPSDConeBridge{Float64, MOI.VectorOfVariables}
|  may introduce:
|   * Supported constraint: MOI.VectorAffineFunction{Float64}-in-SCS.ScaledPSDCone
* Unsupported constraint: MOI.VectorOfVariables-in-MOI.RootDetConeSquare
|  bridged by:
|   MOIB.Constraint.SquareBridge{Float64, MOI.VectorOfVariables,
↪  MOI.ScalarAffineFunction{Float64}, MOI.RootDetConeTriangle, MOI.RootDetConeSquare}
|  may introduce:
|   * Unsupported constraint: MOI.VectorOfVariables-in-MOI.RootDetConeTriangle
|   |  bridged by:
|   |   MOIB.Constraint.VectorFunctionizeBridge{Float64, MOI.RootDetConeTriangle}
|   |  may introduce:
|   |   * Unsupported constraint: MOI.VectorAffineFunction{Float64}-in-MOI.RootDetConeTriangle
|   |   |  bridged by:
|   |   |   MOIB.Constraint.RootDetBridge{Float64, MOI.VectorAffineFunction{Float64},
↪  MOI.VectorAffineFunction{Float64}, MOI.VectorAffineFunction{Float64}}
|   |   |  may introduce:
|   |   |   * Unsupported constraint:
↪  MOI.VectorAffineFunction{Float64}-in-MOI.PositiveSemidefiniteConeTriangle
```

```
|   |   |   |   bridged by:
|   |   |   |    SCS.ScaledPSDConeBridge{Float64, MOI.VectorAffineFunction{Float64}}
|   |   |   |   may introduce:
|   |   |   |    * Supported constraint: MOI.VectorAffineFunction{Float64}-in-SCS.ScaledPSDCone
|   |   |   * Unsupported constraint: MOI.VectorAffineFunction{Float64}-in-MOI.GeometricMeanCone
|   |   |   |   bridged by:
|   |   |   |    MOIB.Constraint.GeoMeanToPowerBridge{Float64, MOI.VectorAffineFunction{Float64}}
|   |   |   |   may introduce:
|   |   |   |    * Supported constraint:
↪   MOI.VectorAffineFunction{Float64}-in-MOI.PowerCone{Float64}
|   |   |   |    * Unsupported variable: MOI.Nonnegatives
|   |   |   |    |  adding as constraint:
|   |   |   |    |   * Supported variable: MOI.Reals
|   |   |   |    |   * Unsupported constraint: MOI.VectorOfVariables-in-MOI.Nonnegatives
|   |   |   |    |   |  bridged by:
|   |   |   |    |   |   MOIB.Constraint.VectorFunctionizeBridge{Float64, MOI.Nonnegatives}
|   |   |   |    |   |  may introduce:
|   |   |   |    |   |   * Supported constraint:
↪   MOI.VectorAffineFunction{Float64}-in-MOI.Nonnegatives
|   |   |    * Supported variable: MOI.Reals
|    * Unsupported constraint: MOI.ScalarAffineFunction{Float64}-in-MOI.EqualTo{Float64}
|   |    bridged by:
|   |     MOIB.Constraint.VectorizeBridge{Float64, MOI.VectorAffineFunction{Float64}, MOI.Zeros,
↪   MOI.ScalarAffineFunction{Float64}}
|   |    may introduce:
|   |     * Supported constraint: MOI.VectorAffineFunction{Float64}-in-MOI.Zeros
```

There's a lot going on here, but the first bullet is:

```
* Unsupported objective: MOI.VariableIndex
|   bridged by:
|    MOIB.Objective.FunctionizeBridge{Float64}
|   introduces:
|    * Supported objective: MOI.ScalarAffineFunction{Float64}
```

This says that SCS does not support a MOI.VariableIndex objective function, and that JuMP used a MOI.Bridges.Objective.Func
to convert it into a MOI.ScalarAffineFunction{Float64} objective function.

We can leave JuMP to do the reformulation, or we can rewrite our model to have an objective function that SCS
natively supports:

```
@objective(model, Max, 1.0 * t + 0.0);
```

Re-printing the active bridges:

```
print_active_bridges(model)
```

```
* Supported objective: MOI.ScalarAffineFunction{Float64}
* Unsupported constraint: MOI.ScalarAffineFunction{Float64}-in-MOI.LessThan{Float64}
|   bridged by:
```

```
|    MOIB.Constraint.LessToGreaterBridge{Float64, MOI.ScalarAffineFunction{Float64},
↪   MOI.ScalarAffineFunction{Float64}}
|   may introduce:
|    * Unsupported constraint: MOI.ScalarAffineFunction{Float64}-in-MOI.GreaterThan{Float64}
|    |   bridged by:
|    |    MOIB.Constraint.VectorizeBridge{Float64, MOI.VectorAffineFunction{Float64},
↪   MOI.Nonnegatives, MOI.ScalarAffineFunction{Float64}}
|    |   may introduce:
|    |    * Supported constraint: MOI.VectorAffineFunction{Float64}-in-MOI.Nonnegatives
* Unsupported constraint: MOI.VectorAffineFunction{Float64}-in-MOI.PositiveSemidefiniteConeSquare
|  bridged by:
|   MOIB.Constraint.SquareBridge{Float64, MOI.VectorAffineFunction{Float64},
↪   MOI.ScalarAffineFunction{Float64}, MOI.PositiveSemidefiniteConeTriangle,
↪   MOI.PositiveSemidefiniteConeSquare}
|  may introduce:
|   * Unsupported constraint:
↪   MOI.VectorAffineFunction{Float64}-in-MOI.PositiveSemidefiniteConeTriangle
|   |   bridged by:
|   |    SCS.ScaledPSDConeBridge{Float64, MOI.VectorAffineFunction{Float64}}
|   |   may introduce:
|   |    * Supported constraint: MOI.VectorAffineFunction{Float64}-in-SCS.ScaledPSDCone
|   * Unsupported constraint: MOI.ScalarAffineFunction{Float64}-in-MOI.EqualTo{Float64}
|   |   bridged by:
|   |    MOIB.Constraint.VectorizeBridge{Float64, MOI.VectorAffineFunction{Float64}, MOI.Zeros,
↪   MOI.ScalarAffineFunction{Float64}}
|   |   may introduce:
|   |    * Supported constraint: MOI.VectorAffineFunction{Float64}-in-MOI.Zeros
* Unsupported constraint: MOI.VectorOfVariables-in-MOI.PositiveSemidefiniteConeTriangle
|  bridged by:
|   SCS.ScaledPSDConeBridge{Float64, MOI.VectorOfVariables}
|  may introduce:
|   * Supported constraint: MOI.VectorAffineFunction{Float64}-in-SCS.ScaledPSDCone
* Unsupported constraint: MOI.VectorOfVariables-in-MOI.RootDetConeSquare
|  bridged by:
|   MOIB.Constraint.SquareBridge{Float64, MOI.VectorOfVariables,
↪   MOI.ScalarAffineFunction{Float64}, MOI.RootDetConeTriangle, MOI.RootDetConeSquare}
|  may introduce:
|   * Unsupported constraint: MOI.VectorOfVariables-in-MOI.RootDetConeTriangle
|   |   bridged by:
|   |    MOIB.Constraint.VectorFunctionizeBridge{Float64, MOI.RootDetConeTriangle}
|   |   may introduce:
|   |    * Unsupported constraint: MOI.VectorAffineFunction{Float64}-in-MOI.RootDetConeTriangle
|   |   |   bridged by:
|   |   |    MOIB.Constraint.RootDetBridge{Float64, MOI.VectorAffineFunction{Float64},
↪   MOI.VectorAffineFunction{Float64}, MOI.VectorAffineFunction{Float64}}
|   |   |   may introduce:
|   |   |    * Unsupported constraint:
↪   MOI.VectorAffineFunction{Float64}-in-MOI.PositiveSemidefiniteConeTriangle
|   |   |   |   bridged by:
|   |   |   |    SCS.ScaledPSDConeBridge{Float64, MOI.VectorAffineFunction{Float64}}
|   |   |   |   may introduce:
|   |   |   |    * Supported constraint: MOI.VectorAffineFunction{Float64}-in-SCS.ScaledPSDCone
|   |   |    * Unsupported constraint: MOI.VectorAffineFunction{Float64}-in-MOI.GeometricMeanCone
|   |   |   |   bridged by:
|   |   |   |    MOIB.Constraint.GeoMeanToPowerBridge{Float64, MOI.VectorAffineFunction{Float64}}
```

```
|   |   |   |   may introduce:
|   |   |   |     * Supported constraint:
↪   MOI.VectorAffineFunction{Float64}-in-MOI.PowerCone{Float64}
|   |   |   |     * Unsupported variable: MOI.Nonnegatives
|   |   |   |     |  adding as constraint:
|   |   |   |     |    * Supported variable: MOI.Reals
|   |   |   |     |    * Unsupported constraint: MOI.VectorOfVariables-in-MOI.Nonnegatives
|   |   |   |     |    |  bridged by:
|   |   |   |     |    |    MOIB.Constraint.VectorFunctionizeBridge{Float64, MOI.Nonnegatives}
|   |   |   |     |    |  may introduce:
|   |   |   |     |    |    * Supported constraint:
↪   MOI.VectorAffineFunction{Float64}-in-MOI.Nonnegatives
|   |   |     * Supported variable: MOI.Reals
|   * Unsupported constraint: MOI.ScalarAffineFunction{Float64}-in-MOI.EqualTo{Float64}
|   |   bridged by:
|   |     MOIB.Constraint.VectorizeBridge{Float64, MOI.VectorAffineFunction{Float64}, MOI.Zeros,
↪   MOI.ScalarAffineFunction{Float64}}
|   |   may introduce:
|   |     * Supported constraint: MOI.VectorAffineFunction{Float64}-in-MOI.Zeros
```

we get * Supported objective: MOI.ScalarAffineFunction{Float64}.

We can manually implement some other reformulations to change our model to something that SCS more closely supports by:

- Replacing the MOI.VectorOfVariables in MOI.PositiveSemidefiniteConeTriangle constraint @variable(model, Z[1:n, 1:n], PSD) with the MOI.VectorAffineFunction in MOI.PositiveSemidefiniteConeTriangle @constraint(model, Z >= 0, PSDCone()).

- Replacing the MOI.VectorOfVariables in MOI.PositiveSemidefiniteConeSquare constraint [s z'; z Z] >= 0, PSDCone() with the MOI.VectorAffineFunction in MOI.PositiveSemidefiniteConeTriangle @constraint(model, LinearAlgebra.Symmetric([s z'; z Z]) >= 0, PSDCone()).

- Replacing the MOI.ScalarAffineFunction in MOI.GreaterThan constraints with the vectorized equivalent of MOI.VectorAffineFunction in MOI.Nonnegatives

- Replacing the MOI.VectorOfVariables in MOI.RootDetConeSquare constraint with MOI.VectorAffineFunction in MOI.RootDetConeTriangle.

Note that we still need to bridge MOI.PositiveSemidefiniteConeTriangle constraints because SCS uses an internal SCS.ScaledPSDCone set instead.

```
model = Model(SCS.Optimizer)
set_attribute(model, "eps_rel", 1e-6)
set_silent(model)
@variable(model, z[1:n])
@variable(model, s)
@variable(model, t)
# The former @variable(model, Z[1:n, 1:n], PSD)
@variable(model, Z[1:n, 1:n], Symmetric)
@constraint(model, Z >= 0, PSDCone())
# The former [s z'; z Z] >= 0, PSDCone()
@constraint(model, LinearAlgebra.Symmetric([s z'; z Z]) >= 0, PSDCone())
```

```
# The former constraint S[i, :]' * Z * S[i, :] - 2 * S[i, :]' * z + s <= 1
f = [1 - S[i, :]' * Z * S[i, :] + 2 * S[i, :]' * z - s for i in 1:m]
@constraint(model, f in MOI.Nonnegatives(m))
# The former constraint [t; vec(Z)] in MOI.RootDetConeSquare(n)
Z_upper = [Z[i, j] for j in 1:n for i in 1:j]
@constraint(model, 1 * vcat(t, Z_upper) .+ 0 in MOI.RootDetConeTriangle(n))
# The former @objective(model, Max, t)
@objective(model, Max, 1 * t + 0)
optimize!(model)
solve_time_1 = solve_time(model)
```

```
0.961091083
```

This formulation gives the much smaller graph:

```
print_active_bridges(model)
```

```
 * Supported objective: MOI.ScalarAffineFunction{Float64}
 * Supported constraint: MOI.VectorAffineFunction{Float64}-in-MOI.Nonnegatives
 * Unsupported constraint:
↪  MOI.VectorAffineFunction{Float64}-in-MOI.PositiveSemidefiniteConeTriangle
 |  bridged by:
 |   SCS.ScaledPSDConeBridge{Float64, MOI.VectorAffineFunction{Float64}}
 |  may introduce:
 |   * Supported constraint: MOI.VectorAffineFunction{Float64}-in-SCS.ScaledPSDCone
 * Unsupported constraint: MOI.VectorAffineFunction{Float64}-in-MOI.RootDetConeTriangle
 |  bridged by:
 |   MOIB.Constraint.RootDetBridge{Float64, MOI.VectorAffineFunction{Float64},
↪  MOI.VectorAffineFunction{Float64}, MOI.VectorAffineFunction{Float64}}
 |  may introduce:
 |   * Unsupported constraint:
↪  MOI.VectorAffineFunction{Float64}-in-MOI.PositiveSemidefiniteConeTriangle
 |  |   bridged by:
 |  |    SCS.ScaledPSDConeBridge{Float64, MOI.VectorAffineFunction{Float64}}
 |  |   may introduce:
 |  |    * Supported constraint: MOI.VectorAffineFunction{Float64}-in-SCS.ScaledPSDCone
 |   * Unsupported constraint: MOI.VectorAffineFunction{Float64}-in-MOI.GeometricMeanCone
 |  |   bridged by:
 |  |    MOIB.Constraint.GeoMeanToPowerBridge{Float64, MOI.VectorAffineFunction{Float64}}
 |  |   may introduce:
 |  |    * Supported constraint: MOI.VectorAffineFunction{Float64}-in-MOI.PowerCone{Float64}
 |  |    * Unsupported variable: MOI.Nonnegatives
 |  |  |   adding as constraint:
 |  |  |    * Supported variable: MOI.Reals
 |  |  |    * Unsupported constraint: MOI.VectorOfVariables-in-MOI.Nonnegatives
 |  |  |  |   bridged by:
 |  |  |  |    MOIB.Constraint.VectorFunctionizeBridge{Float64, MOI.Nonnegatives}
 |  |  |  |   may introduce:
 |  |  |  |    * Supported constraint: MOI.VectorAffineFunction{Float64}-in-MOI.Nonnegatives
 |   * Supported variable: MOI.Reals
```

The last bullet shows how JuMP reformulated the `MOI.RootDetConeTriangle` constraint by adding a mix of `MOI.PositiveSemidefiniteConeTriangle` and `MOI.GeometricMeanCone` constraints.

Because SCS doesn't natively support the `MOI.GeometricMeanCone`, these constraints were further bridged using a `MOI.Bridges.Constraint.GeoMeanToPowerBridge` to a series of `MOI.PowerCone` constraints.

However, there are many other ways that a `MOI.GeometricMeanCone` can be reformulated into something that SCS supports. Let's see what happens if we use `remove_bridge` to remove the `MOI.Bridges.Constraint.GeoMeanToPowerBridge`.

```
remove_bridge(model, MOI.Bridges.Constraint.GeoMeanToPowerBridge)
optimize!(model)
```

This time, the solve took:

```
solve_time_2 = solve_time(model)
```

```
0.548464119
```

where previously it took

```
solve_time_1
```

```
0.961091083
```

Why was the solve time different?

```
print_active_bridges(model)
```

```
 * Supported objective: MOI.ScalarAffineFunction{Float64}
 * Supported constraint: MOI.VectorAffineFunction{Float64}-in-MOI.Nonnegatives
 * Unsupported constraint:
↪   MOI.VectorAffineFunction{Float64}-in-MOI.PositiveSemidefiniteConeTriangle
 |   bridged by:
 |    SCS.ScaledPSDConeBridge{Float64, MOI.VectorAffineFunction{Float64}}
 |   may introduce:
 |    * Supported constraint: MOI.VectorAffineFunction{Float64}-in-SCS.ScaledPSDCone
 * Unsupported constraint: MOI.VectorAffineFunction{Float64}-in-MOI.RootDetConeTriangle
 |   bridged by:
 |    MOIB.Constraint.RootDetBridge{Float64, MOI.VectorAffineFunction{Float64},
↪   MOI.VectorAffineFunction{Float64}, MOI.VectorAffineFunction{Float64}}
 |   may introduce:
 |    * Unsupported constraint:
↪   MOI.VectorAffineFunction{Float64}-in-MOI.PositiveSemidefiniteConeTriangle
 |    |   bridged by:
 |    |    SCS.ScaledPSDConeBridge{Float64, MOI.VectorAffineFunction{Float64}}
 |    |   may introduce:
```

```
| |   * Supported constraint: MOI.VectorAffineFunction{Float64}-in-SCS.ScaledPSDCone
| * Unsupported constraint: MOI.VectorAffineFunction{Float64}-in-MOI.GeometricMeanCone
| |   bridged by:
| |    MOIB.Constraint.GeoMeanBridge{Float64, MOI.ScalarAffineFunction{Float64},
↪ MOI.VectorAffineFunction{Float64}, MOI.VectorAffineFunction{Float64}}
| |   may introduce:
| |    * Unsupported constraint: MOI.ScalarAffineFunction{Float64}-in-MOI.LessThan{Float64}
| |    |   bridged by:
| |    |    MOIB.Constraint.VectorizeBridge{Float64, MOI.VectorAffineFunction{Float64},
↪ MOI.Nonpositives, MOI.ScalarAffineFunction{Float64}}
| |    |   may introduce:
| |    |    * Unsupported constraint: MOI.VectorAffineFunction{Float64}-in-MOI.Nonpositives
| |    |    |   bridged by:
| |    |    |    MOIB.Constraint.NonposToNonnegBridge{Float64, MOI.VectorAffineFunction{Float64},
↪ MOI.VectorAffineFunction{Float64}}
| |    |    |   may introduce:
| |    |    |    * Supported constraint: MOI.VectorAffineFunction{Float64}-in-MOI.Nonnegatives
| |    * Unsupported constraint: MOI.VectorAffineFunction{Float64}-in-MOI.RotatedSecondOrderCone
| |    |   bridged by:
| |    |    MOIB.Constraint.RSOCtoSOCBridge{Float64, MOI.VectorAffineFunction{Float64},
↪ MOI.VectorAffineFunction{Float64}}
| |    |   may introduce:
| |    |    * Supported constraint: MOI.VectorAffineFunction{Float64}-in-MOI.SecondOrderCone
| |    * Supported constraint: MOI.VectorAffineFunction{Float64}-in-MOI.Nonnegatives
| |    * Supported variable: MOI.Reals
| * Supported variable: MOI.Reals
```

This time, JuMP used a MOI.Bridges.Constraint.GeoMeanBridge to reformulate the constraint into a set of MOI.RotatedSecondOrderCone constraints, which were further reformulated into a set of supported MOI.SecondOrderCone constraints.

Since the two models are equivalent, we can conclude that for this particular model, the MOI.SecondOrderCone formulation is more efficient.

In general though, the performance of a particular reformulation is problem- and solver-specific. Therefore, JuMP chooses to minimize the number of bridges in the default reformulation, leaving you to explore alternative formulations using the tools and techniques shown in this tutorial.

7.10 Quantum state discrimination

This tutorial solves the problem of quantum state discrimination.

The purpose of this tutorial to demonstrate how to solve problems involving complex-valued decision variables and the HermitianPSDCone. See Complex number support for more details.

Required packages

This tutorial makes use of the following packages:

```
using JuMP
import LinearAlgebra
import SCS
```

Formulation

A d-dimensional quantum state, ρ, can be defined by a complex-valued Hermitian matrix with a trace of 1. Assume we have N d-dimensional quantum states, $\{\rho_i\}_{i=1}^{N}$, each of which is equally likely.

The goal of the quantum state discrimination problem is to choose a positive operator-valued measure (POVM), $\{E_i\}_{i=1}^{N}$, such that if we observe E_i then the most probable state that we are in is ρ_i.

Each POVM element, E_i, is a complex-valued Hermitian matrix, and there is a requirement that $\sum\limits_{i=1}^{N} E_i = \mathbf{I}$.

To choose a POVM, we want to maximize the probability that we guess the quantum state correctly. This can be formulated as the following optimization problem:

$$\max_{E} \quad \frac{1}{N} \sum_{i=1}^{N} \operatorname{tr}(\rho_i E_i)$$

$$\text{s.t.} \quad \sum_{i=1}^{N} E_i = \mathbf{I}$$

$$E_i \succeq 0 \ \forall i = 1, \dots, N.$$

Data

To setup our problem, we need N d-dimensional quantum states. To keep the problem simple, we use N = 2 and d = 2.

```
N, d = 2, 2
```

```
(2, 2)
```

We then generated N random d-dimensional quantum states:

```
function random_state(d)
    x = randn(ComplexF64, (d, d))
    y = x * x'
    return LinearAlgebra.Hermitian(y / LinearAlgebra.tr(y))
end

ρ = [random_state(d) for i in 1:N]
```

```
2-element Vector{LinearAlgebra.Hermitian{ComplexF64, Matrix{ComplexF64}}}:
 [0.9317077745932991 + 0.0im 0.17785044289093424 - 0.11927897015052207im; 0.17785044289093424 +
  ↪   0.11927897015052207im 0.06829222540670073 + 0.0im]
 [0.22453156773914584 + 0.0im 0.061840181904862485 - 0.0523571348318324884im; 0.061840181904862485 +
  ↪   0.0523571348318324884im 0.7754684322608543 + 0.0im]
```

JuMP formulation

To model the problem in JuMP, we need a solver that supports positive semidefinite matrices:

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

Then, we construct our set of E variables:

```
E = [@variable(model, [1:d, 1:d] in HermitianPSDCone()) for i in 1:N]
```

```
2-element Vector{LinearAlgebra.Hermitian{GenericAffExpr{ComplexF64, VariableRef},
↪  Matrix{GenericAffExpr{ComplexF64, VariableRef}}}}:
 [_[1] _[2] + _[4] im; _[2] - _[4] im _[3]]
 [_[5] _[6] + _[8] im; _[6] - _[8] im _[7]]
```

Here we have created a vector of matrices. This is different to other modeling languages such as YALMIP, which allow you to create a multi-dimensional array in which 2-dimensional slices of the array are Hermitian matrices.

We also need to enforce the constraint that $\sum_{i=1}^{N} E_i = \mathbf{I}$:

```
@constraint(model, sum(E) == LinearAlgebra.I)
```

$$\begin{bmatrix} {}_{-1} + {}_{-5} - 1 & {}_{-2} + {}_{-6} + {}_{-4}im + {}_{-8}im \\ {}_{-2} + {}_{-6} - {}_{-4}im - {}_{-8}im & {}_{-3} + {}_{-7} - 1 \end{bmatrix} \in \mathsf{Zeros()}$$

This constraint is a complex-valued equality constraint. In the solver, it will be decomposed onto two types of equality constraints: one to enforce equality of the real components, and one to enforce equality of the imaginary components.

Our objective is to maximize the expected probability of guessing correctly:

```
@objective(
    model,
    Max,
    sum(real(LinearAlgebra.tr(ρ[i] * E[i])) for i in 1:N) / N,
)
```

$0.46585388729664956_{\_1} + 0.17785044289093424_{\_2} - 0.11927897015052207_{\_4} + 0.03414611270335036_{\_3} + 0.1122657838695$

Now we optimize:

```
optimize!(model)
solution_summary(model)
```

```
* Solver : SCS

* Status
  Result count       : 1
  Termination status : OPTIMAL
  Message from the solver:
  "solved"

* Candidate solution (result #1)
  Primal status      : FEASIBLE_POINT
  Dual status        : FEASIBLE_POINT
  Objective value    : 8.59872e-01
  Dual objective value : 8.59872e-01

* Work counters
  Solve time (sec)   : 4.92310e-04
```

The probability of guessing correctly is:

```
objective_value(model)
```

```
0.8598717791091357
```

When `N = 2`, there is a known analytical solution of:

```
0.5 + 0.25 * sum(LinearAlgebra.svdvals(ρ[1] - ρ[2]))
```

```
0.859873276389449
```

proving that we found the optimal solution.

Finally, the optimal POVM is:

```
solution = [value.(e) for e in E]
```

```
2-element Vector{Matrix{ComplexF64}}:
 [0.9912654760970278 - 0.0im 0.08059071544261419 - 0.046489668600235685im; 0.08059071544261419 +
 ↪  0.046489668600235685im 0.008734518010155559 - 0.0im]
 [0.00873451833376001 - 0.0im -0.08059071539285066 + 0.046489668564575905im; -0.08059071539285066 -
 ↪  0.046489668564575905im 0.9912654763763605 - 0.0im]
```

Alternative formulation

The formulation above includes N Hermitian matrices and a set of linear equality constraints. We can simplify

the problem by replacing E_N with $E_N = I - \sum_{i=1}^{N-1} E_i$. This results in:

```
model = Model(SCS.Optimizer)
set_silent(model)
E = [@variable(model, [1:d, 1:d] in HermitianPSDCone()) for i in 1:N-1]
E_N = LinearAlgebra.Hermitian(LinearAlgebra.I - sum(E))
@constraint(model, E_N in HermitianPSDCone())
push!(E, E_N)
```

```
2-element Vector{LinearAlgebra.Hermitian{GenericAffExpr{ComplexF64, VariableRef},
↪  Matrix{GenericAffExpr{ComplexF64, VariableRef}}}}:
 [_[1] _[2] + _[4] im; _[2] - _[4] im _[3]]
 [-_[1] + 1 -_[2] - _[4] im; -_[2] + _[4] im -_[3] + 1]
```

The objective can also be simplified, by observing that it is equivalent to:

```
@objective(model, Max, real(LinearAlgebra.dot(ρ, E)) / N)
```

$$0.35358810342707664\_1 + 0.11601026098607176\_2 - 0.06692183531868959\_4 - 0.35358810342707675\_3 + 0.5$$

Then we can check that we get the same solution:

```
optimize!(model)
solution_summary(model)
```

```
* Solver : SCS

* Status
  Result count       : 1
  Termination status : OPTIMAL
  Message from the solver:
  "solved"

* Candidate solution (result #1)
  Primal status      : FEASIBLE_POINT
  Dual status        : FEASIBLE_POINT
  Objective value    : 8.59870e-01
  Dual objective value : 8.59873e-01

* Work counters
  Solve time (sec)   : 4.47109e-04
```

```
objective_value(model)
```

```
0.8598700867637956
```

Chapter 8

Algorithms

8.1 Benders decomposition

This tutorial was originally contributed by Shuvomoy Das Gupta.

This tutorial describes how to implement Benders decomposition in JuMP. It uses the following packages:

```julia
using JuMP
import GLPK
import Printf
```

Theory

Benders decomposition is a useful algorithm for solving convex optimization problems with a large number of variables. It works best when a larger problem can be decomposed into two (or more) smaller problems that are individually much easier to solve. This tutorial demonstrates Benders decomposition on the following mixed-integer linear program:

$$
\begin{aligned}
\min\ & c_1^\top x + c_2^\top y \\
\text{subject to}\ & A_1 x + A_2 y \leq b \\
& x \geq 0 \\
& y \geq 0 \\
& x \in \mathbb{Z}^n
\end{aligned}
$$

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.

Any mixed integer programming problem can be written in the form above.

If there are relatively few integer variables, and many more continuous variables, then it may be beneficial to decompose the problem into a small problem containing only integer variables and a linear program containing only continuous variables. Hopefully, the linear program will be much easier to solve in isolation than in the full mixed-integer linear program.

For example, if we knew a feasible solution for x, we could obtain a solution for y by solving:

$$
\begin{aligned}
V_2(x) = \min\ & c_2^\top y \\
\text{subject to}\quad & A_2 y \leq b - A_1 x \quad [\pi] \\
& y \geq 0,
\end{aligned}
$$

where π is the dual variable associated with the constraints. Because this is a linear program, it is easy to solve.

Replacing the $c_2^\top y$ component of the objective in our original problem with V_2 yields:

$$\min c_1^\top x + V_2(x)$$
$$\text{subject to } x \geq 0$$
$$x \in \mathbb{Z}^n$$

This problem looks a lot simpler to solve, but we need to do something else with V_2 first.

Because x is a constant that appears on the right-hand side of the constraints, V_2 is a convex function with respect to x, and the dual variable π can be multiplied by $-A_1$ to obtain a subgradient of $V_2(x)$ with respect to x. Therefore, if we have a candidate solution x_k, then we can solve $V_2(x_k)$ and obtain a feasible dual vector π_k. Using these values, we can construct a first-order Taylor-series approximation of V_2 about the point x_k:

$$V_2(x) \geq V_2(x_k) + -\pi_k^\top A_1 (x - x_k).$$

By convexity, we know that this inequality holds for all x, and we call these inequalities cuts.

Benders decomposition is an iterative technique that replaces $V_2(x)$ with a new decision variable θ, and approximates it from below using cuts:

$$
\begin{aligned}
V_1^K = \min \quad & c_1^\top x + \theta \\
\text{subject to} \quad & x \geq 0 \\
& x \in \mathbb{Z}^n \\
& \theta \geq M \\
& \theta \geq V_2(x_k) + \pi_k^\top (x - x_k) \quad \forall k = 1, \ldots, K.
\end{aligned}
$$

This integer program is called the first-stage subproblem.

To generate cuts, we solve V_1^K to obtain a candidate first-stage solution x_k, then we use that solution to solve $V_2(x_k)$. Then, using the optimal objective value and dual solution from V_2, we add a new cut to form V_1^{K+1} and repeat.

Bounds

Due to convexity, we know that $V_2(x) \geq \theta$ for all x. Therefore, the optimal objective value of V_1^K provides a valid lower bound on the objective value of the full problem. In addition, if we take a feasible solution for x from the first-stage problem, then $c_1^\top x + V_2(x)$ is a valid upper bound on the objective value of the full problem.

Benders decomposition uses the lower and upper bounds to determine when it has found the global optimal solution.

Input data

As an example for this tutorial, we use the input data is from page 139 of Garfinkel, R. & Nemhauser, G. L. Integer programming. (Wiley, 1972).

```
c_1 = [1, 4]
c_2 = [2, 3]
dim_x = length(c_1)
dim_y = length(c_2)
b = [-2; -3]
A_1 = [1 -3; -1 -3]
A_2 = [1 -2; -1 -1]
M = -1000;
```

Iterative method

> **Warning**
>
> This is a basic implementation for pedagogical purposes. We haven't discussed Benders feasibility cuts, or any of the computational tricks that are required to build a performant implementation for large-scale problems. See In-place iterative method for one improvement that helps computation time.

We start by formulating the first-stage subproblem:

```
model = Model(GLPK.Optimizer)
@variable(model, x[1:dim_x] >= 0, Int)
@variable(model, θ >= M)
@objective(model, Min, c_1' * x + θ)
print(model)
```

```
Min x[1] + 4 x[2] + θ
Subject to
 x[1] ≥ 0
 x[2] ≥ 0
 θ ≥ -1000
 x[1] integer
 x[2] integer
```

For the next step, we need a function that takes a first-stage candidate solution x and returns the optimal solution from the second-stage subproblem:

```
function solve_subproblem(x)
    model = Model(GLPK.Optimizer)
    @variable(model, y[1:dim_y] >= 0)
    con = @constraint(model, A_2 * y .<= b - A_1 * x)
    @objective(model, Min, c_2' * y)
    optimize!(model)
    @assert termination_status(model) == OPTIMAL
    return (obj = objective_value(model), y = value.(y), π = dual.(con))
end
```

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

Note that `solve_subproblem` returns a NamedTuple of the objective value, the optimal primal solution for y, and the optimal dual solution for π.

We're almost ready for our optimization loop, but first, here's a helpful function for logging:

```julia
function print_iteration(k, args...)
    f(x) = Printf.@sprintf("%12.4e", x)
    println(lpad(k, 9), " ", join(f.(args), " "))
    return
end
```

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

We also need to put a limit on the number of iterations before termination:

```
MAXIMUM_ITERATIONS = 100
```

```
100
```

And a way to check if the lower and upper bounds are close-enough to terminate:

```
ABSOLUTE_OPTIMALITY_GAP = 1e-6
```

```
1.0e-6
```

Now we're ready to iterate Benders decomposition:

```julia
println("Iteration  Lower Bound  Upper Bound          Gap")
for k in 1:MAXIMUM_ITERATIONS
    optimize!(model)
    lower_bound = objective_value(model)
    x_k = value.(x)
    ret = solve_subproblem(x_k)
    upper_bound = c_1' * x_k + ret.obj
    gap = (upper_bound - lower_bound) / upper_bound
    print_iteration(k, lower_bound, upper_bound, gap)
    if gap < ABSOLUTE_OPTIMALITY_GAP
        println("Terminating with the optimal solution")
        break
    end
    cut = @constraint(model, θ >= ret.obj + -ret.π' * A_1 * (x .- x_k))
    @info "Adding the cut $(cut)"
end
```

```
Iteration  Lower Bound  Upper Bound       Gap
        1  -1.0000e+03   7.6667e+00   1.3143e+02
[ Info: Adding the cut 2 x[1] + 8 x[2] + θ ≥ 7.666666666666666
        2  -4.9600e+02   1.2630e+03   1.3927e+00
[ Info: Adding the cut -1.5 x[1] + 4.5 x[2] + θ ≥ 3
        3  -1.0800e+02   8.8800e+02   1.1216e+00
[ Info: Adding the cut θ ≥ 0
        4   4.0000e+00   4.0000e+00   0.0000e+00
Terminating with the optimal solution
```

Finally, we can obtain the optimal solution

```
optimize!(model)
x_optimal = value.(x)
```

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

```
optimal_ret = solve_subproblem(x_optimal)
y_optimal = optimal_ret.y
```

```
2-element Vector{Float64}:
 0.0
 0.0
```

Callback method

The Iterative method section implemented Benders decomposition using a loop. In each iteration, we re-solved the first-stage subproblem to generate a candidate solution. However, modern MILP solvers such as CPLEX, Gurobi, and GLPK provide lazy constraint callbacks which allow us to add new cuts while the solver is running. This can be more efficient than an iterative method because we can avoid repeating work such as solving the root node of the first-stage MILP at each iteration.

> **Tip**
>
> For more information on callbacks, read the page Solver-independent callbacks.

As before, we construct the same first-stage subproblem:

```
lazy_model = Model(GLPK.Optimizer)
@variable(lazy_model, x[1:dim_x] >= 0, Int)
@variable(lazy_model, θ >= M)
@objective(lazy_model, Min, θ)
print(lazy_model)
```

```
Min θ
Subject to
 x[1] ≥ 0
 x[2] ≥ 0
 θ ≥ -1000
 x[1] integer
 x[2] integer
```

What differs is that we write a callback function instead of a loop:

```
k = 0

"""
    my_callback(cb_data)

A callback that implements Benders decomposition. Note how similar it is to the
inner loop of the iterative method.
"""
function my_callback(cb_data)
    global k += 1
    x_k = callback_value.(cb_data, x)
    θ_k = callback_value(cb_data, θ)
    lower_bound = c_1' * x_k + θ_k
    ret = solve_subproblem(x_k)
    upper_bound = c_1' * x_k + c_2' * ret.y
    gap = (upper_bound - lower_bound) / upper_bound
    print_iteration(k, lower_bound, upper_bound, gap)
    if gap < ABSOLUTE_OPTIMALITY_GAP
        println("Terminating with the optimal solution")
        return
    end
    cut = @build_constraint(θ >= ret.obj + -ret.π' * A_1 * (x .- x_k))
    MOI.submit(model, MOI.LazyConstraint(cb_data), cut)
    return
end

set_attribute(lazy_model, MOI.LazyConstraintCallback(), my_callback)
```

Now when we optimize!, our callback is run:

```
optimize!(lazy_model)
```

```
        1  -1.0000e+03   7.6667e+00   1.3143e+02
        2  -4.9617e+02   5.0383e+02   1.9848e+00
        3   3.8333e+00   4.0833e+00   6.1224e-02
        4   4.0000e+00   4.0000e+00   0.0000e+00
Terminating with the optimal solution
```

Note how this problem also takes 4 iterations to converge, but the sequence of bounds is different compared to the iterative method.

Finally, we can obtain the optimal solution:

```
x_optimal = value.(x)
```

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

```
optimal_ret = solve_subproblem(x_optimal)
y_optimal = optimal_ret.y
```

```
2-element Vector{Float64}:
 0.0
 0.0
```

In-place iterative method

Our implementation of the iterative method has a problem: every time we need to solve the subproblem, we must rebuild it from scratch. This is expensive, and it can be the bottleneck in the solution process. We can improve our implementation by using re-using the subproblem between solves.

First, we create our first-stage problem as usual:

```
model = Model(GLPK.Optimizer)
@variable(model, x[1:dim_x] >= 0, Int)
@variable(model, θ >= M)
@objective(model, Min, c_1' * x + θ)
print(model)
```

```
Min x[1] + 4 x[2] + θ
Subject to
 x[1] ≥ 0
 x[2] ≥ 0
 θ ≥ -1000
 x[1] integer
 x[2] integer
```

Then, instead of building the subproblem in a function, we build it once here:

```
subproblem = Model(GLPK.Optimizer)
@variable(subproblem, x_copy[1:dim_x])
@variable(subproblem, y[1:dim_y] >= 0)
@constraint(subproblem, A_1 * x_copy + A_2 * y .<= b)
@objective(subproblem, Min, c_2' * y)
print(subproblem)
```

```
Min 2 y[1] + 3 y[2]
Subject to
 x_copy[1] - 3 x_copy[2] + y[1] - 2 y[2] ≤ -2
 -x_copy[1] - 3 x_copy[2] - y[1] - y[2] ≤ -3
 y[1] ≥ 0
 y[2] ≥ 0
```

This formulation is slightly different. We have included a copy of the x variables, x_copy, and used x_copy in the left-hand side of the constraints.

Our function to solve the subproblem is also slightly different. First, we need to fix the value of the x_copy variables to the value of x from the first-stage problem, and second, we compute the dual using the reduced_cost of x_copy, not the dual of the linear constraints:

```julia
function solve_subproblem(model, x)
    fix.(model[:x_copy], x)
    optimize!(model)
    @assert termination_status(model) == OPTIMAL
    return (
        obj = objective_value(model),
        y = value.(model[:y]),
        π = reduced_cost.(model[:x_copy]),
    )
end
```

```
solve_subproblem (generic function with 2 methods)
```

Now we're ready to iterate our in-place Benders decomposition, but this time the cut computation is slightly different. Because we used reduced_cost on the x_copy variables, the value of π is a valid subgradient on the objective of subproblem with respect to x. Therefore, we don't need to multiply the duals by -A_1, and so our cut uses ret.π' instead of -ret.π' * A_1:

```julia
println("Iteration  Lower Bound  Upper Bound          Gap")
for k in 1:MAXIMUM_ITERATIONS
    optimize!(model)
    lower_bound = objective_value(model)
    x_k = value.(x)
    ret = solve_subproblem(subproblem, x_k)
    upper_bound = c_1' * x_k + ret.obj
    gap = (upper_bound - lower_bound) / upper_bound
    print_iteration(k, lower_bound, upper_bound, gap)
    if gap < ABSOLUTE_OPTIMALITY_GAP
        println("Terminating with the optimal solution")
        break
    end
    cut = @constraint(model, θ >= ret.obj + ret.π' * (x .- x_k))
    @info "Adding the cut $(cut)"
end
```

```
Iteration  Lower Bound  Upper Bound        Gap
        1  -1.0000e+03   7.6667e+00   1.3143e+02
[ Info: Adding the cut 2 x[1] + 8 x[2] + θ ≥ 7.666666666666666
        2  -4.9600e+02   1.2630e+03   1.3927e+00
[ Info: Adding the cut -1.5 x[1] + 4.5 x[2] + θ ≥ 3
        3  -1.0800e+02   8.8800e+02   1.1216e+00
[ Info: Adding the cut θ ≥ 0
        4   4.0000e+00   4.0000e+00   0.0000e+00
Terminating with the optimal solution
```

Finally, we can obtain the optimal solution:

```
optimize!(model)
x_optimal = value.(x)
```

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

```
optimal_ret = solve_subproblem(subproblem, x_optimal)
y_optimal = optimal_ret.y
```

```
2-element Vector{Float64}:
 0.0
 0.0
```

For larger problems, the benefit of re-using the same subproblem and not needing to multiply the duals by A_1 in the cut coefficient usually outweighs the cost of needing a copy of the x variables in the subproblem.

As a secondary benefit, because we no longer need an explicit representation of A_1 in the cut, we can build the `model` and `subproblem` formulations using arbitrary JuMP syntax; they do not need to be in matrix form.

8.2   Column generation

This tutorial describes how to implement the Cutting stock problem in JuMP using column generation. It uses the following packages:

```
using JuMP
import GLPK
import SparseArrays
```

Mathematical formulation

The cutting stock problem is about cutting large rolls of paper into smaller pieces. There is a demand different sizes of pieces to meet, and all large rolls have the same width. The goal is to meet the demand while maximizing the total profit.

We denote the set of possible sized pieces that a roll can be cut into by $i \in 1, \ldots, I$. Each piece i has a width, w_i, and a demand, d_i. The width of the large roll is W.

Here's the data that we are going to use in this tutorial:

```julia
struct Piece
    w::Float64
    d::Int
end

struct Data
    pieces::Vector{Piece}
    W::Float64
end

function Base.show(io::IO, d::Data)
    println(io, "Data for the cutting stock problem:")
    println(io, "  W = $(d.W)")
    println(io, "with pieces:")
    println(io, "   i   w_i d_i")
    println(io, "  ------------")
    for (i, p) in enumerate(d.pieces)
        println(io, lpad(i, 4), " ", lpad(p.w, 5), " ", lpad(p.d, 3))
    end
    return
end

function get_data()
    data = [
        75.0 38
        75.0 44
        75.0 30
        75.0 41
        75.0 36
        53.8 33
        53.0 36
        51.0 41
        50.2 35
        32.2 37
        30.8 44
        29.8 49
        20.1 37
        16.2 36
        14.5 42
        11.0 33
        8.6 47
        8.2 35
        6.6 49
        5.1 42
    ]
    return Data([Piece(data[i, 1], data[i, 2]) for i in axes(data, 1)], 100.0)
```

```
end

data = get_data()
```

```
Data for the cutting stock problem:
  W = 100.0
with pieces:
   i   w_i d_i
   ------------
   1  75.0  38
   2  75.0  44
   3  75.0  30
   4  75.0  41
   5  75.0  36
   6  53.8  33
   7  53.0  36
   8  51.0  41
   9  50.2  35
  10  32.2  37
  11  30.8  44
  12  29.8  49
  13  20.1  37
  14  16.2  36
  15  14.5  42
  16  11.0  33
  17   8.6  47
  18   8.2  35
  19   6.6  49
  20   5.1  42
```

To formulate the cutting stock problem as a mixed-integer linear program, we assume that there is a set of large rolls $j = 1, \ldots, J$ to use. Then, we introduce two classes of decision variables:

- $x_{ij} \geq 0$, integer, $\forall i = 1, \ldots, I, j = 1, \ldots, J$

- $y_j \in \{0, 1\} \forall j = 1, \ldots, J.$

y_j is a binary variable that indicates if we use roll j, and x_{ij} counts how many pieces of size i that we cut from roll j.

Our mixed-integer linear program is therefore:

$$\min \sum_{j=1}^{J} y_j \tag{8.1}$$

$$\text{s.t.} \sum_{i=1}^{N} w_i x_{ij} \leq W y_j \qquad \forall j = 1, \dots, J \tag{8.2}$$

$$\sum_{j=1}^{J} x_{ij} \geq d_i \qquad \forall i = 1, \dots, I \tag{8.3}$$

$$x_{ij} \geq 0 \qquad \forall i = 1, \dots, N, j = 1, \dots, J \tag{8.4}$$

$$x_{ij} \in \mathbb{Z} \qquad \forall i = 1, \dots, I, j = 1, \dots, J \tag{8.5}$$

$$y_j \in \{0, 1\} \qquad \forall j = 1, \dots, J \tag{8.6}$$

$$\tag{8.7}$$

The objective is to minimize the number of rolls that we use, and the two constraints ensure that we respect the total width of each large roll and that we satisfy demand exactly.

```
I = length(data.pieces)
J = 1000  # Some large number
model = Model(GLPK.Optimizer)
@variable(model, x[1:I, 1:J] >= 0, Int)
@variable(model, y[1:J], Bin)
@constraint(
    model,
    [j in 1:J],
    sum(data.pieces[i].w * x[i, j] for i in 1:I) <= data.W * y[j],
)
@constraint(model, [i in 1:I], sum(x[i, j] for j in 1:J) >= data.pieces[i].d)
@objective(model, Min, sum(y[j] for j in 1:J))
model
```

```
A JuMP Model
Minimization problem with:
Variables: 21000
Objective function type: AffExpr
`AffExpr`-in-`MathOptInterface.GreaterThan{Float64}`: 20 constraints
`AffExpr`-in-`MathOptInterface.LessThan{Float64}`: 1000 constraints
`VariableRef`-in-`MathOptInterface.GreaterThan{Float64}`: 20000 constraints
`VariableRef`-in-`MathOptInterface.Integer`: 20000 constraints
`VariableRef`-in-`MathOptInterface.ZeroOne`: 1000 constraints
Model mode: AUTOMATIC
CachingOptimizer state: EMPTY_OPTIMIZER
Solver name: GLPK
Names registered in the model: x, y
```

Unfortunately, we won't attempt to solve this formulation because it takes a very long time to solve. (Try it and see.)

```
# optimize!(model)
```

However, there is a formulation that solves much faster, and that is to use a column generation scheme.

Column generation theory

The key insight for column generation is to recognize that the x variables above encode cutting patterns. For example, if we look only at the roll $j = 1$, then feasible solutions are:

- $x_{1,1} = 1$, $x_{13,1} = 1$ and all the rest 0, which is 1 roll of piece #1 and 1 roll of piece #13

- $x_{1,20} = 19$ and all the rest 0, which is 19 rolls of piece #20.

Cutting patterns like $x_{1,1} = 1$ and $x_{2,1} = 1$ are infeasible because the combined length is greater than W.

Since there are a finite number of ways that we could cut a roll into a valid cutting pattern, we can create a set of all possible cutting patterns $p = 1, \ldots, P$, with data $a_{i,p}$ indicating how many pieces of size i we cut in pattern p. Then, we can formulate our mixed-integer linear program as:

$$\min \sum_{p=1}^{P} x_p \tag{8.8}$$

$$\text{s.t.} \sum_{p=1}^{P} a_{ip} x_p \geq d_i \qquad\qquad \forall i = 1, \ldots, I \tag{8.9}$$

$$x_p \geq 0 \qquad\qquad \forall p = 1, \ldots, P \tag{8.10}$$

$$x_p \in \mathbb{Z} \qquad\qquad \forall p = 1, \ldots, P \tag{8.11}$$

Unfortunately, there will be a very large number of these patterns, so it is often intractable to enumerate all columns $p = 1, \ldots, P$.

Column generation is an iterative algorithm that starts with a small set of initial patterns, and then cleverly chooses new columns to add to the main MILP so that we find the optimal solution without having to enumerate every column.

Choosing new columns

For now we assume that we have our mixed-integer linear program with a subset of the columns. If we have all of the columns that appear in an optimal solution then we are done. Otherwise, how do we choose a new column that leads to an improved solution?

Column generation chooses a new column by relaxing the integrality constraint on x and looking at the dual variable π_i associated with demand constraint i.

Using the economic interpretation of the dual variable, we can say that a one unit increase in demand for piece i will cost an extra π_i rolls. Alternatively, we can say that a one unit increase in the left-hand side (for example, due to a new cutting pattern) will save us π_i rolls. Therefore, we want a new column that maximizes the savings associated with the dual variables, while respecting the total width of the roll:

$$\max \sum_{i=1}^{I} \pi_i y_i \tag{8.12}$$

$$\text{s.t.} \sum_{i=1}^{I} w_i y_i \leq W \tag{8.13}$$

$$y_i \geq 0 \qquad\qquad \forall i = 1, \ldots, I \tag{8.14}$$

$$y_i \in \mathbb{Z} \qquad\qquad \forall i = 1, \ldots, I \tag{8.15}$$

$$\tag{8.16}$$

If this problem, called the pricing problem, has an objective value greater than 1, then we estimate than adding y as the coefficients of a new column will decrease the objective by more than the cost of an extra roll.

Here is code to solve the pricing problem:

```
function solve_pricing(data::Data, π::Vector{Float64})
    I = length(π)
    model = Model(GLPK.Optimizer)
    set_silent(model)
    @variable(model, y[1:I] >= 0, Int)
    @constraint(model, sum(data.pieces[i].w * y[i] for i in 1:I) <= data.W)
    @objective(model, Max, sum(π[i] * y[i] for i in 1:I))
    optimize!(model)
    if objective_value(model) > 1
        return round.(Int, value.(y))
    end
    return nothing
end
```

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

Choosing the initial set of patterns

For the initial set of patterns, we create a trivial cutting pattern which cuts as many pieces of size i as will fit, or the amount demanded, whichever is smaller.

```
patterns = Vector{Int}[]
for i in 1:I
    pattern = zeros(Int, I)
    pattern[i] = floor(Int, min(data.W / data.pieces[i].w, data.pieces[i].d))
    push!(patterns, pattern)
end
P = length(patterns)
```

```
20
```

We can visualize the patterns by looking at the sparse matrix of the non-zeros:

```
SparseArrays.sparse(hcat(patterns...))
```

```
20×20 SparseArrays.SparseMatrixCSC{Int64, Int64} with 20 stored entries:
```

Solving the problem

First, we create our initial linear program:

```
model = Model(GLPK.Optimizer)
set_silent(model)
@variable(model, x[1:P] >= 0)
@objective(model, Min, sum(x))
@constraint(model, demand[i = 1:I], patterns[i]' * x == data.pieces[i].d)
model
```

```
A JuMP Model
Minimization problem with:
Variables: 20
Objective function type: AffExpr
`AffExpr`-in-`MathOptInterface.EqualTo{Float64}`: 20 constraints
`VariableRef`-in-`MathOptInterface.GreaterThan{Float64}`: 20 constraints
Model mode: AUTOMATIC
CachingOptimizer state: EMPTY_OPTIMIZER
Solver name: GLPK
Names registered in the model: demand, x
```

Then, we run the iterative column generation scheme:

```
while true
    # Solve the linear relaxation
    optimize!(model)
    # Obtain a new dual vector
    π = dual.(demand)
    # Solve the pricing problem
    new_pattern = solve_pricing(data, π)
    # Stop iterating if there is no new pattern
    if new_pattern === nothing
        break
    end
    push!(patterns, new_pattern)
    # Create a new column
    push!(x, @variable(model, lower_bound = 0))
    # Update the objective coefficients
```

```julia
        set_objective_coefficient(model, x[end], 1.0)
        # Update the non-zeros in the coefficient matrix
        for i in 1:I
            if new_pattern[i] > 0
                set_normalized_coefficient(demand[i], x[end], new_pattern[i])
            end
        end
end
```

Let's have a look at the patterns now:

```julia
SparseArrays.sparse(hcat(patterns...))
```

```
20×41 SparseArrays.SparseMatrixCSC{Int64, Int64} with 88 stored entries:
```

We found over 20 new patterns.

Here's pattern 21:

```julia
for i in 1:I
    if patterns[21][i] > 0
        println(patterns[21][i], " unit(s) of piece $i")
    end
end
```

```
1 unit(s) of piece 8
2 unit(s) of piece 13
1 unit(s) of piece 17
```

Looking at the solution

Since we only solved a linear relaxation, some of our columns have fractional solutions. We can create a integer feasible solution by rounding up the orders:

```julia
for p in 1:length(x)
    v = ceil(Int, value(x[p]))
    if v > 0
        println(lpad(v, 2), " roll(s) of pattern $p")
    end
end
```

```
31 roll(s) of pattern 1
44 roll(s) of pattern 2
30 roll(s) of pattern 3
41 roll(s) of pattern 4
15 roll(s) of pattern 5
15 roll(s) of pattern 21
23 roll(s) of pattern 22
16 roll(s) of pattern 25
 1 roll(s) of pattern 26
26 roll(s) of pattern 28
11 roll(s) of pattern 31
10 roll(s) of pattern 32
 3 roll(s) of pattern 33
19 roll(s) of pattern 34
11 roll(s) of pattern 36
 4 roll(s) of pattern 37
 8 roll(s) of pattern 38
15 roll(s) of pattern 39
12 roll(s) of pattern 40
 8 roll(s) of pattern 41
```

This requires 343 rolls:

```
sum(ceil.(Int, value.(x)))
```

```
343
```

Alternatively, we can re-introduce the integrality constraints and resolve the problem:

```
set_integer.(x)
optimize!(model)
for p in 1:length(x)
    v = round(Int, value(x[p]))
    if v > 0
        println(lpad(v, 2), " roll(s) of pattern $p, each roll of which makes:")
        for i in 1:I
            if patterns[p][i] > 0
                println("  ", patterns[p][i], " unit(s) of piece $i")
            end
        end
    end
end
```

```
33 roll(s) of pattern 1, each roll of which makes:
  1 unit(s) of piece 1
44 roll(s) of pattern 2, each roll of which makes:
  1 unit(s) of piece 2
30 roll(s) of pattern 3, each roll of which makes:
```

```
   1 unit(s) of piece 3
41 roll(s) of pattern 4, each roll of which makes:
  1 unit(s) of piece 4
 2 roll(s) of pattern 5, each roll of which makes:
  1 unit(s) of piece 5
 3 roll(s) of pattern 7, each roll of which makes:
  1 unit(s) of piece 7
 1 roll(s) of pattern 9, each roll of which makes:
  1 unit(s) of piece 9
11 roll(s) of pattern 21, each roll of which makes:
  1 unit(s) of piece 8
  2 unit(s) of piece 13
  1 unit(s) of piece 17
22 roll(s) of pattern 22, each roll of which makes:
  1 unit(s) of piece 9
  1 unit(s) of piece 12
  1 unit(s) of piece 15
  1 unit(s) of piece 20
 1 roll(s) of pattern 24, each roll of which makes:
  1 unit(s) of piece 8
  1 unit(s) of piece 12
  1 unit(s) of piece 17
  2 unit(s) of piece 20
17 roll(s) of pattern 25, each roll of which makes:
  1 unit(s) of piece 8
  1 unit(s) of piece 12
  1 unit(s) of piece 16
  1 unit(s) of piece 18
 2 roll(s) of pattern 26, each roll of which makes:
  1 unit(s) of piece 9
  1 unit(s) of piece 11
  1 unit(s) of piece 17
  2 unit(s) of piece 20
24 roll(s) of pattern 28, each roll of which makes:
  1 unit(s) of piece 7
  1 unit(s) of piece 11
  1 unit(s) of piece 14
12 roll(s) of pattern 31, each roll of which makes:
  1 unit(s) of piece 8
  1 unit(s) of piece 10
  1 unit(s) of piece 14
 9 roll(s) of pattern 32, each roll of which makes:
  1 unit(s) of piece 9
  1 unit(s) of piece 10
  2 unit(s) of piece 18
 1 roll(s) of pattern 33, each roll of which makes:
  1 unit(s) of piece 9
  1 unit(s) of piece 10
  1 unit(s) of piece 16
  1 unit(s) of piece 19
18 roll(s) of pattern 34, each roll of which makes:
  1 unit(s) of piece 6
  1 unit(s) of piece 11
  2 unit(s) of piece 19
 9 roll(s) of pattern 36, each roll of which makes:
```

```
  1 unit(s) of piece 7
  1 unit(s) of piece 12
  2 unit(s) of piece 17
 4 roll(s) of pattern 37, each roll of which makes:
  1 unit(s) of piece 5
  3 unit(s) of piece 19
  1 unit(s) of piece 20
 5 roll(s) of pattern 38, each roll of which makes:
  1 unit(s) of piece 1
  1 unit(s) of piece 15
  2 unit(s) of piece 20
15 roll(s) of pattern 39, each roll of which makes:
  1 unit(s) of piece 6
  1 unit(s) of piece 10
  1 unit(s) of piece 16
15 roll(s) of pattern 40, each roll of which makes:
  1 unit(s) of piece 5
  1 unit(s) of piece 15
  1 unit(s) of piece 17
15 roll(s) of pattern 41, each roll of which makes:
  1 unit(s) of piece 5
  1 unit(s) of piece 13
```

This now requires 334 rolls:

```
total_rolls = sum(ceil.(Int, value.(x)))
```

```
334
```

Note that this may not be the global minimum because we are not adding new columns during the solution of the mixed-integer problem model (an algorithm known as branch and price). Nevertheless, the column generation algorithm typically finds good integer feasible solutions to an otherwise intractable optimization problem.

8.3 Traveling Salesperson Problem

This tutorial was originally contributed by Daniel Schermer.

This tutorial describes how to implement the Traveling Salesperson Problem in JuMP using solver-independent lazy constraints that dynamically separate subtours. To be more precise, we use lazy constraints to cut off infeasible subtours only when necessary and not before needed.

It uses the following packages:

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

Mathematical Formulation

Assume that we are given a complete graph $\mathcal{G}(V, E)$ where V is the set of vertices (or cities) and E is the set of edges (or roads). For each pair of vertices $i, j \in V, i \neq j$ the edge $(i, j) \in E$ is associated with a weight (or distance) $d_{ij} \in \mathbb{R}^+$.

For this tutorial, we assume the problem to be symmetric, that is, $d_{ij} = d_{ji} \, \forall i, j \in V$.

In the Traveling Salesperson Problem, we are tasked with finding a tour with minimal length that visits every vertex exactly once and then returns to the point of origin, that is, a Hamiltonian cycle with minimal weight.

To model the problem, we introduce a binary variable, $x_{ij} \in \{0, 1\} \; \forall i, j \in V$, that indicates if edge (i, j) is part of the tour or not. Using these variables, the Traveling Salesperson Problem can be modeled as the following integer linear program.

Objective Function

The objective is to minimize the length of the tour (due to the assumed symmetry, the second sum only contains $i < j$):

$$\min \sum_{i \in V} \sum_{j \in V, i < j} d_{ij} x_{ij}.$$

Note that it is also possible to use the following objective function instead:

$$\min \sum_{i \in V} \sum_{j \in V} \frac{d_{ij} x_{ij}}{2}.$$

Constraints

There are four classes of constraints in our formulation.

First, due to the presumed symmetry, the following constraints must hold:

$$x_{ij} = x_{ji} \quad \forall i, j \in V.$$

Second, for each vertex i, exactly two edges must be selected that connect it to other vertices j in the graph G:

$$\sum_{j \in V} x_{ij} = 2 \quad \forall i \in V.$$

Third, we do not permit loops to occur:

$$x_{ii} = 0 \quad \forall i \in V.$$

The fourth constraint is more complicated. A major difficulty of the Traveling Salesperson Problem arises from the fact that we need to prevent subtours, that is, several distinct Hamiltonian cycles existing on subgraphs of G.

Note that the previous constraints do not guarantee that the solution will be free of subtours. To this end, by S we label a subset of vertices. Then, for each proper subset $S \subset V$, the following constraints guarantee that no subtour may occur:

$$\sum_{i \in S} \sum_{j \in S, i < j} x_{ij} \leq |S| - 1 \quad \forall S \subset V.$$

Problematically, we require exponentially many of these constraints as $|V|$ increases. Therefore, we will add these constraints only when necessary.

Implementation

There are two ways we can eliminate subtours in JuMP, both of which will be shown in what follows:

- iteratively solving a new model that incorporates previously identified subtours,

- or adding violated subtours as lazy constraints.

Data

The vertices are assumed to be randomly distributed in the Euclidean space; thus, the weight (distance) of each edge is defined as follows.

```
function generate_distance_matrix(n; random_seed = 1)
    rng = Random.MersenneTwister(random_seed)
    X = 100 * rand(rng, n)
    Y = 100 * rand(rng, n)
    d = [sqrt((X[i] - X[j])^2 + (Y[i] - Y[j])^2) for i in 1:n, j in 1:n]
    return X, Y, d
end

n = 40
X, Y, d = generate_distance_matrix(n)
```

```
([23.603334566204694, 34.651701419196044, 31.27069683360675, 0.790928339056074, 48.86128300795012,
↪    21.096820215853597, 95.1916339835734, 99.99046588986135, 25.166218303197184, 98.66663668987997
↪    …  46.33502459235987, 18.582130997265377, 11.198087695816717, 97.6311881619359,
↪    5.161462067432709, 53.80295812064833, 45.56920516275036, 27.93951106725605, 17.824610354168602,
↪    54.89828719625274], [37.097066286146884, 89.41659192657593, 64.80537482231894,
↪    41.70393538841062, 14.456554241360564, 62.24031828206811, 87.23344353741976, 52.49746566167794,
↪    24.159060827129643, 88.48369255734127   …  66.12321555087209, 19.45678064479248,
↪    39.3193497656424, 99.07406554003964, 55.03342139580574, 58.07816346526631, 76.83586278313636,
↪    51.952465724186084, 51.486297110544356, 99.813605570779374], [0.0 53.473350122820904 …
↪    15.506244459460921 70.09092934998034; 53.473350122820904 0.0 … 41.49527995497558
↪    22.760099542720535; … ; 15.506244459460921 41.49527995497558 … 0.0 60.9096566304971;
↪    70.09092934998034 22.760099542720535 … 60.9096566304971 0.0])
```

For the JuMP model, we first initialize the model object. Then, we create the binary decision variables and add the objective function and constraints. By defining the x matrix as Symmetric, we do not need to add explicit constraints that x[i, j] == x[j, i].

```julia
function build_tsp_model(d, n)
    model = Model(GLPK.Optimizer)
    @variable(model, x[1:n, 1:n], Bin, Symmetric)
    @objective(model, Min, sum(d .* x) / 2)
    @constraint(model, [i in 1:n], sum(x[i, :]) == 2)
    @constraint(model, [i in 1:n], x[i, i] == 0)
    return model
end
```

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

To search for violated constraints, based on the edges that are currently in the solution (that is, those that have value $x_{ij} = 1$), we identify the shortest cycle through the function subtour(). Whenever a subtour has been identified, a constraint corresponding to the form above can be added to the model.

```julia
function subtour(edges::Vector{Tuple{Int,Int}}, n)
    shortest_subtour, unvisited = collect(1:n), Set(collect(1:n))
    while !isempty(unvisited)
        this_cycle, neighbors = Int[], unvisited
        while !isempty(neighbors)
            current = pop!(neighbors)
            push!(this_cycle, current)
            if length(this_cycle) > 1
                pop!(unvisited, current)
            end
            neighbors =
                [j for (i, j) in edges if i == current && j in unvisited]
        end
        if length(this_cycle) < length(shortest_subtour)
            shortest_subtour = this_cycle
        end
    end
    return shortest_subtour
end
```

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

Let us declare a helper function selected_edges() that will be repeatedly used in what follows.

```julia
function selected_edges(x::Matrix{Float64}, n)
    return Tuple{Int,Int}[(i, j) for i in 1:n, j in 1:n if x[i, j] > 0.5]
end
```

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

Other helper functions for computing subtours:

```
subtour(x::Matrix{Float64}) = subtour(selected_edges(x, size(x, 1)), size(x, 1))
subtour(x::AbstractMatrix{VariableRef}) = subtour(value.(x))
```

```
subtour (generic function with 3 methods)
```

Iterative method

An iterative way of eliminating subtours is the following.

However, it is reasonable to assume that this is not the most efficient way: whenever a new subtour elimination constraint is added to the model, the optimization has to start from the very beginning.

That way, the solver will repeatedly discard useful information encountered during previous solves (for example, all cuts, the incumbent solution, or lower bounds).

> **Info**
>
> Note that, in principle, it would also be feasible to add all subtours (instead of just the shortest one) to the model. However, preventing just the shortest cycle is often sufficient for breaking other subtours and will keep the model size smaller.

```
iterative_model = build_tsp_model(d, n)
optimize!(iterative_model)
time_iterated = solve_time(iterative_model)
cycle = subtour(iterative_model[:x])
while 1 < length(cycle) < n
    println("Found cycle of length $(length(cycle))")
    S = [(i, j) for (i, j) in Iterators.product(cycle, cycle) if i < j]
    @constraint(
        iterative_model,
        sum(iterative_model[:x][i, j] for (i, j) in S) <= length(cycle) - 1,
    )
    optimize!(iterative_model)
    global time_iterated += solve_time(iterative_model)
    global cycle = subtour(iterative_model[:x])
end

objective_value(iterative_model)
```

```
525.7039004442727
```

```
time_iterated
```

```
0.037682533264160156
```

As a quick sanity check, we visualize the optimal tour to verify that no subtour is present:

```julia
function plot_tour(X, Y, x)
    plot = Plots.plot()
    for (i, j) in selected_edges(x, size(x, 1))
        Plots.plot!([X[i], X[j]], [Y[i], Y[j]]; legend = false)
    end
    return plot
end

plot_tour(X, Y, value.(iterative_model[:x]))
```
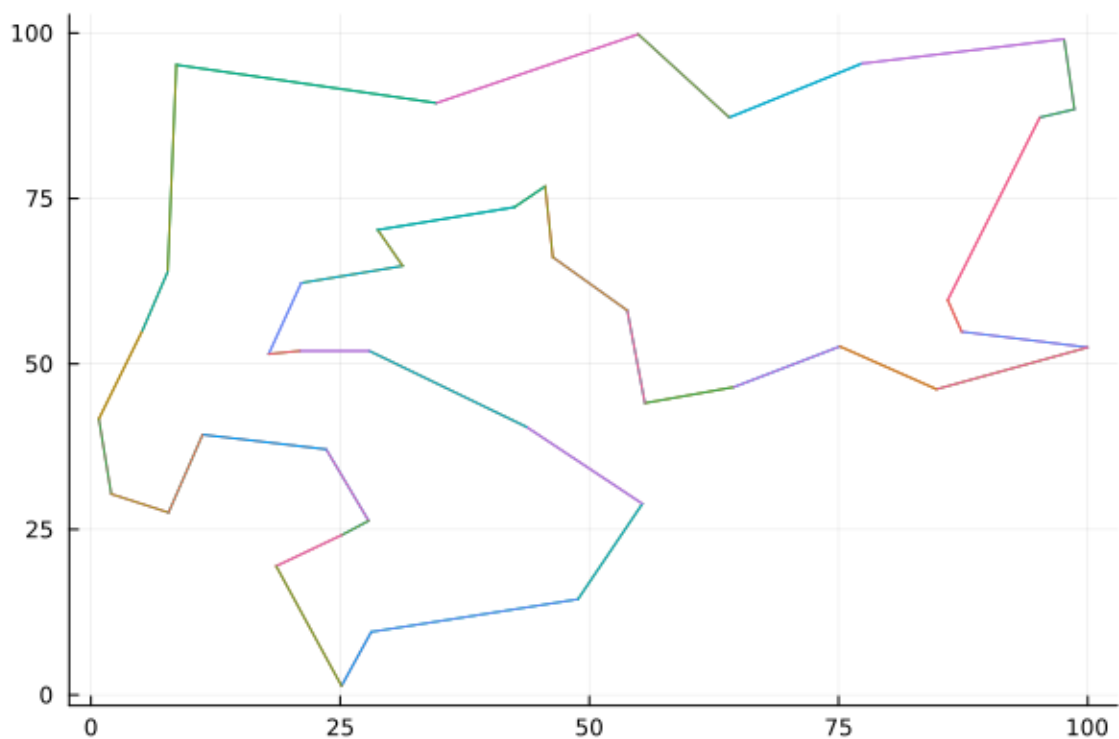

Lazy constraint method

A more sophisticated approach makes use of lazy constraints. To be more precise, we do this through the subtour_elimination_callback() below, which is only run whenever we encounter a new integer-feasible solution.

```julia
lazy_model = build_tsp_model(d, n)
function subtour_elimination_callback(cb_data)
    status = callback_node_status(cb_data, lazy_model)
    if status != MOI.CALLBACK_NODE_STATUS_INTEGER
        return  # Only run at integer solutions
    end
    cycle = subtour(callback_value.(cb_data, lazy_model[:x]))
    if !(1 < length(cycle) < n)
        return  # Only add a constraint if there is a cycle
```

```
    end
    println("Found cycle of length $(length(cycle))")
    S = [(i, j) for (i, j) in Iterators.product(cycle, cycle) if i < j]
    con = @build_constraint(
        sum(lazy_model[:x][i, j] for (i, j) in S) <= length(cycle) - 1,
    )
    MOI.submit(lazy_model, MOI.LazyConstraint(cb_data), con)
    return
end
set_attribute(
    lazy_model,
    MOI.LazyConstraintCallback(),
    subtour_elimination_callback,
)
optimize!(lazy_model)
objective_value(lazy_model)
```

```
525.7039004442727
```

This finds the same optimal tour:

```
plot_tour(X, Y, value.(lazy_model[:x]))
```
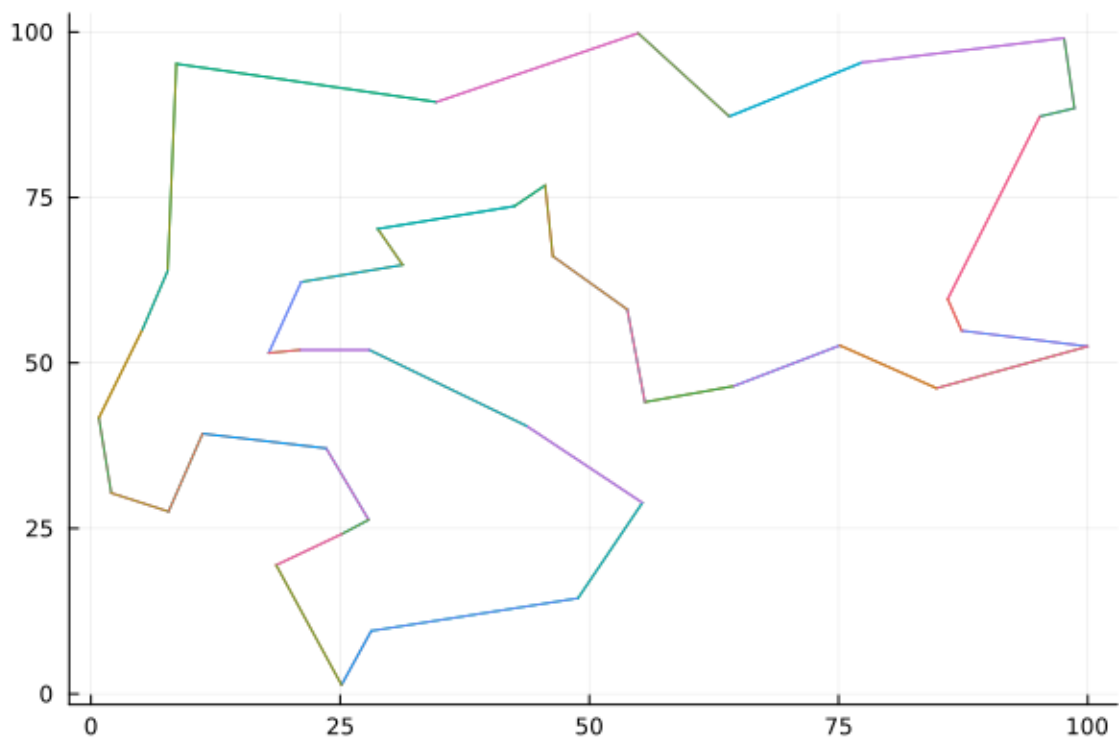

Surprisingly, for this particular model with GLPK, the solution time is worse than the iterative method:

```
time_lazy = solve_time(lazy_model)
```

```
0.15597081184387207
```

In most other cases and solvers, however, the lazy time should be faster than the iterative method.

8.4 Parallelism

The purpose of this tutorial is to give a brief overview of parallelism in Julia as it pertains to JuMP, and to explain some of the things to be aware of when writing parallel algorithms involving JuMP models.

Multi-threading and distributed computing

There are two main types of parallelism in Julia:

1. Multi-threading

2. Distributed computing

In multi-threading, multiple tasks are run in a single Julia process and share the same memory. In distributed computing, tasks are run in multiple Julia processes with independent memory spaces. This can include processes across multiple physical machines, such as in a high-performance computing cluster.

Choosing and understanding the type of parallelism you are using is important because the code you write for each type is different, and there are different limitations and benefits to each approach. However, the best choice is highly problem dependent, so you may want to experiment with both approaches to determine what works for your situation.

Multi-threading

To use multi-threading with Julia, you must either start Julia with the command line flag `--threads=N`, or you must set the JULIA_NUM_THREADS environment variable before launching Julia. For this documentation, we set the environment variable to:

```
julia> ENV["JULIA_NUM_THREADS"]
"4"
```

You can check how many threads are available using:

```
julia> Threads.nthreads()
4
```

The easiest way to use multi-threading in Julia is by placing the `Threads.@threads` macro in front of a `for`-loop:

```
julia> @time begin
           ids = Int[]
           my_lock = Threads.ReentrantLock()
           Threads.@threads for i in 1:Threads.nthreads()
               global ids, my_lock
               Threads.lock(my_lock) do
                   push!(ids, Threads.threadid())
               end
               sleep(1.0)
           end
       end
  1.037087 seconds (31.32 k allocations: 1.836 MiB, 2.02% compilation time)
```

This for-loop sleeps for 1 second on each iteration. Thus, if it had executed sequentially, it should have taken the same number of seconds as there are threads available. Instead, it took only 1 second, showing that the iterations were executed simultaneously. We can verify this by checking the Threads.threadid() of the thread that executed each iteration:

```
julia> ids
4-element Vector{Int64}:
 2
 4
 1
 3
```

> **Warn**
>
> When working with threads, you need to avoid race conditions, in which two threads attempt to write to the same variable at the same time. In the above example we avoided a race condition by using ReentrantLock. See the Multi-threading section of the Julia documentation for more details.

Distributed computing

To use distributed computing with Julia, use the Distributed package:

```
julia> import Distributed
```

Like multi-threading, we need to tell Julia how many processes to add. We can do this either by starting Julia with the -p N command line argument, or by using Distributed.addprocs:

```
julia> import Pkg

julia> project = Pkg.project();

julia> workers = Distributed.addprocs(4; exeflags = "--project=$(project.path)")
4-element Vector{Int64}:
 2
 3
 4
 5
```

> **Warning**
>
> Not loading the parent environment with `--project` is a common mistake.

The added processes are "worker" processes that we can use to do computation with. They are orches-
trated by the process with the id 1. You can check what process the code is currently running on using
`Distributed.myid()`

```julia
julia> Distributed.myid()
1
```

As a general rule, to get maximum performance you should add as many processes as you have logical cores
available.

Unlike the `for`-loop approach of multi-threading, distributed computing extends the Julia map function to a
"parallel-map" function `Distributed.pmap`. For each element in the list of arguments to map over, Julia will
copy the element to an idle worker process and evaluate the function, passing the element as an input argu-
ment.

```julia
julia> function hard_work(i::Int)
           sleep(1.0)
           return Distributed.myid()
       end
hard_work (generic function with 1 method)

julia> Distributed.pmap(hard_work, 1:4)
ERROR: On worker 2:
UndefVarError: #hard_work not defined
Stacktrace:
[...]
```

Unfortunately, if you try this code directly, you will get an error message that says `On worker 2: UndefVarError:`
`hard_work not defined`. The error is thrown because, although process 1 knows what the `hard_work` function
is, the worker processes do not.

To fix the error, we need to use `Distributed.@everywhere`, which evaluates the code on every process:

```julia
julia> Distributed.@everywhere begin
           function hard_work(i::Int)
               sleep(1.0)
               return Distributed.myid()
           end
       end
```

Now if we run pmap, we see that it took only 1 second instead of 4, and that it executed on each of the worker
processes:

```julia
julia> @time ids = Distributed.pmap(hard_work, 1:4)
  1.202006 seconds (216.39 k allocations: 13.301 MiB, 4.07% compilation time)
4-element Vector{Int64}:
 2
```

```
3
5
4
```

> **Tip**
>
> For more information, read the Julia documentation Distributed Computing.

Using parallelism the wrong way

When building a JuMP model

For large problems, building the model in JuMP can be a bottleneck, and you may consider trying to write code that builds the model in parallel, for example, by wrapping a `for`-loop that adds constraints with `Threads.@threads`.

Unfortunately, you cannot build a JuMP model in parallel, and attempting to do so may error or produce incorrect results.

In most cases, we find that the reason for the bottleneck is not JuMP, but in how you are constructing the problem data, and that with changes, it is possible to build a model in a way that is not the bottleneck in the solution process.

> **Tip**
>
> Looking for help to make your code run faster? Ask for help on the community forum. Make sure to include a reproducible example of your code.

With a single JuMP model

A common approach people try is to use parallelism with a single JuMP model. For example, to optimize a model over multiple right-hand side vectors, you may try:

```julia
using JuMP
import HiGHS
model = Model(HiGHS.Optimizer)
set_silent(model)
@variable(model, x)
@objective(model, Min, x)
solutions = Pair{Int,Float64}[]
my_lock = Threads.ReentrantLock()
Threads.@threads for i in 1:10
    set_lower_bound(x, i)
    optimize!(model)
    Threads.lock(my_lock) do
        push!(solutions, i => objective_value(model))
    end
end
```

This will not work, and attempting to do so may error, crash Julia or produce incorrect results.

Using parallelism the right way

To use parallelism with JuMP, the simplest rule to remember is that each worker must have its own instance of a JuMP model.

With multi-threading

With multi-threading, create a new instance of `model` in each iteration of the for-loop:

```julia
julia> using JuMP

julia> import HiGHS

julia> solutions = Pair{Int,Float64}[]

julia> my_lock = Threads.ReentrantLock();

julia> Threads.@threads for i in 1:10
           model = Model(HiGHS.Optimizer)
           set_silent(model)
           set_attribute(model, MOI.NumberOfThreads(), 1)
           @variable(model, x)
           @objective(model, Min, x)
           set_lower_bound(x, i)
           optimize!(model)
           Threads.lock(my_lock) do
               push!(solutions, i => objective_value(model))
           end
       end

julia> solutions
10-element Vector{Pair{Int64, Float64}}:
   7 => 7.0
   4 => 4.0
   1 => 1.0
   9 => 9.0
   5 => 5.0
   8 => 8.0
  10 => 10.0
   2 => 2.0
   6 => 6.0
   3 => 3.0
```

> **Warning**
>
> For some solvers, it may be necessary to limit the number of threads used internally by the solver to 1 by setting the `MOI.NumberOfThreads` attribute.

With distributed computing

With distributed computing, remember to evaluate all of the code on all of the processes using `Distributed.@everywhere`, and then write a function which creates a new instance of the model on every evaluation:

```
julia> Distributed.@everywhere begin
           using JuMP
           import HiGHS
       end

julia> Distributed.@everywhere begin
           function solve_model_with_right_hand_side(i)
               model = Model(HiGHS.Optimizer)
               set_silent(model)
               @variable(model, x)
               @objective(model, Min, x)
               set_lower_bound(x, i)
               optimize!(model)
               return objective_value(model)
           end
       end

julia> solutions = Distributed.pmap(solve_model_with_right_hand_side, 1:10)
10-element Vector{Float64}:
  1.0
  2.0
  3.0
  4.0
  5.0
  6.0
  7.0
  8.0
  9.0
 10.0
```

Other types of parallelism

GPU

JuMP does not support GPU programming, and few solvers support execution on a GPU.

Parallelism within the solver

Many solvers use parallelism internally. For example, commercial solvers like Gurobi and CPLEX both parallelize the search in branch-and-bound. Solvers supporting internal parallelism will typically support the MOI.NumberOfThreads attribute, which you can set using set_attribute.

Chapter 9

Applications

9.1 Power Systems

This tutorial was 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:

```julia
using JuMP
import DataFrames
import HiGHS
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 ith 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:

```julia
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

```julia
WindGenerator(variable_cost::Float64) = (variable_cost = variable_cost,)

wind_generator = WindGenerator(50.0)
```

```
(variable_cost = 50.0,)
```

And a scenario

```julia
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_economic_dispatch`, which solves the economic dispatch problem for a given set of input parameters.

```julia
function solve_economic_dispatch(generators::Vector, wind, scenario)
    # Define the economic dispatch (ED) model
    model = Model(HiGHS.Optimizer)
    set_silent(model)
    # Define decision variables
    # power output of generators
    N = length(generators)
    @variable(model, generators[i].min <= g[i = 1:N] <= generators[i].max)
    # wind power injection
    @variable(model, 0 <= w <= scenario.wind)
    # Define the objective function
    @objective(
        model,
        Min,
        sum(generators[i].variable_cost * g[i] for i in 1:N) +
        wind.variable_cost * w,
    )
    # Define the power balance constraint
    @constraint(model, sum(g[i] for i in 1:N) + w == scenario.demand)
    # Solve statement
    optimize!(model)
    # 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(model),
    )
end
```

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

Solve the economic dispatch problem

```julia
solution = solve_economic_dispatch(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.

```julia
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 economic-dispatch problem with the updated incremental cost
    sol = solve_economic_dispatch(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: 0.14911293983459473 seconds
```

```
c_g_scale_df
```

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 the constraints or objective function, as it shown in the example below.

	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	300.0	1000.0	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

Compare the computing time in case of the above and below models.

```julia
function solve_economic_dispatch_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
    model = Model(HiGHS.Optimizer)
    set_silent(model)
    N = length(generators)
    @variable(model, generators[i].min <= g[i = 1:N] <= generators[i].max)
    @variable(model, 0 <= w <= scenario.wind)
    @objective(
        model,
        Min,
        sum(generators[i].variable_cost * g[i] for i in 1:N) +
        wind.variable_cost * w,
```

```julia
    )
    @constraint(model, sum(g[i] for i in 1:N) + w == scenario.demand)
    for c_g1_scale in scale
        @objective(
            model,
            Min,
            c_g1_scale * generators[1].variable_cost * g[1] +
            generators[2].variable_cost * g[2] +
            wind.variable_cost * w,
        )
        optimize!(model)
        push!(obj_out, objective_value(model))
        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_economic_dispatch_inplace(
    generators,
    wind_generator,
    scenario,
    0.5:0.1:3.0,
)
print(string("elapsed time: ", time() - start, " seconds"))
```

```
elapsed time: 0.21984601020812988 seconds
```

For small models, adjusting specific constraints or the objective function is sometimes faster and sometimes slower than re-building the entire model. However, as the problem size increases, updating the model in-place is usually faster.

```
inplace_df
```

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, that is, all available wind generation is spilled, to meet the minimum power output constraints on generators.

	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

In the following example, we adjust the total demand and observed how it affects wind spillage.

```julia
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 0.2:0.1:1.4
    new_scenario = scale_demand(scenario, demand_scale)
    sol = solve_economic_dispatch(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	850.0	300.0	200.0	0.0	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

```
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)
```
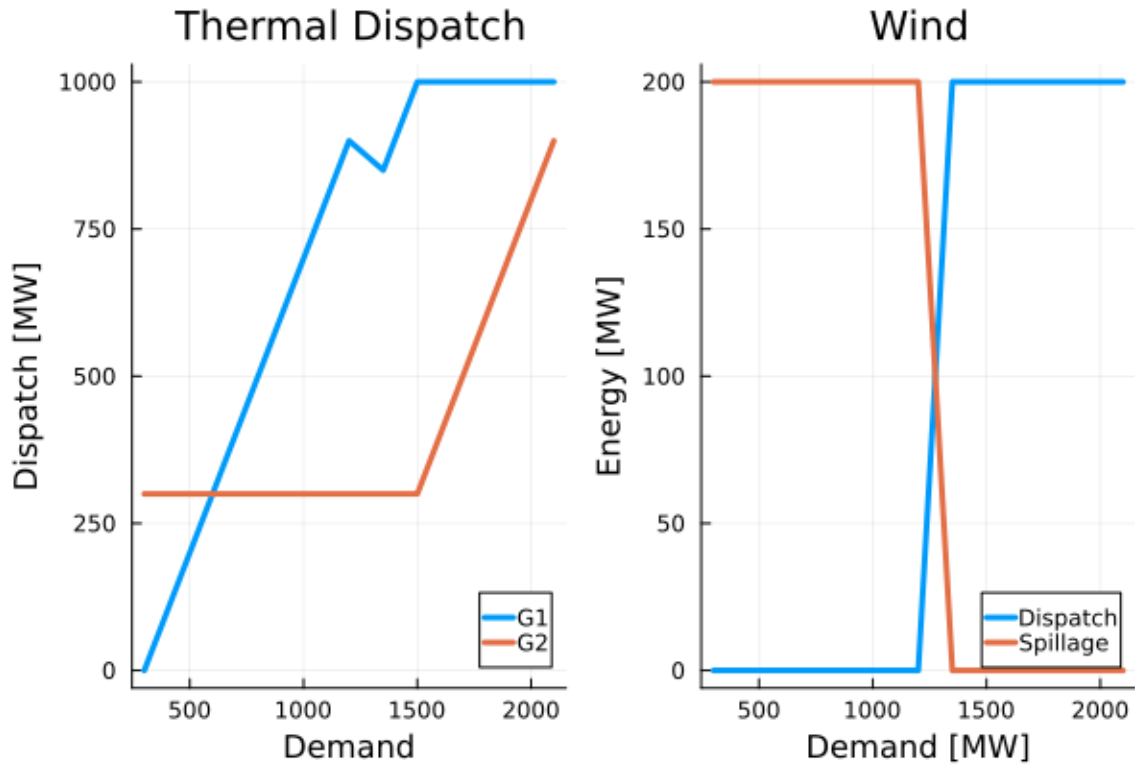

This particular drawback can be overcome by introducing binary decisions on the "on/off" status of generators. This model is called unit commitment and considered later in these notes.

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, that is, 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.

```julia
function solve_unit_commitment(generators::Vector, wind, scenario)
    model = Model(HiGHS.Optimizer)
    set_silent(model)
    N = length(generators)
    @variable(model, 0 <= g[i = 1:N] <= generators[i].max)
    @variable(model, 0 <= w <= scenario.wind)
    @constraint(model, sum(g[i] for i in 1:N) + w == scenario.demand)
    # !!! New: add binary on-off variables for each generator
    @variable(model, u[i = 1:N], Bin)
    @constraint(model, [i = 1:N], g[i] <= generators[i].max * u[i])
    @constraint(model, [i = 1:N], g[i] >= generators[i].min * u[i])
    @objective(
        model,
        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!(model)
    status = termination_status(model)
    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(model),
    )
end
```

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

Solve the unit commitment problem

```julia
solution = solve_unit_commitment(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 unit commitment model, we can now assess the interplay between the minimum power output constraints on generators and wind generation.

```julia
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 0.2:0.1:1.4
    new_scenario = scale_demand(scenario, demand_scale)
    sol = solve_unit_commitment(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
```

```
uc_df
```

	demand	commitment_G1	commitment_G2	dispatch_G1	dispatch_G2	dispatch_wind	spillage_wind	total_cost
	Float64	Float64	Float64	Float64	Float64	Float64	Float64	Float64
1	300.0	1.0	0.0	100.0	0.0	200.0	0.0	16000.0
2	450.0	1.0	0.0	250.0	0.0	200.0	0.0	23500.0
3	600.0	1.0	0.0	400.0	0.0	200.0	0.0	31000.0
4	750.0	1.0	0.0	550.0	0.0	200.0	0.0	38500.0
5	900.0	1.0	0.0	700.0	0.0	200.0	0.0	46000.0
6	1050.0	1.0	0.0	850.0	0.0	200.0	0.0	53500.0
7	1200.0	1.0	0.0	1000.0	0.0	200.0	0.0	61000.0
8	1350.0	1.0	1.0	850.0	300.0	200.0	0.0	83500.0
9	1500.0	1.0	1.0	1000.0	300.0	200.0	0.0	91000.0
10	1650.0	1.0	1.0	1000.0	450.0	200.0	0.0	106000.0
11	1800.0	1.0	1.0	1000.0	600.0	200.0	0.0	121000.0
12	1950.0	1.0	1.0	1000.0	750.0	200.0	0.0	136000.0
13	2100.0	1.0	1.0	1000.0	900.0	200.0	0.0	151000.0

```julia
commitment_plot = StatsPlots.@df(
    uc_df,
    Plots.plot(
        :demand,
        [:commitment_G1, :commitment_G2],
        labels = ["G1" "G2"],
        title = "Commitment",
        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)
```
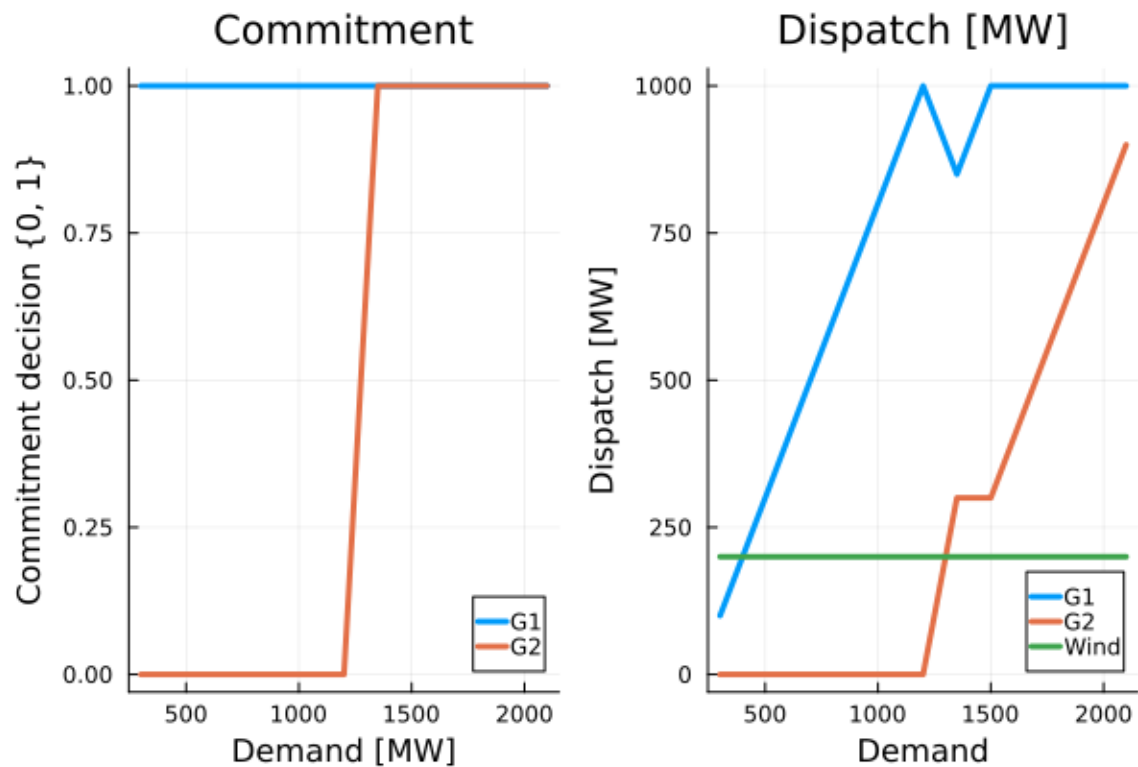
Nonlinear economic dispatch

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
```

```julia
function solve_nonlinear_economic_dispatch(
    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_economic_dispatch(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:

```julia
wind_cost = 0.0:1:100
wind_dispatch = Float64[]
for c in wind_cost
    sol = solve_nonlinear_economic_dispatch(
        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,
)
```
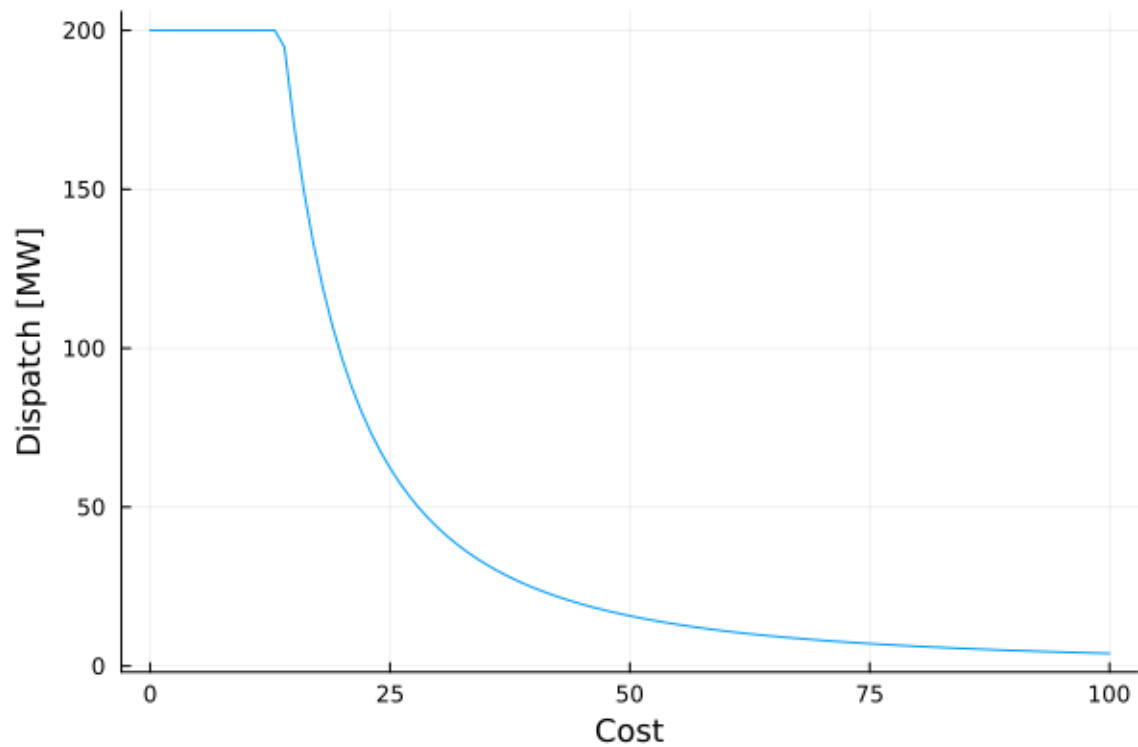

9.2 Serving web apps

This tutorial demonstrates how to setup and serve JuMP models via a REST API.

In the example app we are building, we solve a trivial mixed-integer program, which is parameterized by the lower bound of a variable. To call the service, users send an HTTP POST request with JSON contents indicating the lower bound. The returned value is the solution of the mixed-integer program as JSON.

First, we need JuMP and a solver:

```
using JuMP
import HiGHS
```

We also need HTTP.jl to act as our REST server, and JSON.jl to marshal data.

```
import HTTP
import JSON
```

The server side

The core components of our REST server are endpoints. These are functions which accept a `Dict{String,Any}` of input parameters, and return a `Dict{String,Any}` as output. The types are `Dict{String,Any}` because we're going to read these to and from JSON.

Here's a very simple endpoint: it accepts `params` as input, formulates and solves a trivial mixed-integer program, and then returns a dictionary with the result.

```julia
function endpoint_solve(params::Dict{String,Any})
    if !haskey(params, "lower_bound")
        return Dict{String,Any}(
            "status" => "failure",
            "reason" => "missing lower_bound param",
        )
    elseif !(params["lower_bound"] isa Real)
        return Dict{String,Any}(
            "status" => "failure",
            "reason" => "lower_bound is not a number",
        )
    end
    model = Model(HiGHS.Optimizer)
    set_silent(model)
    @variable(model, x >= params["lower_bound"], Int)
    optimize!(model)
    ret = Dict{String,Any}(
        "status" => "okay",
        "terminaton_status" => termination_status(model),
        "primal_status" => primal_status(model),
    )
    # Only include the `x` key if it has a value.
    if has_values(model)
        ret["x"] = value(x)
    end
    return ret
end
```

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

When we call this, we get:

```julia
endpoint_solve(Dict{String,Any}("lower_bound" => 1.2))
```

```
Dict{String, Any} with 4 entries:
  "status"            => "okay"
  "x"                 => 2.0
  "primal_status"     => FEASIBLE_POINT
  "terminaton_status" => OPTIMAL
```

```
endpoint_solve(Dict{String,Any}())
```

```
Dict{String, Any} with 2 entries:
  "status" => "failure"
  "reason" => "missing lower_bound param"
```

For a second function, we need a function that accepts an HTTP.Request object and returns an HTTP.Response object.

```
function serve_solve(request::HTTP.Request)
    data = JSON.parse(String(request.body))
    solution = endpoint_solve(data)
    return HTTP.Response(200, JSON.json(solution))
end
```

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

Finally, we need an HTTP server. There are a variety of ways you can do this in HTTP.jl. We use an explicit Sockets.listen so we have manual control of when we shutdown the server.

```
function setup_server(host, port)
    server = HTTP.Sockets.listen(host, port)
    HTTP.serve!(host, port; server = server) do request
        try
            # Extend the server by adding other endpoints here.
            if request.target == "/api/solve"
                return serve_solve(request)
            else
                return HTTP.Response(404, "target $(request.target) not found")
            end
        catch err
            # Log details about the exception server-side
            @info "Unhandled exception: $err"
            # Return a response to the client
            return HTTP.Response(500, "internal error")
        end
    end
    return server
end
```

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

> **Warning**
>
> HTTP.jl does not serve requests on a separate thread. Therefore, a long-running job will block the main thread, preventing concurrent users from submitting requests. To work-around this, read HTTP.jl issue 798 or watch Building Microservices and Applications in Julia from JuliaCon 2020.

```julia
server = setup_server(HTTP.ip"127.0.0.1", 8080)
```

```
Sockets.TCPServer(RawFD(25) active)
```

The client side

Now that we have a server, we can send it requests via this function:

```julia
function send_request(data::Dict; endpoint::String = "solve")
    ret = HTTP.request(
        "POST",
        # This should match the URL and endpoint we defined for our server.
        "http://127.0.0.1:8080/api/$endpoint",
        ["Content-Type" => "application/json"],
        JSON.json(data),
    )
    if ret.status != 200
        # This could happen if there are time-outs, network errors, etc.
        return Dict(
            "status" => "failure",
            "code" => ret.status,
            "body" => String(ret.body),
        )
    end
    return JSON.parse(String(ret.body))
end
```

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

Let's see what happens:

```julia
send_request(Dict("lower_bound" => 0))
```

```
Dict{String, Any} with 4 entries:
  "status"            => "okay"
  "x"                 => 0.0
  "primal_status"     => "FEASIBLE_POINT"
  "terminaton_status" => "OPTIMAL"
```

```julia
send_request(Dict("lower_bound" => 1.2))
```

```
Dict{String, Any} with 4 entries:
  "status"            => "okay"
  "x"                 => 2.0
  "primal_status"     => "FEASIBLE_POINT"
  "terminaton_status" => "OPTIMAL"
```

If we don't send a `lower_bound`, we get:

```julia
send_request(Dict("invalid_param" => 1.2))
```

```
Dict{String, Any} with 2 entries:
  "status" => "failure"
  "reason" => "missing lower_bound param"
```

If we don't send a `lower_bound` that is a number, we get:

```julia
send_request(Dict("lower_bound" => "1.2"))
```

```
Dict{String, Any} with 2 entries:
  "status" => "failure"
  "reason" => "lower_bound is not a number"
```

Finally, we can shutdown our HTTP server:

```julia
close(server)
```

Next steps

For more complicated examples relating to HTTP servers, consult the HTTP.jl documentation.

To see how you can integrate this with a larger JuMP model, read Design patterns for larger models.

Part III

Manual

Chapter 10

Models

JuMP models are the fundamental building block that we use to construct optimization problems. They hold things like the variables and constraints, as well as which solver to use and even solution information.

> **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, `HiGHS` is a solver, and `HiGHS.Optimizer` is an optimizer.

> **Tip**
>
> See Supported solvers for a list of available solvers.

10.1 Create a model

Create a model by passing an optimizer to GenericModel:

```julia
julia> model = Model(HiGHS.Optimizer)
A JuMP Model
Feasibility problem with:
Variables: 0
Model mode: AUTOMATIC
CachingOptimizer state: EMPTY_OPTIMIZER
Solver name: HiGHS
```

If you don't know which optimizer you will be using at creation time, create a model without an optimizer, and then call set_optimizer at any time prior to optimize!:

```julia
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, HiGHS.Optimizer)
```

> **Tip**
>
> Don't know what the fields Model mode and CachingOptimizer state mean? Read the Backends section.

What is the difference?

For most models, there is no difference between passing the optimizer to GenericModel, 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 GenericModel 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 are building 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,s" try passing add_bridges = false.

```
julia> model = Model(HiGHS.Optimizer; add_bridges = false);
```

or

```
julia> model = Model();

julia> set_optimizer(model, HiGHS.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 your 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.
[...]
```

Solvers which expect environments

Some solvers accept (or require) positional arguments such as a license environment or a path to a binary executable. For these solvers, you can pass a function to GenericModel which takes zero arguments and returns an instance of the optimizer.

A common use-case for this is passing an environment or sub-solver to the optimizer:

```
julia> import HiGHS

julia> import MultiObjectiveAlgorithms as MOA

julia> model = Model(() -> MOA.Optimizer(HiGHS.Optimizer))
A JuMP Model
Feasibility problem with:
Variables: 0
Model mode: AUTOMATIC
CachingOptimizer state: EMPTY_OPTIMIZER
Solver name: MOA[algorithm=MultiObjectiveAlgorithms.Lexicographic, optimizer=HiGHS]
```

10.2 Solver options

JuMP uses "attribute" as a synonym for "option." Use optimizer_with_attributes to create an optimizer with some attributes initialized:

```
julia> model = Model(
           optimizer_with_attributes(HiGHS.Optimizer, "output_flag" => false),
       )
A JuMP Model
Feasibility problem with:
Variables: 0
Model mode: AUTOMATIC
CachingOptimizer state: EMPTY_OPTIMIZER
Solver name: HiGHS
```

Alternatively, use set_attribute to set an attribute after the model has been created:

```
julia> model = Model(HiGHS.Optimizer);
```

```julia
julia> set_attribute(model, "output_flag", false)

julia> get_attribute(model, "output_flag")
false
```

You can also modify attributes within an optimizer_with_attributes object:

```julia
julia> solver = optimizer_with_attributes(HiGHS.Optimizer, "output_flag" => true);

julia> get_attribute(solver, "output_flag")
true

julia> set_attribute(solver, "output_flag", false)

julia> get_attribute(solver, "output_flag")
false

julia> model = Model(solver);
```

10.3   Changing the number types

By default, the coefficients of affine and quadratic expressions are numbers of type either Float64 or Complex{Float64} (see Complex number support). The type Float64 can be changed using the GenericModel type parameter as follows:

```julia
julia> model = GenericModel{Rational{BigInt}}();

julia> @variable(model, x)
x

julia> a = x / 3
1//3 x

julia> typeof(a)
GenericAffExpr{Rational{BigInt}, GenericVariableRef{Rational{BigInt}}}
```

Note that this should mostly be used if the underlying solver actually solves the problem using a number type different from Float64.

> **Warning**
>
> Nonlinear Modeling is currently only supported with Float64 number type.

10.4   Print the model

By default, show(model) will print a summary of the problem:

```julia
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
```

> **Warning**
>
> This format is specific to JuMP and may change in any future release. It is not intended to be an instance format. 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)
$$ \begin{aligned}
\max\quad & x\\
\text{Subject to} \quad & x \geq 0\\
\end{aligned} $$
```

In IJulia (and Documenter), ending a cell in with `latex_formulation` will render the model in LaTeX:

```
latex_formulation(model)
```

$$
\begin{aligned}
\max\quad & x\\
\text{Subject to} \quad & x \geq 0
\end{aligned}
$$

10.5 Turn off output

Use `set_silent` and `unset_silent` to disable or enable printing output from the solver.

```
julia> model = Model(HiGHS.Optimizer);

julia> set_silent(model)

julia> unset_silent(model)
```

> **Tip**
>
> Most solvers will also have a solver-specific option to provide finer-grained control over the output. Consult their README's for details.

10.6 Set a time limit

Use set_time_limit_sec, unset_time_limit_sec, and time_limit_sec to manage time limits.

```julia
julia> model = Model(HiGHS.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)
Inf
```

If your time limit is encoded as a `Dates.Period` object, use the following code to convert it to `Float64` for set_time_limit_sec:

```julia
julia> import Dates

julia> seconds(x::Dates.Period) = 1e-3 * Dates.value(round(x, Dates.Millisecond))
seconds (generic function with 1 method)

julia> set_time_limit_sec(model, seconds(Dates.Hour(1)))

julia> time_limit_sec(model)
3600.0
```

> **Info**
>
> Some solvers do not support time limits. In these cases, an error will be thrown.

10.7 Write a model to file

JuMP can write models to a variety of file-formats using write_to_file and Base.write.

For most common file formats, the file type will be detected from the extension.

For example, here is how to write an MPS file:

```julia
julia> model = Model();

julia> write_to_file(model, "model.mps")
```

Other supported file formats include:

- `.cbf` for the Conic Benchmark Format

- `.lp` for the LP file format

- `.mof.json` for the MathOptFormat

- `.nl` for AMPL's NL file format

- `.rew` for the REW file format

- `.sdpa` and ".dat-s" for the SDPA file format

To write to a specific `io::IO`, use `Base.write`. Specify the file type by passing a `MOI.FileFormats.FileFormat` enum.

```julia
julia> model = Model();

julia> io = IOBuffer();

julia> write(io, model; format = MOI.FileFormats.FORMAT_MPS)
```

10.8 Read a model from file

JuMP models can be created from file formats using `read_from_file` and `Base.read`.

```julia
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.

10.9   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 to re-add the removed constraints:

```julia
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 (for example, 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, instead of:

```julia
using JuMP, Cbc
model = Model(Cbc.Optimizer)
@variable(model, x, Int)
undo = relax_integrality(model)
optimize!(model)
reduced_cost(x)  # Errors
```

do:

```julia
using JuMP, Cbc, Clp
model = Model(Cbc.Optimizer)
@variable(model, x, Int)
undo = relax_integrality(model)
set_optimizer(model, Clp.Optimizer)
optimize!(model)
reduced_cost(x)  # Works
```

10.10 Backends

> **Info**
>
> This section discusses advanced features of JuMP. For new users, you may want to skip this section. You don't need to know how JuMP manages problems behind the scenes to create and solve JuMP models.

A JuMP GenericModel is a thin layer around a backend of type MOI.ModelLike that stores the optimization problem and acts as the optimization solver.

However, if you construct a model like Model(HiGHS.Optimizer), the backend is not a HiGHS.Optimizer, but a more complicated object.

From JuMP, the MOI backend can be accessed using the backend function. Let's see what the backend of a JuMP GenericModel is:

```
julia> model = Model(HiGHS.Optimizer);

julia> b = backend(model)
MOIU.CachingOptimizer{MOIB.LazyBridgeOptimizer{HiGHS.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{HiGHS.Optimizer}
  with 0 variable bridges
  with 0 constraint bridges
  with 0 objective bridges
  with inner model A HiGHS model with 0 columns and 0 rows.
```

Uh oh. Even though we passed a HiGHS.Optimizer, the backend is a much more complicated object.

CachingOptimizer

A MOIU.CachingOptimizer is a layer that abstracts the difference between solvers that support incremental modification (for example, they support adding variables one-by-one), and solvers that require the entire problem in a single API call (for example, 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{HiGHS.Optimizer}
   with 0 variable bridges
   with 0 constraint bridges
   ```

```
with 0 objective bridges
with inner model A HiGHS model with 0 columns and 0 rows.
```

> **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 GenericModel will create a `CachingOptimizer` in AUTOMATIC mode.

LazyBridgeOptimizer

The second layer that JuMP applies automatically is a MOI.Bridges.LazyBridgeOptimizer. A MOI.Bridges.LazyBridgeOptimizer is an MOI layer that attempts to transform the problem from the formulation provided by the user into an equivalent problem 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 constraint 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
julia> model = Model(HiGHS.Optimizer; add_bridges = false)
A JuMP Model
Feasibility problem with:
Variables: 0
Model mode: AUTOMATIC
CachingOptimizer state: EMPTY_OPTIMIZER
```

```
Solver name: HiGHS

julia> backend(model)
MOIU.CachingOptimizer{HiGHS.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 HiGHS model with 0 columns and 0 rows.
```

Bridges can be added and removed from a `MOI.Bridges.LazyBridgeOptimizer` using `add_bridge` and `remove_bridge`. Use `print_active_bridges` to see which bridges are used to reformulate the model. Read the Ellipsoid approximation tutorial for more details.

Unsafe backend

In some advanced use-cases, it is necessary to work with the inner optimization model directly. To access this model, use `unsafe_backend`:

```
julia> backend(model)
MOIU.CachingOptimizer{MOIB.LazyBridgeOptimizer{HiGHS.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{HiGHS.Optimizer}
  with 0 variable bridges
  with 0 constraint bridges
  with 0 objective bridges
  with inner model A HiGHS model with 0 columns and 0 rows.

julia> unsafe_backend(model)
A HiGHS model with 0 columns and 0 rows.
```

> **Warning**
>
> `backend` and `unsafe_backend` are advanced routines. Read their docstrings to understand the caveats of their usage, and only call them if you wish to access low-level solver-specific functions.

10.11 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(HiGHS.Optimizer())
A JuMP Model
Feasibility problem with:
Variables: 0
Model mode: DIRECT
Solver name: HiGHS
```

> **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 (for example, Cachingoptimizer or LazyBridgeOptimizer) between model and the provided optimizer:

```julia
julia> backend(model)
A HiGHS model with 0 columns and 0 rows.
```

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, `HiGHS.jl` does not implement quadratic constraints:

```julia
julia> model = direct_model(HiGHS.Optimizer());

julia> set_silent(model)

julia> @variable(model, x[1:2]);

julia> @constraint(model, x[1]^2 + x[2]^2 <= 2)
ERROR: Constraints of type
↪   MathOptInterface.ScalarQuadraticFunction{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 your 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.
Stacktrace:
```

> **Warning**
>
> Another downside of direct mode is that the behavior of querying solution information after modifying the problem is solver-specific. This can lead to errors, or the solver silently returning an incorrect value. See OptimizeNotCalled errors for more information.

Chapter 11

Variables

The term variable in mathematical optimization has many meanings. For example, optimization variables (also called decision variables) are the unknowns x that we are solving for in the problem:

$$\min_{x \in \mathbb{R}^n} \quad f_0(x) \tag{11.1}$$

$$\text{s.t.} \quad f_i(x) \in \mathcal{S}_i \qquad i = 1 \dots m \tag{11.2}$$

To complicate things, Julia uses variable to mean a binding between a name and a value. For example, in the statement:

```julia
julia> x = 1
1
```

x is a variable that stores the value 1.

JuMP uses variable in a third way, to mean an instance of the VariableRef struct. JuMP variables are the link between Julia and the optimization variables inside a JuMP model.

This page explains how to create and manage JuMP variables in a variety of contexts.

11.1   Create a variable

Create variables using the @variable macro:

```julia
julia> model = Model();

julia> @variable(model, x)
x

julia> typeof(x)
VariableRef (alias for GenericVariableRef{Float64})

julia> num_variables(model)
1
```

Here x is a Julia variable that is bound to a VariableRef object, and we have added 1 decision variable to our model.

To make the binding more explicit, we could have written:

```julia
julia> model = Model();

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

but there is no need to in general; the macro does it for us.

When creating a variable, you can also specify variable bounds:

```julia
julia> model = Model();

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

julia> print(model)
Feasibility
Subject to
 x_fixed = 4
 x_lower ≥ 0
 x_interval ≥ 2
 x_upper ≤ 1
 x_interval ≤ 3
```

> **Warning**
>
> When creating a variable with a single lower- or upper-bound, and the value of the bound is not a numeric literal (for example, 1 or 1.0), the name of the variable must appear on the left-hand side. Putting the name on the right-hand side is 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
> ```

Containers of variables

The @variable macro also supports creating collections of JuMP variables. We'll cover some brief syntax here; read the Variable containers section for more details.

You can create arrays of JuMP variables:

```julia
julia> model = Model();

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> x[1, 2]
x[1,2]
```

Index sets can be named, and bounds can depend on those names:

```julia
julia> model = Model();

julia> @variable(model, sqrt(i) <= x[i = 1:3] <= i^2)
3-element Vector{VariableRef}:
 x[1]
 x[2]
 x[3]

julia> x[2]
x[2]
```

Sets can be any Julia type that supports iteration:

```julia
julia> model = Model();

julia> @variable(model, x[i = 2:3, j = 1:2:3, ["red", "blue"]] >= 0)
3-dimensional DenseAxisArray{VariableRef,3,...} with index sets:
    Dimension 1, 2:3
    Dimension 2, 1:2:3
    Dimension 3, ["red", "blue"]
And data, a 2×2×2 Array{VariableRef, 3}:
[:, :, "red"] =
 x[2,1,red]  x[2,3,red]
 x[3,1,red]  x[3,3,red]

[:, :, "blue"] =
 x[2,1,blue]  x[2,3,blue]
 x[3,1,blue]  x[3,3,blue]

julia> x[2, 1, "red"]
x[2,1,red]
```

Sets can depend upon previous indices:

```
julia> model = Model();

julia> @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]
```

and we can filter elements in the sets using the ; syntax:

```
julia> model = Model();

julia> @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]
```

11.2 Registered variables

When you create variables, JuMP registers them inside the model using their corresponding symbol. Get a registered name using model[:key]:

```
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> model
A JuMP Model
Feasibility problem with:
Variable: 1
Model mode: AUTOMATIC
CachingOptimizer state: NO_OPTIMIZER
Solver name: No optimizer attached.
Names registered in the model: x

julia> model[:x] === x
true
```

Registered names are most useful when you start to write larger models and want to break up the model construction into functions:

```
julia> function set_objective(model::Model)
           @objective(model, Min, 2 * model[:my_x] + 1)
           return
       end
set_objective (generic function with 1 method)

julia> model = Model();

julia> @variable(model, my_x);

julia> set_objective(model)

julia> print(model)
Min 2 my_x + 1
Subject to
```

11.3 Anonymous variables

To reduce the likelihood of accidental bugs, and because JuMP registers variables inside a model, creating two variables with the same name is an error:

```
julia> model = Model();

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]`.
[...]
```

A common reason for encountering this error is adding variables in a loop.

As a work-around, JuMP provides anonymous variables. Create a scalar valued anonymous variable by omitting the name argument:

```
julia> model = Model();

julia> x = @variable(model)
_[1]
```

Anonymous variables get printed as an underscore followed by a unique index of the variable.

> **Warning**
>
> The index of the variable may not correspond to the column of the variable in the solver.

Create a container of anonymous JuMP variables by dropping the name in front of the [:

```julia
julia> model = Model();

julia> y = @variable(model, [1:2])
2-element Vector{VariableRef}:
 _[1]
 _[2]
```

The <= and >= short-hand cannot be used to set bounds on scalar-valued anonymous JuMP variables. Instead, use the lower_bound and upper_bound keywords:

```julia
julia> model = Model();

julia> x_lower = @variable(model, lower_bound = 1.0)
_[1]

julia> x_upper = @variable(model, upper_bound = 2.0)
_[2]

julia> x_interval = @variable(model, lower_bound = 3.0, upper_bound = 4.0)
_[3]
```

11.4 Variable names

In addition to the symbol that variables are registered with, JuMP variables have a String name that is used for printing and writing to file formats.

Get and set the name of a variable using name and set_name:

```julia
julia> model = Model();

julia> @variable(model, x)
x

julia> name(x)
"x"

julia> set_name(x, "my_x_name")

julia> x
my_x_name
```

Override the default choice of name using the base_name keyword:

```julia
julia> model = Model();

julia> @variable(model, x[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
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]
```

> **Tip**
>
> For some models, setting the string name of each variable can take a non-trivial portion of the total time required to build the model. Turn off `String` names by passing `set_string_name = false` to `@variable`:
>
> ```julia
> julia> model = Model();
>
> julia> @variable(model, x, set_string_name = false)
> _[1]
> ```
>
> See Disable string names for more information.

Retrieve a variable by name

Retrieve a variable from a model using variable_by_name:

```julia
julia> variable_by_name(model, "my_x")
my_x
```

If the name is not present, nothing will be returned:

```julia
julia> variable_by_name(model, "bad_name")
```

You can only look up individual variables using variable_by_name. Something like this will not work:

```julia
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
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]
```

11.5   String names, symbolic names, and bindings

It's common for new users to experience confusion relating to JuMP variables. Part of the problem is the overloaded use of "variable" in mathematical optimization, along with the difference between the name that a variable is registered under and the `String` name used for printing.

Here's a summary of the differences:

- JuMP variables are created using `@variable`.

- JuMP variables can be named or anonymous.

- Named JuMP variables have the form `@variable(model, x)`. For named variables:

    - The `String` name of the variable is set to `"x"`.
    - A Julia variable `x` is created that binds `x` to the JuMP variable.
    - The name `:x` is registered as a key in the model with the value `x`.

- Anonymous JuMP variables have the form `x = @variable(model)`. For anonymous variables:

    - The `String` name of the variable is set to `""`. When printed, this is replaced with `"_[i]"` where `i` is the index of the variable.
    - You control the name of the Julia variable used as the binding.
    - No name is registered as a key in the model.

- The `base_name` keyword can override the `String` name of the variable.

- You can manually register names in the model via `model[:key] = value`

Here's an example that should make things clearer:

```julia
julia> model = Model();

julia> x_binding = @variable(model, base_name = "x")
x

julia> model
A JuMP Model
Feasibility problem with:
Variable: 1
```

```
Model mode: AUTOMATIC
CachingOptimizer state: NO_OPTIMIZER
Solver name: No optimizer attached.

julia> x
ERROR: UndefVarError: x not defined

julia> x_binding
x

julia> name(x_binding)
"x"

julia> model[:x_register] = x_binding
x

julia> model
A JuMP Model
Feasibility problem with:
Variable: 1
Model mode: AUTOMATIC
CachingOptimizer state: NO_OPTIMIZER
Solver name: No optimizer attached.
Names registered in the model: x_register

julia> model[:x_register]
x

julia> model[:x_register] === x_binding
true

julia> x
ERROR: UndefVarError: x not defined
```

11.6   Create, delete, and modify variable bounds

Query whether a variable has a bound using has_lower_bound, has_upper_bound, and is_fixed:

```
julia> has_lower_bound(x_free)
false

julia> has_upper_bound(x_upper)
true

julia> is_fixed(x_fixed)
true
```

If a variable has a particular bound, query the value of it using lower_bound, upper_bound, and fix_value:

```
julia> lower_bound(x_interval)
2.0

julia> upper_bound(x_interval)
```

```
3.0

julia> fix_value(x_fixed)
4.0
```

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

Delete variable bounds using delete_lower_bound, delete_upper_bound, and unfix:

```
julia> delete_lower_bound(x_lower)

julia> has_lower_bound(x_lower)
false

julia> delete_upper_bound(x_upper)

julia> has_upper_bound(x_upper)
false

julia> unfix(x_fixed)

julia> is_fixed(x_fixed)
false
```

Set or update variable bounds using set_lower_bound, set_upper_bound, and fix:

```
julia> set_lower_bound(x_lower, 1.1)

julia> set_upper_bound(x_upper, 2.1)

julia> fix(x_fixed, 4.1)
```

Fixing a variable with existing bounds will throw an error. To delete the bounds prior to fixing, use fix(variable, value; force = true).

```
julia> model = Model();

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.

11.7 Binary variables

Binary variables are constrained to the set $x \in \{0, 1\}$.

Create a binary variable by passing `Bin` as an optional positional argument:

```julia
julia> model = Model();

julia> @variable(model, x, Bin)
x
```

Check if a variable is binary using `is_binary`:

```julia
julia> is_binary(x)
true
```

Delete a binary constraint using `unset_binary`:

```julia
julia> unset_binary(x)

julia> is_binary(x)
false
```

Binary variables can also be created by setting the `binary` keyword to `true`:

```julia
julia> model = Model();

julia> @variable(model, x, binary=true)
x
```

or by using `set_binary`:

```julia
julia> model = Model();

julia> @variable(model, x)
x

julia> set_binary(x)
```

11.8 Integer variables

Integer variables are constrained to the set $x \in \mathbb{Z}$.

Create an integer variable by passing `Int` as an optional positional argument:

```
julia> model = Model();

julia> @variable(model, x, Int)
x
```

Check if a variable is integer using `is_integer`:

```
julia> is_integer(x)
true
```

Delete an integer constraint using `unset_integer`.

```
julia> unset_integer(x)

julia> is_integer(x)
false
```

Integer variables can also be created by setting the `integer` keyword to `true`:

```
julia> model = Model();

julia> @variable(model, x, integer=true)
x
```

or by using `set_integer`:

```
julia> model = Model();

julia> @variable(model, x)
x

julia> set_integer(x)
```

> **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.

11.9 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> model = Model();

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
```

The start keyword argument can depend on the indices of a variable container:

```
julia> model = Model();

julia> @variable(model, z[i = 1:2], start = i^2)
2-element Vector{VariableRef}:
 z[1]
 z[2]

julia> start_value.(z)
2-element Vector{Float64}:
 1.0
 4.0
```

> **Warning**
>
> Some solvers do not support start values.   If a solver does not support start values, an
> MathOptInterface.UnsupportedAttribute{MathOptInterface.VariablePrimalStart} error will
> be thrown.

> **Tip**
>
> To set the optimal solution from a previous solve as a new starting value, use all_variables to get a
> vector of all the variables in the model, then run:
>
> ```
> x = all_variables(model)
> x_solution = value.(x)
> set_start_value.(x, x_solution)
> ```
>
> Alternatively, use set_start_values.

11.10   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> model = Model();

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 corresponding name 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.

11.11   Variable containers

JuMP provides a mechanism for creating collections of variables in three types of data structures, which we refer to as containers.

The three types are `Arrays`, `DenseAxisArrays`, and `SparseAxisArrays`. 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 be extended to create multi-dimensional arrays of JuMP variables. For example:

```julia
julia> model = Model();

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
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
julia> model = Model();

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 (for example, 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> model = Model();

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]
```

Bounds can depend upon indices:

```
julia> model = Model();

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
julia> model = Model();

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
julia> model = Model();

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]
```

Performance considerations

When using the semi-colon as a filter, JuMP iterates over all indices and evaluates the conditional for each combination. If there are many index dimensions and a large amount of sparsity, this can be inefficient.

For example:

```julia
julia> model = Model();

julia> N = 10
10

julia> S = [(1, 1, 1), (N, N, N)]
2-element Vector{Tuple{Int64, Int64, Int64}}:
 (1, 1, 1)
 (10, 10, 10)

julia> @time @variable(model, x1[i=1:N, j=1:N, k=1:N; (i, j, k) in S])
  0.203861 seconds (392.22 k allocations: 23.977 MiB, 99.10% compilation time)
JuMP.Containers.SparseAxisArray{VariableRef, 3, Tuple{Int64, Int64, Int64}} with 2 entries:
  [1, 1, 1   ]  =  x1[1,1,1]
  [10, 10, 10]  =  x1[10,10,10]

julia> @time @variable(model, x2[S])
  0.045407 seconds (65.24 k allocations: 3.771 MiB, 99.15% compilation time)
1-dimensional DenseAxisArray{VariableRef,1,...} with index sets:
    Dimension 1, [(1, 1, 1), (10, 10, 10)]
And data, a 2-element Vector{VariableRef}:
 x2[(1, 1, 1)]
 x2[(10, 10, 10)]
```

The first option is slower because it is equivalent to:

```julia
julia> model = Model();

julia> x1 = Dict{NTuple{3,Int},VariableRef}()
Dict{Tuple{Int64, Int64, Int64}, VariableRef}()

julia> for i in 1:N
           for j in 1:N
               for k in 1:N
                   if (i, j, k) in S
                       x1[i, j, k] = @variable(model, base_name = "x1[$i,$j,$k]")
                   end
               end
           end
       end

julia> x1
Dict{Tuple{Int64, Int64, Int64}, VariableRef} with 2 entries:
  (1, 1, 1)    => x1[1,1,1]
  (10, 10, 10) => x1[10,10,10]
```

If performance is a concern, explicitly construct the set of indices instead of using the filtering syntax.

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, JuMP does not always make the best choice. To illustrate this, consider the following example:

```julia
julia> model = Model();

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
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. Choosing an invalid container type will throw an error.

User-defined containers

In addition to the built-in container types, you can create your own collections of JuMP variables.

> **Tip**
>
> This is a point that users often overlook: you are not restricted to the built-in container types in JuMP.

For example, the following code creates a dictionary with symmetric matrices as the values:

```julia
julia> model = Model();

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:

```julia
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:

```julia
julia> model = Model();

julia> x = model[:x] = @variable(model, [1:2], lower_bound = 0, base_name = "x")
2-element Vector{VariableRef}:
 x[1]
 x[2]

julia> append!(x, @variable(model, [1:2], lower_bound = 0, base_name = "y"));

julia> model[:x]
4-element Vector{VariableRef}:
 x[1]
 x[2]
 y[1]
 y[2]
```

11.12 Semidefinite variables

Declare a square matrix of JuMP variables to be positive semidefinite by passing PSD as an optional positional argument:

```julia
julia> model = Model();

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]
```

This will ensure that x is symmetric, and that all of its eigenvalues are nonnegative.

> **Note**
>
> x must be a square 2-dimensional Array of JuMP variables; it cannot be a DenseAxisArray or a SparseAxisArray.

11.13 Symmetric variables

Declare a square matrix of JuMP variables to be symmetric (but not necessarily positive semidefinite) by passing Symmetric as an optional positional argument:

```julia
julia> model = Model();

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]
```

11.14 The @variables macro

If you have many @variable calls, JuMP provides the macro @variables that can improve readability:

```julia
julia> model = Model();

julia> @variables(model, begin
           x
           y[i=1:2] >= i, (start = i, base_name = "Y_$i")
           z, Bin
       end)
(x, VariableRef[Y_1[1], Y_2[2]], z)

julia> print(model)
Feasibility
Subject to
 Y_1[1] ≥ 1
 Y_2[2] ≥ 2
 z binary
```

The @variables macro returns a tuple of the variables that were defined.

> **Note**
>
> Keyword arguments must be contained within parentheses.

11.15   Variables constrained on creation

All uses of the @variable macro documented so far translate into separate calls for variable creation and the adding of any bound or integrality constraints.

For example, @variable(model, x >= 0, Int), is equivalent to:

```julia
julia> model = Model();

julia> @variable(model, x)
x

julia> set_lower_bound(x, 0.0)

julia> set_integer(x)
```

Importantly, the bound and integrality constraints are added after the variable has been created.

However, some solvers require a set specifying the variable domain to be given when the variable is first created. 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 that belong to the SecondOrderCone:

```julia
julia> model = Model();

julia> @variable(model, y[1:3] in SecondOrderCone())
3-element Vector{VariableRef}:
 y[1]
 y[2]
 y[3]
```

For contrast, the standard syntax is as follows:

```julia
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)
```

An alternate syntax to x in Set is to use the set keyword of @variable. This is most useful when creating anonymous variables:

```
julia> model = Model();

julia> x = @variable(model, [1:3], set = SecondOrderCone())
3-element Vector{VariableRef}:
 _[1]
 _[2]
 _[3]
```

> **Note**
>
> You cannot delete the constraint associated with a variable constrained on creation.

Example: positive semidefinite variables

An alternative to the syntax in Semidefinite variables, declare a matrix of JuMP variables to be positive semidefinite using PSDCone:

```
julia> model = Model();

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]
```

Example: symmetric variables

As an alternative to the syntax in Symmetric variables, declare a matrix of JuMP variables to be symmetric using SymmetricMatrixSpace:

```
julia> model = Model();

julia> @variable(model, x[1:2, 1:2] in SymmetricMatrixSpace())
2×2 LinearAlgebra.Symmetric{VariableRef, Matrix{VariableRef}}:
 x[1,1]  x[1,2]
 x[1,2]  x[2,2]
```

Example: skew-symmetric variables

Declare a matrix of JuMP variables to be skew-symmetric using SkewSymmetricMatrixSpace:

```
julia> model = Model();

julia> @variable(model, x[1:2, 1:2] in SkewSymmetricMatrixSpace())
2×2 Matrix{AffExpr}:
 0        x[1,2]
 -x[1,2]  0
```

> **Note**
>
> Even though x is a 2 by 2 matrix, only one decision variable is added to model; the remaining elements in x are linear transformations of the single variable.

Because the returned matrix x is `Matrix{AffExpr}`, you cannot use variable-related functions on its elements:

```julia
julia> set_lower_bound(x[1, 2], 0.0)
ERROR: MethodError: no method matching set_lower_bound(::AffExpr, ::Float64)
[...]
```

However, you can convert the matrix into one in which the upper triangular elements are `VariableRef` and the lower triangular elements are `AffExpr` as follows:

```julia
julia> y = Union{VariableRef,AffExpr}[
           j > i ? first(keys(x[i, j].terms)) : x[i, j]
           for i in 1:size(x, 1), j in 1:size(x, 2)
       ]
2×2 Matrix{Union{AffExpr, VariableRef}}:
 0        x[1,2]
 -x[1,2]  0

julia> set_lower_bound(y[1, 2], 0.0)
```

Example: Hermitian positive semidefinite variables

Declare a matrix of JuMP variables to be Hermitian positive semidefinite using HermitianPSDCone:

```julia
julia> model = Model();

julia> @variable(model, H[1:2, 1:2] in HermitianPSDCone())
2×2 LinearAlgebra.Hermitian{GenericAffExpr{ComplexF64, VariableRef},
↪   Matrix{GenericAffExpr{ComplexF64, VariableRef}}}:
 real(H[1,1])                    real(H[1,2]) + imag(H[1,2]) im
 real(H[1,2]) - imag(H[1,2]) im  real(H[2,2])
```

This adds 4 real variables in the MOI.HermitianPositiveSemidefiniteConeTriangle:

```julia
julia> first(all_constraints(model, Vector{VariableRef},
↪   MOI.HermitianPositiveSemidefiniteConeTriangle))
[real(H[1,1]), real(H[1,2]), real(H[2,2]), imag(H[1,2])] ∈
↪   MathOptInterface.HermitianPositiveSemidefiniteConeTriangle(2)
```

Example: Hermitian variables

Declare a matrix of JuMP variables to be Hermitian using the `Hermitian` tag:

```julia
julia> model = Model();

julia> @variable(model, x[1:2, 1:2], Hermitian)
```

```
2×2 LinearAlgebra.Hermitian{GenericAffExpr{ComplexF64, VariableRef},
↪  Matrix{GenericAffExpr{ComplexF64, VariableRef}}}:
 real(x[1,1])                   real(x[1,2]) + imag(x[1,2]) im
 real(x[1,2]) - imag(x[1,2]) im  real(x[2,2])
```

This is equivalent to declaring the variable in HermitianMatrixSpace:

```
julia> model = Model();

julia> @variable(model, x[1:2, 1:2] in HermitianMatrixSpace())
2×2 LinearAlgebra.Hermitian{GenericAffExpr{ComplexF64, VariableRef},
↪  Matrix{GenericAffExpr{ComplexF64, VariableRef}}}:
 real(x[1,1])                   real(x[1,2]) + imag(x[1,2]) im
 real(x[1,2]) - imag(x[1,2]) im  real(x[2,2])
```

Why use variables constrained on creation?

For most users, it does not matter if you use the constrained on creation syntax. Therefore, use whatever syntax you find most convenient.

However, if you use direct_model, you may be forced to use the constrained on creation syntax.

The technical difference between variables constrained on creation and the standard JuMP syntax is that variables constrained on creation calls `MOI.add_constrained_variables`, while the standard JuMP syntax calls `MOI.add_variables` and then `MOI.add_constraint`.

Consult the implementation of solver package you are using to see if your solver requires `MOI.add_constrained_variables`.

Chapter 12

Constraints

JuMP is based on the MathOptInterface (MOI) API. Because of this, JuMP uses the following standard form to represent problems:

$$\min_{x \in \mathbb{R}^n} \quad f_0(x) \tag{12.1}$$

$$\text{s.t.} \quad f_i(x) \in \mathcal{S}_i \qquad i = 1 \dots m \tag{12.2}$$

Each constraint, $f_i(x) \in \mathcal{S}_i$, is composed of a function and a set. For example, instead of calling $a^\top x \le b$ a less-than-or-equal-to constraint, we say that it is a scalar-affine-in-less-than constraint, where the function $a^\top x$ belongs to the less-than set $(-\infty, b]$. We use the shorthand function-in-set to refer to constraints composed of different types of functions and sets.

This page explains how to write various types of constraints in JuMP. For nonlinear constraints, see Nonlinear Modeling instead.

12.1 Add a constraint

Add a constraint to a JuMP model using the @constraint macro. The syntax to use depends on the type of constraint you wish to add.

Add a linear constraint

Create linear constraints using the @constraint macro:

```julia
julia> model = Model();

julia> @variable(model, x[1:3]);

julia> @constraint(model, c1, sum(x) <= 1)
c1 : x[1] + x[2] + x[3] ≤ 1

julia> @constraint(model, c2, x[1] + 2 * x[3] >= 2)
c2 : x[1] + 2 x[3] ≥ 2

julia> @constraint(model, c3, sum(i * x[i] for i in 1:3) == 3)
c3 : x[1] + 2 x[2] + 3 x[3] = 3
```

```
julia> @constraint(model, c4, 4 <= 2 * x[2] <= 5)
c4 : 2 x[2] ∈ [4, 5]
```

Normalization

JuMP normalizes 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> model = Model();

julia> @variable(model, x);

julia> @constraint(model, c, 2x + 1 <= 4x + 4)
c : -2 x ≤ 3
```

Add a quadratic constraint

In addition to affine functions, JuMP also supports constraints with quadratic terms. For example:

```
julia> model = Model();

julia> @variable(model, x[i=1:2])
2-element Vector{VariableRef}:
 x[1]
 x[2]

julia> @variable(model, t >= 0)
t

julia> @constraint(model, my_q, x[1]^2 + x[2]^2 <= t^2)
my_q : x[1]² + x[2]² - t² ≤ 0
```

> **Tip**
>
> Because solvers can take advantage of the knowledge that a constraint is quadratic, prefer adding quadratic constraints using @constraint, rather than @NLconstraint.

12.2   Vectorized constraints

You can also add constraints to JuMP using vectorized linear algebra. For example:

```
julia> model = Model();

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_vector, A * x == b)
con_vector : [x[1] + 2 x[2] - 5, 3 x[1] + 4 x[2] - 6] ∈ MathOptInterface.Zeros(2)

julia> @constraint(model, con_scalar, A * x .== b)
2-element Vector{ConstraintRef{Model,
↪   MathOptInterface.ConstraintIndex{MathOptInterface.ScalarAffineFunction{Float64},
↪   MathOptInterface.EqualTo{Float64}}, ScalarShape}}:
 con_scalar : x[1] + 2 x[2] = 5
 con_scalar : 3 x[1] + 4 x[2] = 6
```

The two constraints, == and .== are similar, but subtly different. The first creates a single constraint that is a MOI.VectorAffineFunction in MOI.Zeros constraint. The second creates a vector of MOI.ScalarAffineFunction in MOI.EqualTo constraints.

Which formulation to choose depends on the solver, and what you want to do with the constraint object con_vector or con_scalar.

- If you are using a conic solver, expect the dual of con_vector to be a Vector{Float64}, and do not intend to delete a row in the constraint, choose the == formulation.

- If you are using a solver that expects a list of scalar constraints, for example HiGHS, or you wish to delete part of the constraint or access a single row of the constraint, for example, dual(con_scalar[2]), then use the broadcast .==.

JuMP reformulates both constraints into the other form if needed by the solver, but choosing the right format for a particular solver is more efficient.

You can also use <=, .<= , >=, and .>= as comparison operators in the constraint.

```
julia> @constraint(model, A * x <= b)
[x[1] + 2 x[2] - 5, 3 x[1] + 4 x[2] - 6] ∈ MathOptInterface.Nonpositives(2)

julia> @constraint(model, A * x .<= b)
2-element Vector{ConstraintRef{Model,
↪   MathOptInterface.ConstraintIndex{MathOptInterface.ScalarAffineFunction{Float64},
↪   MathOptInterface.LessThan{Float64}}, ScalarShape}}:
 x[1] + 2 x[2] ≤ 5
 3 x[1] + 4 x[2] ≤ 6

julia> @constraint(model, A * x >= b)
[x[1] + 2 x[2] - 5, 3 x[1] + 4 x[2] - 6] ∈ MathOptInterface.Nonnegatives(2)

julia> @constraint(model, A * x .>= b)
2-element Vector{ConstraintRef{Model,
↪   MathOptInterface.ConstraintIndex{MathOptInterface.ScalarAffineFunction{Float64},
↪   MathOptInterface.GreaterThan{Float64}}, ScalarShape}}:
```

```
 x[1] + 2 x[2] ≥ 5
 3 x[1] + 4 x[2] ≥ 6
```

Vectorized matrix constraints

In most cases, you cannot use the non-broadcasting syntax for general matrices. For example:

```julia
julia> model = Model();

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)
ERROR: At none:1: `@constraint(model, X >= 0)`: Unsupported matrix in vector-valued set. Did you
↪   mean to use the broadcasting syntax `.>=` instead? Alternatively, perhaps you are missing a set
↪   argument like `@constraint(model, X >= 0, PSDCone())` or `@constraint(model, X >= 0,
↪   HermmitianPSDCone())`.
Stacktrace:
[...]
```

Instead, to represent matrix inequalities you must always use the element-wise broadcasting .==, .>=, or .<=, or use the [Set inequality syntax](.).

There are two exceptions: if the result of the left-hand side minus the right-hand side is a `LinearAlgebra.Symmetric` matrix or a `LinearAlgebra.Hermitian` matrix, you may use the non-broadcasting equality syntax:

```julia
julia> using LinearAlgebra

julia> model = Model();

julia> @variable(model, X[1:2, 1:2], Symmetric)
2×2 Symmetric{VariableRef, Matrix{VariableRef}}:
 X[1,1]  X[1,2]
 X[1,2]  X[2,2]

julia> @constraint(model, X == LinearAlgebra.I)
[X[1,1] - 1  X[1,2];
 X[1,2]      X[2,2] - 1] ∈ Zeros()
```

Despite the model showing the matrix in [Zeros](.), this will add only three rows to the constraint matrix because the symmetric constraints are redundant. In contrast, the broadcasting syntax adds four linear constraints:

```julia
julia> @constraint(model, X .== LinearAlgebra.I)
2×2 Matrix{ConstraintRef{Model,
↪   MathOptInterface.ConstraintIndex{MathOptInterface.ScalarAffineFunction{Float64},
↪   MathOptInterface.EqualTo{Float64}}, ScalarShape}}:
 X[1,1] = 1  X[1,2] = 0
 X[1,2] = 0  X[2,2] = 1
```

The same holds for `LinearAlgebra.Hermitian` matrices:

```
julia> using LinearAlgebra

julia> model = Model();

julia> @variable(model, X[1:2, 1:2] in HermitianPSDCone())
2×2 Hermitian{GenericAffExpr{ComplexF64, VariableRef}, Matrix{GenericAffExpr{ComplexF64,
↪   VariableRef}}}:
 real(X[1,1])                    real(X[1,2]) + imag(X[1,2]) im
 real(X[1,2]) - imag(X[1,2]) im  real(X[2,2])

julia> @constraint(model, X == LinearAlgebra.I)
[real(X[1,1]) - 1                    real(X[1,2]) + imag(X[1,2]) im;
 real(X[1,2]) - imag(X[1,2]) im  real(X[2,2]) - 1] ∈ Zeros()

julia> @constraint(model, X .== LinearAlgebra.I)
2×2 Matrix{ConstraintRef{Model,
↪   MathOptInterface.ConstraintIndex{MathOptInterface.ScalarAffineFunction{ComplexF64},
↪   MathOptInterface.EqualTo{ComplexF64}}, ScalarShape}}:
 real(X[1,1]) = 1                    real(X[1,2]) + imag(X[1,2]) im = 0
 real(X[1,2]) - imag(X[1,2]) im = 0  real(X[2,2]) = 1
```

12.3 Containers of constraints

The @constraint macro supports creating collections of constraints. We'll cover some brief syntax here; read
the Constraint containers section for more details:

Create arrays of constraints:

```
julia> model = Model();

julia> @variable(model, x[1:3]);

julia> @constraint(model, c[i=1:3], x[i] <= i^2)
3-element Vector{ConstraintRef{Model,
↪   MathOptInterface.ConstraintIndex{MathOptInterface.ScalarAffineFunction{Float64},
↪   MathOptInterface.LessThan{Float64}}, ScalarShape}}:
 c[1] : x[1] ≤ 1
 c[2] : x[2] ≤ 4
 c[3] : x[3] ≤ 9

julia> c[2]
c[2] : x[2] ≤ 4
```

Sets can be any Julia type that supports iteration:

```
julia> model = Model();

julia> @variable(model, x[1:3]);

julia> @constraint(model, c[i=2:3, ["red", "blue"]], x[i] <= i^2)
2-dimensional DenseAxisArray{ConstraintRef{Model,
↪   MathOptInterface.ConstraintIndex{MathOptInterface.ScalarAffineFunction{Float64},
↪   MathOptInterface.LessThan{Float64}}, ScalarShape},2,...} with index sets:
```

```
    Dimension 1, 2:3
    Dimension 2, ["red", "blue"]
And data, a 2×2 Matrix{ConstraintRef{Model,
↪   MathOptInterface.ConstraintIndex{MathOptInterface.ScalarAffineFunction{Float64},
↪   MathOptInterface.LessThan{Float64}}, ScalarShape}}:
 c[2,red] : x[2] ≤ 4  c[2,blue] : x[2] ≤ 4
 c[3,red] : x[3] ≤ 9  c[3,blue] : x[3] ≤ 9

julia> c[2, "red"]
c[2,red] : x[2] ≤ 4
```

Sets can depend upon previous indices:

```
julia> model = Model();

julia> @variable(model, x[1:3]);

julia> @constraint(model, c[i=1:3, j=i:3], x[i] <= j)
JuMP.Containers.SparseAxisArray{ConstraintRef{Model,
↪   MathOptInterface.ConstraintIndex{MathOptInterface.ScalarAffineFunction{Float64},
↪   MathOptInterface.LessThan{Float64}}, ScalarShape}, 2, Tuple{Int64, Int64}} with 6 entries:
  [1, 1]  =  c[1,1] : x[1] ≤ 1
  [1, 2]  =  c[1,2] : x[1] ≤ 2
  [1, 3]  =  c[1,3] : x[1] ≤ 3
  [2, 2]  =  c[2,2] : x[2] ≤ 2
  [2, 3]  =  c[2,3] : x[2] ≤ 3
  [3, 3]  =  c[3,3] : x[3] ≤ 3
```

and you can filter elements in the sets using the ; syntax:

```
julia> model = Model();

julia> @variable(model, x[1:9]);

julia> @constraint(model, c[i=1:9; mod(i, 3) == 0], x[i] <= i)
JuMP.Containers.SparseAxisArray{ConstraintRef{Model,
↪   MathOptInterface.ConstraintIndex{MathOptInterface.ScalarAffineFunction{Float64},
↪   MathOptInterface.LessThan{Float64}}, ScalarShape}, 1, Tuple{Int64}} with 3 entries:
  [3]  =  c[3] : x[3] ≤ 3
  [6]  =  c[6] : x[6] ≤ 6
  [9]  =  c[9] : x[9] ≤ 9
```

12.4 Registered constraints

When you create constraints, JuMP registers them inside the model using their corresponding symbol. Get a registered name using model[:key]:

```
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, my_c, 2x <= 1)
my_c : 2 x ≤ 1

julia> model
A JuMP Model
Feasibility problem with:
Variable: 1
`AffExpr`-in-`MathOptInterface.LessThan{Float64}`: 1 constraint
Model mode: AUTOMATIC
CachingOptimizer state: NO_OPTIMIZER
Solver name: No optimizer attached.
Names registered in the model: my_c, x

julia> model[:my_c] === my_c
true
```

12.5 Anonymous constraints

To reduce the likelihood of accidental bugs, and because JuMP registers constraints inside a model, creating two constraints with the same name is an error:

```
julia> model = Model();

julia> @variable(model, x)
x

julia> @constraint(model, c, 2x <= 1)
c : 2 x ≤ 1

julia> @constraint(model, c, 2x <= 1)
ERROR: An object of name c 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, :c)` 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[:c]`.
[...]
```

A common reason for encountering this error is adding constraints in a loop.

As a work-around, JuMP provides anonymous constraints. Create an anonymous constraint by omitting the name argument:

```
julia> model = Model();

julia> @variable(model, x);

julia> c = @constraint(model, 2x <= 1)
2 x ≤ 1
```

Create a container of anonymous constraints by dropping the name in front of the [:

```
julia> model = Model();

julia> @variable(model, x[1:3]);

julia> c = @constraint(model, [i = 1:3], x[i] <= i)
3-element Vector{ConstraintRef{Model,
↪  MathOptInterface.ConstraintIndex{MathOptInterface.ScalarAffineFunction{Float64},
↪  MathOptInterface.LessThan{Float64}}, ScalarShape}}:
 x[1] ≤ 1
 x[2] ≤ 2
 x[3] ≤ 3
```

12.6   Constraint names

In addition to the symbol that constraints are registered with, constraints have a String name that is used for printing and writing to file formats.

Get and set the name of a constraint using name(::JuMP.ConstraintRef) and set_name(::JuMP.ConstraintRef, ::String):

```
julia> model = Model(); @variable(model, x);

julia> @constraint(model, con, x <= 1)
con : x ≤ 1

julia> name(con)
"con"

julia> set_name(con, "my_con_name")

julia> con
my_con_name : x ≤ 1
```

Override the default choice of name using the base_name keyword:

```
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
 my_con[2] : x ≤ 2
```

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
 my_con[2] : x ≤ 2
```

> **Tip**
>
> For some models, setting the string name of each constraint can take a non-trivial portion of the total time required to build the model. Turn off String names by passing set_string_name = false to @constraint:
>
> ```
> julia> model = Model();
>
> julia> @variable(model, x);
>
> julia> @constraint(model, con, x <= 2, set_string_name = false)
> x ≤ 2
> ```
>
> See Disable string names for more information.

Retrieve a constraint by name

Retrieve a constraint from a model using constraint_by_name:

```
julia> constraint_by_name(model, "c")
c : x ≤ 1
```

If the name is not present, nothing will be returned:

```
julia> constraint_by_name(model, "bad_name")
```

You can only look up individual 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
```

```
 my_con[2] : x ≤ 2

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
 my_con[2] : x ≤ 2

julia> model[:con]
2-element Vector{ConstraintRef{Model,
↪   MathOptInterface.ConstraintIndex{MathOptInterface.ScalarAffineFunction{Float64},
↪   MathOptInterface.LessThan{Float64}}, ScalarShape}}:
 my_con[1] : x ≤ 1
 my_con[2] : x ≤ 2
```

12.7   String names, symbolic names, and bindings

It's common for new users to experience confusion relating to constraints. Part of the problem is the difference between the name that a constraint is registered under and the String name used for printing.

Here's a summary of the differences:

- Constraints are created using @constraint.

- Constraints can be named or anonymous.

- Named constraints have the form @constraint(model, c, expr). For named constraints:

    - The String name of the constraint is set to "c".

    - A Julia variable c is created that binds c to the JuMP constraint.

    - The name :c is registered as a key in the model with the value c.

- Anonymous constraints have the form c = @constraint(model, expr). For anonymous constraints:

    - The String name of the constraint is set to "".

    - You control the name of the Julia variable used as the binding.

    - No name is registered as a key in the model.

- The base_name keyword can override the String name of the constraint.

- You can manually register names in the model via model[:key] = value.

Here's an example of the differences:

```
julia> model = Model();

julia> @variable(model, x)
x

julia> c_binding = @constraint(model, 2x <= 1, base_name = "c")
c : 2 x ≤ 1

julia> model
A JuMP Model
Feasibility problem with:
Variable: 1
`AffExpr`-in-`MathOptInterface.LessThan{Float64}`: 1 constraint
Model mode: AUTOMATIC
CachingOptimizer state: NO_OPTIMIZER
Solver name: No optimizer attached.
Names registered in the model: x

julia> c
ERROR: UndefVarError: c not defined

julia> c_binding
c : 2 x ≤ 1

julia> name(c_binding)
"c"

julia> model[:c_register] = c_binding
c : 2 x ≤ 1

julia> model
A JuMP Model
Feasibility problem with:
Variable: 1
`AffExpr`-in-`MathOptInterface.LessThan{Float64}`: 1 constraint
Model mode: AUTOMATIC
CachingOptimizer state: NO_OPTIMIZER
Solver name: No optimizer attached.
Names registered in the model: c_register, x

julia> model[:c_register]
c : 2 x ≤ 1

julia> model[:c_register] === c_binding
true

julia> c
ERROR: UndefVarError: c not defined
```

12.8 The @constraints macro

If you have many @constraint calls, use the @constraints macro to improve readability:

```julia
julia> model = Model();

julia> @variable(model, x);

julia> @constraints(model, begin
           2x <= 1
           c, x >= -1
       end)
(2 x ≤ 1, c : x ≥ -1)

julia> print(model)
Feasibility
Subject to
 c : x ≥ -1
 2 x ≤ 1
```

The @constraints macro returns a tuple of the constraints that were defined.

12.9 Duality

JuMP adopts the notion of conic duality from MathOptInterface. 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.

> **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.

The dual value associated with a constraint in the most recent solution can be accessed using the dual function. Use has_duals to check if the model has a dual solution available to query. For example:

```julia
julia> model = Model(HiGHS.Optimizer);

julia> set_silent(model)

julia> @variable(model, x)
x

julia> @constraint(model, con, x <= 1)
con : x ≤ 1

julia> @objective(model, Min, -2x)
-2 x

julia> has_duals(model)
false

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 shadow_price, 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
```

Duals of variable bounds

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).

```
julia> model = Model(HiGHS.Optimizer);

julia> set_silent(model)

julia> @variable(model, x <= 1)
x

julia> @objective(model, Min, -2x)
-2 x

julia> optimize!(model)

julia> dual(UpperBoundRef(x))
-2.0

julia> reduced_cost(x)
-2.0
```

12.10   Modify a constant term

This section explains how to modify the constant term in a constraint. There are multiple ways to achieve this goal; we explain three options.

Option 1: change the right-hand side

Use set_normalized_rhs to modify the right-hand side (constant) term of a linear or quadratic constraint. Use normalized_rhs to query the right-hand side term.

```
julia> model = Model();

julia> @variable(model, x);

julia> @constraint(model, con, 2x <= 1)
con : 2 x ≤ 1

julia> set_normalized_rhs(con, 3)

julia> con
con : 2 x ≤ 3

julia> normalized_rhs(con)
3.0
```

> **Warning**
>
> set_normalized_rhs sets the right-hand side term of the normalized constraint. See Normalization for more details.

Option 2: use fixed variables

If constraints are complicated, for example, 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 is 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 new variable and fixing it to different values. Here is an example:

```
julia> model = Model();

julia> @variable(model, x);

julia> @variable(model, const_term)
const_term

julia> @constraint(model, con, 2x <= const_term + 1)
con : 2 x - const_term ≤ 1

julia> fix(const_term, 1.0)
```

The constraint con is now equivalent to 2x <= 2.

> **Warning**
>
> Fixed variables are not replaced with constants when communicating the problem to a solver. There-
> fore, even though `const_term` is fixed, it is still a decision variable, and so `const_term * x` is bilinear.

Option 3: modify the function's constant term

The third 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
julia> model = Model();

julia> @variable(model, x);

julia> @constraint(model, con, 2x <= 1)
con : 2 x ≤ 1

julia> add_to_function_constant(con, 2)

julia> con
con : 2 x ≤ -1

julia> normalized_rhs(con)
-1.0
```

In the case of interval constraints, the constant is removed from each bound:

```julia
julia> model = Model();

julia> @variable(model, x);

julia> @constraint(model, con, 0 <= 2x + 1 <= 2)
con : 2 x ∈ [-1, 1]

julia> add_to_function_constant(con, 3)

julia> con
con : 2 x ∈ [-4, -2]
```

12.11   Modify a variable coefficient

Scalar constraints

To modify the coefficients for a linear term (modifying the coefficient of a quadratic term is not supported) in a constraint, use `set_normalized_coefficient`. To query the current coefficient, use `normalized_coefficient`.

```julia
julia> model = Model();

julia> @variable(model, x[1:2]);
```

```
julia> @constraint(model, con, 2x[1] + x[2] <= 1)
con : 2 x[1] + x[2] ≤ 1

julia> set_normalized_coefficient(con, x[2], 0)

julia> con
con : 2 x[1] ≤ 1

julia> normalized_coefficient(con, x[2])
0.0
```

> **Warning**
>
> set_normalized_coefficient sets the coefficient of the normalized constraint. See Normalization for more details.

Vector constraints

To modify the coefficients of a vector-valued constraint, use set_normalized_coefficients.

```
julia> model = Model();

julia> @variable(model, x)
x

julia> @constraint(model, con, [2x + 3x, 4x] in MOI.Nonnegatives(2))
con : [5 x, 4 x] ∈ MathOptInterface.Nonnegatives(2)

julia> set_normalized_coefficients(con, x, [(1, 3.0)])

julia> con
con : [3 x, 4 x] ∈ MathOptInterface.Nonnegatives(2)

julia> set_normalized_coefficients(con, x, [(1, 2.0), (2, 5.0)])

julia> con
con : [2 x, 5 x] ∈ MathOptInterface.Nonnegatives(2)
```

12.12   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> model = Model();

julia> @variable(model, x);

julia> @constraint(model, con, 2x <= 1)
con : 2 x ≤ 1

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
```

> **Info**
>
> delete does not automatically unregister because we do not distinguish between names that are au-tomatically 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.

12.13   Start values

Provide a starting value (also called warmstart) for a constraint's primal and dual solutions using set_start_value and set_dual_start_value.

Query the starting value for a constraint's primal and dual solution using start_value and dual_start_value. If no start value has been set, the methods will return nothing.

```
julia> model = Model();

julia> @variable(model, x)
x

julia> @constraint(model, con, x >= 10)
con : x ≥ 10
```

```
julia> start_value(con)

julia> set_start_value(con, 10.0)

julia> start_value(con)
10.0

julia> dual_start_value(con)

julia> set_dual_start_value(con, 2)

julia> dual_start_value(con)
2.0
```

Vector-valued constraints require a vector:

```
julia> model = Model();

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
```

> **Tip**
>
> To simplify setting start values for all variables and constraints in a model, see set_start_values. The Primal and dual warm-starts tutorial also gives a detailed description of how to iterate over constraints in the model to set custom start values.

12.14 Constraint containers

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: Arrays, DenseAxisArrays, and SparseAxisArrays. 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
julia> model = Model();

julia> @variable(model, x);

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
 con[2] : 2 x ≤ 3
 con[3] : 3 x ≤ 4
```

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
julia> con[1]
con[1] : x ≤ 2

julia> con[2:3]
2-element Vector{ConstraintRef{Model,
↪   MathOptInterface.ConstraintIndex{MathOptInterface.ScalarAffineFunction{Float64},
↪   MathOptInterface.LessThan{Float64}}, ScalarShape}}:
 con[2] : 2 x ≤ 3
 con[3] : 3 x ≤ 4
```

Anonymous containers can also be constructed by dropping the name (for example, con) before the square brackets:

```julia
julia> con = @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
 2 x ≤ 3
```

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 (for example, 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> model = Model();

julia> @variable(model, x);

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    con[1,3] : x ≤ 4
 con[2,2] : 2 x ≤ 3  con[2,3] : 2 x ≤ 4
```

SparseAxisArrays

The syntax for constructing a SparseAxisArray of constraints is very similar to the syntax for constructing a
SparseAxisArray of variables.

```
julia> model = Model();

julia> @variable(model, x);

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
  [2, 1]  =  con[2,1] : 2 x ≤ 2
```

> **Warning**
>
> If you have many index dimensions and a large amount of sparsity, read Performance considerations.

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, you can force the type of container using the container keyword. For syntax and the
reason behind this, take a look at the variable docs.

Constraints with similar indices

Containers are often used to create constraints over a set of indices. However, you'll often have cases in which
you are repeating the indices:

```
julia> model = Model();

julia> @variable(model, x[1:2]);
```

```
julia> @variable(model, y[1:2]);

julia> @constraints(model, begin
           [i=1:2, j=1:2, k=1:2], i * x[j] <= k
           [i=1:2, j=1:2, k=1:2], i * y[j] <= k
       end);
```

This is hard to read and leads to a lot of copy-paste. A more readable way is to use a for-loop:

```
julia> for i=1:2, j=1:2, k=1:2
           @constraints(model, begin
               i * x[j] <= k
               i * y[j] <= k
           end)
       end
```

12.15 Accessing constraints from a model

Query the types of function-in-set constraints in a model using list_of_constraint_types:

```
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)
```

For a given combination of function and set type, use num_constraints to access the number of constraints
and all_constraints to access a list of their references:

```
julia> num_constraints(model, VariableRef, MOI.Integer)
2

julia> cons = all_constraints(model, VariableRef, MOI.Integer)
2-element Vector{ConstraintRef{Model,
↪   MathOptInterface.ConstraintIndex{MathOptInterface.VariableIndex, MathOptInterface.Integer},
↪   ScalarShape}}:
 x[1] integer
 x[2] integer
```

You can also count the total number of constraints in the model, but you must explicitly choose whether to
count VariableRef constraints such as bound and integrality constraints:

```
julia> num_constraints(model; count_variable_in_set_constraints = true)
5

julia> num_constraints(model; count_variable_in_set_constraints = false)
1
```

The same also applies for all_constraints:

```
julia> all_constraints(model; include_variable_in_set_constraints = true)
5-element Vector{ConstraintRef}:
 x[1] + x[2] ≤ 1
 x[1] ≥ 1
 x[2] ≥ 2
 x[1] integer
 x[2] integer

julia> all_constraints(model; include_variable_in_set_constraints = false)
1-element Vector{ConstraintRef}:
 x[1] + x[2] ≤ 1
```

If you need finer-grained control on which constraints to include, use a variant of:

```
julia> sum(
           num_constraints(model, F, S) for
           (F, S) in list_of_constraint_types(model) if F != VariableRef
       )
1
```

Use constraint_object to get an instance of an AbstractConstraint object that stores the constraint data:

```
julia> con = constraint_object(cons[1])
ScalarConstraint{VariableRef, MathOptInterface.Integer}(x[1], MathOptInterface.Integer())

julia> con.func
x[1]

julia> con.set
MathOptInterface.Integer()
```

12.16   MathOptInterface constraints

Because JuMP is based on MathOptInterface, you can add any constraints supported by MathOptInterface using the function-in-set syntax. For a list of supported functions and sets, read Standard form problem.

> **Note**
>
> We use MOI as an alias for the MathOptInterface module. This alias is defined by using JuMP. You may also define it in your code as follows:
>
> ```
> import MathOptInterface as MOI
> ```

For example, the following two constraints are equivalent:

```julia
julia> model = Model();

julia> @variable(model, x[1:3]);

julia> @constraint(model, 2 * x[1] <= 1)
2 x[1] ≤ 1

julia> @constraint(model, 2 * x[1] in MOI.LessThan(1.0))
2 x[1] ≤ 1
```

You can also use any set defined by MathOptInterface:

```julia
julia> @constraint(model, x - [1; 2; 3] in MOI.Nonnegatives(3))
[x[1] - 1, x[2] - 2, x[3] - 3] ∈ MathOptInterface.Nonnegatives(3)

julia> @constraint(model, x in MOI.ExponentialCone())
[x[1], x[2], x[3]] ∈ MathOptInterface.ExponentialCone()
```

> **Info**
>
> Similar to how JuMP defines the <= and >= syntax as a convenience way to specify `MOI.LessThan` and `MOI.GreaterThan` constraints, the remaining sections in this page describe functions and syntax that have been added for the convenience of common modeling situations.

12.17   Set inequality syntax

For modeling convenience, the syntax `@constraint(model, x >= y, Set())` is short-hand for `@constraint(model, x - y in Set())`. Therefore, the following calls are equivalent:

```julia
julia> model = Model();

julia> @variable(model, x[1:2]);

julia> y = [0.5, 0.75];

julia> @constraint(model, x >= y, MOI.Nonnegatives(2))
[x[1] - 0.5, x[2] - 0.75] ∈ MathOptInterface.Nonnegatives(2)

julia> @constraint(model, y <= x, MOI.Nonnegatives(2))
[x[1] - 0.5, x[2] - 0.75] ∈ MathOptInterface.Nonnegatives(2)

julia> @constraint(model, x - y in MOI.Nonnegatives(2))
[x[1] - 0.5, x[2] - 0.75] ∈ MathOptInterface.Nonnegatives(2)
```

Non-zero constants are not supported in this syntax:

```julia
julia> @constraint(model, x >= 1, MOI.Nonnegatives(2))
ERROR: Operation `sub_mul` between `Vector{VariableRef}` and `Int64` is not allowed. This most
↪  often happens when you write a constraint like `x >= y` where `x` is an array and `y` is a
↪  constant. Use the broadcast syntax `x .- y >= 0` instead.
```

```
Stacktrace:
[...]
```

Use instead:

```
julia> @constraint(model, x .- 1 >= 0, MOI.Nonnegatives(2))
[x[1] - 1, x[2] - 1] ∈ MathOptInterface.Nonnegatives(2)
```

12.18   Second-order cone constraints

A SecondOrderCone constrains the variables t and x to the set:

$$||x||_2 \leq t,$$

and $t \geq 0$. It can be added as follows:

```
julia> model = Model();

julia> @variable(model, t)
t

julia> @variable(model, x[1:2])
2-element Vector{VariableRef}:
 x[1]
 x[2]

julia> @constraint(model, [t; x] in SecondOrderCone())
[t, x[1], x[2]] ∈ MathOptInterface.SecondOrderCone(3)
```

12.19   Rotated second-order cone constraints

A RotatedSecondOrderCone constrains the variables t, u, and x to the set:

$$||x||_2^2 \leq 2t \cdot u$$

and $t, u \geq 0$. It can be added as follows:

```
julia> model = Model();

julia> @variable(model, t)
t

julia> @variable(model, u)
u

julia> @variable(model, x[1:2])
```

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

julia> @constraint(model, [t; u; x] in RotatedSecondOrderCone())
[t, u, x[1], x[2]] ∈ MathOptInterface.RotatedSecondOrderCone(4)
```

12.20   Semi-integer and semi-continuous variables

Semi-continuous variables are constrained to the set $x \in \{0\} \cup [l, u]$.

Create a semi-continuous variable using the Semicontinuous set:

```
julia> model = Model();

julia> @variable(model, x);

julia> @constraint(model, x in Semicontinuous(1.5, 3.5))
x in MathOptInterface.Semicontinuous{Float64}(1.5, 3.5)
```

Semi-integer variables are constrained to the set $x \in \{0\} \cup \{l, l+1, \ldots, u\}$.

Create a semi-integer variable using the Semiinteger set:

```
julia> model = Model();

julia> @variable(model, x);

julia> @constraint(model, x in Semiinteger(1.0, 3.0))
x in MathOptInterface.Semiinteger{Float64}(1.0, 3.0)
```

12.21   Special Ordered Sets of 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> model = Model();

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 (for example, 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, a vector of weights can be provided, and the variables are ordered according to their corresponding weight.

For example, in the constraint:

```julia
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].

12.22 Special Ordered Sets of 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
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])
```

The possible non-zero pairs are (x[1], x[3]) and (x[2], x[3]):

If the weight vector is omitted, JuMP induces an ordering from 1:length(x):

```julia
julia> @constraint(model, x in SOS2())
[x[1], x[2], x[3]] in MathOptInterface.SOS2{Float64}([1.0, 2.0, 3.0])
```

12.23 Indicator constraints

Indicator constraints consist of a binary variable and a linear constraint. The constraint holds when the binary variable takes the value 1. The constraint may or may not hold when the binary variable takes the value 0.

To enforce the constraint x + y <= 1 when the binary variable a is 1, use:

```julia
julia> model = Model();

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}
```

If the constraint must hold when a is zero, add ! or ¬ before the binary variable;

```
julia> @constraint(model, !a => {x + y <= 1})
!a => {x + y ≤ 1}
```

12.24 Semidefinite constraints

To constrain a matrix to be positive semidefinite (PSD), use PSDCone:

```
julia> model = Model();

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()
```

> **Tip**
>
> Where possible, prefer constructing a matrix of Semidefinite variables using the @variable macro,
> rather than adding a constraint like @constraint(model, X >= 0, PSDCone()). In some solvers,
> adding the constraint via @constraint is less efficient, and can result in additional intermediate vari-
> ables and constraints being added to the model.

The inequality X >= Y between two square matrices X and Y is understood as constraining X - Y to be 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()
```

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 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].)

12.25 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> model = Model();

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
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)
```

Chapter 13

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.

13.1 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.

```julia
julia> model = Model();

julia> @variable(model, x)
x

julia> @variable(model, y)
y

julia> ex = @expression(model, 2x + y - 1)
2 x + y - 1
```

This expression can be used in the objective or added to a constraint. For example:

```julia
julia> @objective(model, Min, 2 * ex - 1)
4 x + 2 y - 3

julia> objective_function(model)
4 x + 2 y - 3
```

Just like variables and constraints, named expressions can also be created. For example

```julia
julia> model = Model();
```

```
julia> @variable(model, x[i = 1:3]);

julia> @expression(model, expr[i = 1:3], i * sum(x[j] for j in i:3));

julia> expr
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.

```
julia> model = Model();

julia> @variable(model, x)
x

julia> @variable(model, y)
y

julia> ex = 2x + y - 1
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.

```
julia> model = Model();

julia> @variable(model, x)
x

julia> @variable(model, y)
y

julia> ex = AffExpr(-1.0, x => 2.0, y => 1.0)
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.

```
julia> model = Model();

julia> @variable(model, x)
x

julia> @variable(model, y)
y

julia> ex = AffExpr(-1.0)
-1

julia> add_to_expression!(ex, 2.0, x)
2 x - 1

julia> add_to_expression!(ex, 1.0, y)
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
```

13.2 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.

```
julia> model = Model();

julia> @variable(model, x)
x

julia> @variable(model, y)
y

julia> ex = @expression(model, x^2 + 2 * x * y + y^2 + x + y - 1)
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.

```
julia> model = Model();

julia> @variable(model, x)
x

julia> @variable(model, y)
y

julia> ex = x^2 + 2 * x * y + y^2 + x + y - 1
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.

```
julia> model = Model();

julia> @variable(model, x)
x

julia> @variable(model, y)
y

julia> aff_expr = AffExpr(-1.0, x => 1.0, y => 1.0)
x + y - 1

julia> quad_expr = QuadExpr(
           aff_expr,
           UnorderedPair(x, x) => 1.0,
           UnorderedPair(x, y) => 2.0,
           UnorderedPair(y, y) => 1.0,
       )
x² + 2 x*y + y² + x + y - 1
```

add_to_expression!

Finally, add_to_expression! can also be used to add quadratic terms.

```
julia> model = Model();

julia> @variable(model, x)
x

julia> @variable(model, y)
y

julia> ex = QuadExpr(x + y - 1.0)
x + y - 1

julia> add_to_expression!(ex, 1.0, x, x)
x² + x + y - 1

julia> add_to_expression!(ex, 2.0, x, y)
x² + 2 x*y + x + y - 1

julia> add_to_expression!(ex, 1.0, y, y)
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
```

13.3 Nonlinear expressions

Nonlinear expressions can be constructed only using the @NLexpression macro and can be used only in @NLobjective, @NLconstraint, and other @NLexpressions. For more details, see the Nonlinear Modeling section.

13.4 Initializing arrays

JuMP implements zero(AffExpr) and one(AffExpr) to support various functions in LinearAlgebra (for example, 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 14

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.

14.1   Set a linear objective

Use the @objective macro to set a linear objective function.

Use Min to create a minimization objective:

```julia
julia> model = Model();

julia> @variable(model, x);

julia> @objective(model, Min, 2x + 1)
2 x + 1
```

Use Max to create a maximization objective:

```julia
julia> model = Model();

julia> @variable(model, x);

julia> @objective(model, Max, 2x + 1)
2 x + 1
```

14.2   Set a quadratic objective

Use the @objective macro to set a quadratic objective function.

Use ^2 to have a variable squared:

```julia
julia> model = Model();

julia> @variable(model, x);

julia> @objective(model, Min, x^2 + 2x + 1)
x² + 2 x + 1
```

You can also have bilinear terms between variables:

```
julia> model = Model();

julia> @variable(model, x)
x

julia> @variable(model, y)
y

julia> @objective(model, Max, x * y + x + y)
x*y + x + y
```

14.3 Query the objective function

Use objective_function to return the current objective function.

```
julia> model = Model();

julia> @variable(model, x);

julia> @objective(model, Min, 2x + 1)
2 x + 1

julia> objective_function(model)
2 x + 1
```

14.4 Evaluate the objective function at a point

Use value to evaluate an objective function at a point specifying values for variables.

```
julia> model = Model();

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
```

14.5 Query the objective sense

Use objective_sense to return the current objective sense.

```
julia> model = Model();

julia> @variable(model, x);

julia> @objective(model, Min, 2x + 1)
2 x + 1

julia> objective_sense(model)
MIN_SENSE::OptimizationSense = 0
```

14.6   Modify an objective

To modify an objective, call @objective with the new objective function.

```
julia> model = Model();

julia> @variable(model, x);

julia> @objective(model, Min, 2x)
2 x

julia> @objective(model, Max, -2x)
-2 x
```

Alternatively, use set_objective_function.

```
julia> model = Model();

julia> @variable(model, x);

julia> @objective(model, Min, 2x)
2 x

julia> new_objective = @expression(model, -2 * x)
-2 x

julia> set_objective_function(model, new_objective)
```

14.7   Modify an objective coefficient

Use set_objective_coefficient to modify an objective coefficient.

```
julia> model = Model();

julia> @variable(model, x);

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.

14.8 Modify the objective sense

Use set_objective_sense to modify the objective sense.

```
julia> model = Model();

julia> @variable(model, x);

julia> @objective(model, Min, 2x)
2 x

julia> objective_sense(model)
MIN_SENSE::OptimizationSense = 0

julia> set_objective_sense(model, MAX_SENSE);

julia> objective_sense(model)
MAX_SENSE::OptimizationSense = 1
```

Alternatively, call @objective and pass the existing objective function.

```
julia> model = Model();

julia> @variable(model, x);

julia> @objective(model, Min, 2x)
2 x

julia> @objective(model, Max, objective_function(model))
2 x
```

14.9 Set a vector-valued objective

Define a multi-objective optimization problem by passing a vector of objectives:

```
julia> model = Model();

julia> @variable(model, x[1:2]);

julia> @objective(model, Min, [1 + x[1], 2 * x[2]])
2-element Vector{AffExpr}:
 x[1] + 1
```

```
 2 x[2]
```

```
julia> f = objective_function(model)
2-element Vector{AffExpr}:
 x[1] + 1
 2 x[2]
```

> **Tip**
>
> The Multi-objective knapsack tutorial provides an example of solving a multi-objective integer program.

In most cases, multi-objective optimization solvers will return multiple solutions, corresponding to points on the Pareto frontier. See Multiple solutions for information on how to query and work with multiple solutions.

Note that you must set a single objective sense, that is, you cannot have both minimization and maximization objectives. Work around this limitation by choosing `Min` and negating any objectives you want to maximize:

```
julia> model = Model();

julia> @variable(model, x[1:2]);

julia> @expression(model, obj1, 1 + x[1])
x[1] + 1

julia> @expression(model, obj2, 2 * x[1])
2 x[1]

julia> @objective(model, Min, [obj1, -obj2])
2-element Vector{AffExpr}:
 x[1] + 1
 -2 x[1]
```

Defining your objectives as expressions allows flexibility in how you can solve variations of the same problem, with some objectives removed and constrained to be no worse that a fixed value.

```
julia> model = Model();

julia> @variable(model, x[1:2]);

julia> @expression(model, obj1, 1 + x[1])
x[1] + 1

julia> @expression(model, obj2, 2 * x[1])
2 x[1]

julia> @expression(model, obj3, x[1] + x[2])
x[1] + x[2]

julia> @objective(model, Min, [obj1, obj2, obj3])  # Three-objective problem
3-element Vector{AffExpr}:
 x[1] + 1
 2 x[1]
```

```julia
 x[1] + x[2]

julia> # optimize!(model), look at the solution, talk to stakeholders, then
       # decide you want to solve a new problem where the third objective is
       # removed and constrained to be better than 2.0.
       nothing

julia> @objective(model, Min, [obj1, obj2])    # Two-objective problem
2-element Vector{AffExpr}:
 x[1] + 1
 2 x[1]

julia> @constraint(model, obj3 <= 2.0)
x[1] + x[2] ≤ 2
```

Chapter 15

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, for example, `name[key=index]`, or unnamed, for example, `[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, for example, `[i = 1:4]`, or they can be unnamed, for example, `[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`.

15.1 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 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)
```

Tables

Use Containers.rowtable to convert the Array into a Tables.jl compatible Vector{<:NamedTuple}:

```julia
julia> table = Containers.rowtable(x; header = [:I, :J, :value])
6-element Vector{NamedTuple{(:I, :J, :value), Tuple{Int64, Int64, Tuple{Int64, Int64}}}}:
 (I = 1, J = 1, value = (1, 1))
 (I = 2, J = 1, value = (2, 1))
 (I = 1, J = 2, value = (1, 2))
 (I = 2, J = 2, value = (2, 2))
 (I = 1, J = 3, value = (1, 3))
 (I = 2, J = 3, value = (2, 3))
```

Because it supports the Tables.jl interface, you can pass it to any function which accepts a table as input:

```julia
julia> import DataFrames;

julia> DataFrames.DataFrame(table)
6×3 DataFrame
 Row │ I      J      value
     │ Int64  Int64  Tuple…
─────┼──────────────────────
   1 │     1      1  (1, 1)
   2 │     2      1  (2, 1)
   3 │     1      2  (1, 2)
   4 │     2      2  (2, 2)
   5 │     1      3  (1, 3)
   6 │     2      3  (2, 3)
```

15.2 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
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
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
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
julia> axes(x)
(Base.OneTo(2), [:A, :B])
```

Broadcasting

Broadcasting over a DenseAxisArray returns a DenseAxisArray

```julia
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
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
julia> x.data
2×2 Matrix{Tuple{Int64, Symbol}}:
 (1, :A)  (1, :B)
 (2, :A)  (2, :B)
```

Tables

Use `Containers.rowtable` to convert the `DenseAxisArray` into a Tables.jl compatible `Vector{<:NamedTuple}`:

```julia
julia> table = Containers.rowtable(x; header = [:I, :J, :value])
4-element Vector{NamedTuple{(:I, :J, :value), Tuple{Int64, Symbol, Tuple{Int64, Symbol}}}}:
 (I = 1, J = :A, value = (1, :A))
 (I = 2, J = :A, value = (2, :A))
 (I = 1, J = :B, value = (1, :B))
 (I = 2, J = :B, value = (2, :B))
```

Because it supports the Tables.jl interface, you can pass it to any function which accepts a table as input:

```julia
julia> import DataFrames;

julia> DataFrames.DataFrame(table)
4×3 DataFrame
 Row │ I      J       value
     │ Int64  Symbol  Tuple…
─────┼───────────────────────
   1 │     1  A       (1, :A)
   2 │     2  A       (2, :A)
   3 │     1  B       (1, :B)
   4 │     2  B       (2, :B)
```

Keyword indexing

If all axes are named, you can use keyword indexing:

```julia
julia> x[i = 2, j = :A]
(2, :A)

julia> x[i = :, j = :B]
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, :B)
 (2, :B)
```

15.3 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], (i, j))
JuMP.Containers.SparseAxisArray{Tuple{Int64, Symbol}, 2, Tuple{Int64, Symbol}} with 4 entries:
  [2, A]  =  (2, :A)
  [2, B]  =  (2, :B)
  [3, A]  =  (3, :A)
  [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.

Slicing

Slicing is supported:

```
julia> y = x[:, :B]
JuMP.Containers.SparseAxisArray{Tuple{Int64, Symbol}, 1, Tuple{Int64}} with 2 entries:
  [2]  =  (2, :B)
  [3]  =  (3, :B)
```

Looping

Use eachindex to loop over the elements:

```
julia> for key in eachindex(y)
           println(y[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.(y)
JuMP.Containers.SparseAxisArray{Tuple{Symbol, Int64}, 1, Tuple{Int64}} with 2 entries:
  [2]  =  (:B, 2)
  [3]  =  (:B, 3)
```

Tables

Use Containers.rowtable to convert the SparseAxisArray into a Tables.jl compatible Vector{<:NamedTuple}:

```
julia> table = Containers.rowtable(x; header = [:I, :J, :value])
4-element Vector{NamedTuple{(:I, :J, :value), Tuple{Int64, Symbol, Tuple{Int64, Symbol}}}}:
 (I = 3, J = :B, value = (3, :B))
 (I = 2, J = :A, value = (2, :A))
 (I = 2, J = :B, value = (2, :B))
 (I = 3, J = :A, value = (3, :A))
```

Because it supports the Tables.jl interface, you can pass it to any function which accepts a table as input:

```
julia> import DataFrames;

julia> DataFrames.DataFrame(table)
4×3 DataFrame
 Row │ I      J       value
     │ Int64  Symbol  Tuple…
─────┼───────────────────────
   1 │     3  B       (3, :B)
   2 │     2  A       (2, :A)
   3 │     2  B       (2, :B)
   4 │     3  A       (3, :A)
```

Keyword indexing

If all axes are named, you can use keyword indexing:

```
julia> x[i = 2, j = :A]
(2, :A)

julia> x[i = :, j = :B]
JuMP.Containers.SparseAxisArray{Tuple{Int64, Symbol}, 1, Tuple{Int64}} with 2 entries:
  [2]  =  (2, :B)
  [3]  =  (3, :B)
```

15.4 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.

15.5   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
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
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
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 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, the only requirement is that it is an expression 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 16

Solutions

This section of the manual describes how to access a solved solution to a problem. It uses the following model as an example:

```julia
julia> begin
           model = Model(HiGHS.Optimizer)
           set_silent(model)
           @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)
       end
Max -12 x - 20 y[a]
Subject to
 6 x + 8 y[a] ≥ 100
 c1 : 7 x + 12 y[a] ≥ 120
 x ≥ 0
 y[a] ≤ 1
 y[b] ≤ 1
```

16.1 Solutions summary

solution_summary can be used for checking the summary of the optimization solutions.

```julia
julia> solution_summary(model)
* Solver : HiGHS

* Status
  Result count       : 1
  Termination status : OPTIMAL
  Message from the solver:
  "kHighsModelStatusOptimal"

* Candidate solution (result #1)
  Primal status      : FEASIBLE_POINT
```

```
  Dual status        : FEASIBLE_POINT
  Objective value    : -2.05143e+02
  Objective bound    : -0.00000e+00
  Relative gap       : Inf
  Dual objective value : -2.05143e+02

* Work counters
  Solve time (sec)   : 6.70987e-04
  Simplex iterations : 2
  Barrier iterations : 0
  Node count         : -1

julia> solution_summary(model; verbose = true)
* Solver : HiGHS

* Status
  Result count       : 1
  Termination status : OPTIMAL
  Message from the solver:
  "kHighsModelStatusOptimal"

* Candidate solution (result #1)
  Primal status      : FEASIBLE_POINT
  Dual status        : FEASIBLE_POINT
  Objective value    : -2.05143e+02
  Objective bound    : -0.00000e+00
  Relative gap       : Inf
  Dual objective value : -2.05143e+02
  Primal solution :
    x : 1.54286e+01
    y[a] : 1.00000e+00
    y[b] : 1.00000e+00
  Dual solution :
    c1 : 1.71429e+00

* Work counters
  Solve time (sec)   : 6.70987e-04
  Simplex iterations : 2
  Barrier iterations : 0
  Node count         : -1
```

16.2   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
julia> raw_status(model)
"kHighsModelStatusOptimal"
```

16.3 Primal solutions

Primal solution status

Use primal_status to return an MOI.ResultStatusCode enum describing the status of the primal solution.

```julia
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
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
julia> objective_value(model)
-205.14285714285714

julia> objective_bound(model)  # HiGHS only implements objective bound for MIPs
-0.0

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
julia> value(x)
15.428571428571429
```

Broadcast value over containers:

```julia
julia> value.(y)
1-dimensional DenseAxisArray{Float64,1,...} with index sets:
    Dimension 1, [:a, :b]
And data, a 2-element Vector{Float64}:
 1.0
 1.0
```

value also works on expressions:

```julia
julia> value(my_expr)
100.57142857142857
```

and constraints:

```julia
julia> value(c1)
120.0
```

> **Info**
>
> Calling value on a constraint returns the constraint function evaluated at the solution.

16.4   Dual solutions

Dual solution status

Use dual_status to return an MOI.ResultStatusCode enum describing the status of the dual solution.

```julia
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
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, [:a, :b]
And data, a 2-element Vector{Float64}:
 -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, [:a, :b]
And data, a 2-element Vector{Float64}:
  0.5714285714285694
 -0.0
```

16.5 Recommended workflow

The recommended workflow for solving a model and querying the solution is something like the following:

```
julia> begin
           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
        end
Solution is optimal
  objective value = -205.14285714285714
  primal solution: x = 15.428571428571429
  dual solution: c1 = 1.7142857142857142
```

16.6 OptimizeNotCalled errors

Due to differences in how solvers cache solutions internally, modifying a model after calling `optimize!` will reset the model into the `MOI.OPTIMIZE_NOT_CALLED` state. If you then 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:

```julia
julia> model = Model(HiGHS.Optimizer);

julia> set_silent(model)

julia> @variable(model, x >= 0);

julia> optimize!(model)

julia> termination_status(model)
OPTIMAL::TerminationStatusCode = 1

julia> set_upper_bound(x, 1)

julia> x_val = value(x)
┌ Warning: The model has been modified since the last call to `optimize!` (or `optimize!` has not
└  been called yet). If you are iteratively querying solution information and modifying a model,
└  query all the results first, then modify the model.
└ @ JuMP ~/work/JuMP.jl/JuMP.jl/src/optimizer_interface.jl:712
ERROR: OptimizeNotCalled()
Stacktrace:
[...]

julia> termination_status(model)
OPTIMIZE_NOT_CALLED::TerminationStatusCode = 0
```

do

```
julia> model = Model(HiGHS.Optimizer);

julia> set_silent(model)

julia> @variable(model, x >= 0);

julia> optimize!(model);

julia> x_val = value(x)
0.0

julia> termination_status(model)
OPTIMAL::TerminationStatusCode = 1

julia> set_upper_bound(x, 1)

julia> set_lower_bound(x, x_val)

julia> termination_status(model)
OPTIMIZE_NOT_CALLED::TerminationStatusCode = 0
```

If you know that your particular solver supports querying solution information after modifications, you can use direct_model to bypass the MOI.OPTIMIZE_NOT_CALLED state:

```
julia> model = direct_model(HiGHS.Optimizer());

julia> set_silent(model)

julia> @variable(model, x >= 0);

julia> optimize!(model)

julia> termination_status(model)
OPTIMAL::TerminationStatusCode = 1

julia> set_upper_bound(x, 1)

julia> x_val = value(x)
0.0

julia> set_lower_bound(x, x_val)

julia> termination_status(model)
OPTIMAL::TerminationStatusCode = 1
```

> **Warning**
>
> Be careful doing this. If your particular solver does not support querying solution information after modification, it may silently return incorrect solutions or throw an error.

16.7 Accessing attributes

MathOptInterface defines many 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

16.8 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.

> **Tip**
>
> Read the Sensitivity analysis of a linear program for more information on sensitivity analysis.

To give a simple example, we could analyze the sensitivity of the optimal solution to the following (non-degenerate) LP problem:

```julia
julia> begin
           model = Model(HiGHS.Optimizer)
           set_silent(model)
           @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)
       end
Max x[1]
Subject to
 c1 : x[1] + x[2] ≤ 1
 c2 : x[1] - x[2] ≤ 1
 x[2] ≥ -0.5
 x[2] ≤ 0.5
```

To analyze the sensitivity of the problem we could check the allowed perturbation ranges of, for example, 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, for example, 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, for example, 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.

16.9 Conflicts

When the model you input is infeasible, some solvers can help you find the cause of this infeasibility by offering a conflict, that is, a subset of the constraints that create this infeasibility. Depending on the solver, this can also be called an IIS (irreducible inconsistent subsystem).

If supported by the solver, use compute_conflict! to trigger the computation of a conflict. Once this process is finished, query the MOI.ConflictStatus attribute to check if a conflict was found.

If found, copy the IIS to a new model using copy_conflict, which you can then print or write to a file for easier debugging:

```
julia> using JuMP

julia> import Gurobi

julia> model = Model(Gurobi.Optimizer)
A JuMP Model
Feasibility problem with:
Variables: 0
Model mode: AUTOMATIC
CachingOptimizer state: EMPTY_OPTIMIZER
Solver name: Gurobi

julia> set_silent(model)

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

julia> @constraint(model, c1, x >= 2)
c1 : x ≥ 2.0

julia> @constraint(model, c2, x <= 1)
c2 : x ≤ 1.0

julia> optimize!(model)

julia> compute_conflict!(model)

julia> if get_attribute(model, MOI.ConflictStatus()) == MOI.CONFLICT_FOUND
           iis_model, _ = copy_conflict(model)
           print(iis_model)
       end
Feasibility
Subject to
 c1 : x ≥ 2.0
 c2 : x ≤ 1.0
```

If you need more control over the list of constraints that appear in the conflict, iterate over the list of constraints and query the MOI.ConstraintConflictStatus attribute:

```
julia> list_of_conflicting_constraints = ConstraintRef[]
ConstraintRef[]

julia> for (F, S) in list_of_constraint_types(model)
           for con in all_constraints(model, F, S)
               if get_attribute(con, MOI.ConstraintConflictStatus()) == MOI.IN_CONFLICT
                   push!(list_of_conflicting_constraints, con)
               end
           end
       end

julia> list_of_conflicting_constraints
2-element Vector{ConstraintRef}:
 c1 : x ≥ 2.0
 c2 : x ≤ 1.0
```

16.10 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, for example, primal_status, dual_status, value, dual, and solution_summary 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.

```julia
julia> using JuMP

julia> import MultiObjectiveAlgorithms as MOA

julia> import HiGHS

julia> model = Model(() -> MOA.Optimizer(HiGHS.Optimizer));

julia> set_attribute(model, MOA.Algorithm(), MOA.Dichotomy())

julia> set_silent(model)

julia> @variable(model, x1 >= 0)
x1

julia> @variable(model, 0 <= x2 <= 3)
x2

julia> @objective(model, Min, [3x1 + x2, -x1 - 2x2])
2-element Vector{AffExpr}:
 3 x1 + x2
 -x1 - 2 x2

julia> @constraint(model, 3x1 - x2 <= 6)
3 x1 - x2 ≤ 6

julia> optimize!(model)

julia> solution_summary(model; result = 1)
* Solver : MOA[algorithm=MultiObjectiveAlgorithms.Dichotomy, optimizer=HiGHS]

* Status
  Result count      : 3
  Termination status : OPTIMAL
  Message from the solver:
  "Solve complete. Found 3 solution(s)"

* Candidate solution (result #1)
  Primal status      : FEASIBLE_POINT
  Dual status        : NO_SOLUTION
  Objective value    : [0.00000e+00,0.00000e+00]
```

```
  Objective bound    : [0.00000e+00,-9.00000e+00]
  Relative gap       : Inf
  Dual objective value : -9.00000e+00

* Work counters
  Solve time (sec)   : 1.82720e-03
  Simplex iterations : 1
  Barrier iterations : 0
  Node count         : -1

julia> for i in 1:result_count(model)
           println("Solution $i")
           println("   x = ", value.([x1, x2]; result = i))
           println(" obj = ", objective_value(model; result = i))
       end
Solution 1
   x = [0.0, 0.0]
 obj = [0.0, 0.0]
Solution 2
   x = [0.0, 3.0]
 obj = [3.0, -6.0]
Solution 3
   x = [3.0, 3.0]
 obj = [12.0, -9.0]
```

> **Tip**
>
> The Multi-objective knapsack tutorial provides more examples of querying multiple solutions.

16.11 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(HiGHS.Optimizer);

julia> set_silent(model)

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; atol = 0.0)
Dict{Any, Float64}()
```

Calling `primal_feasibility_report` without the `point` argument is useful when `primal_status` is FEASIBLE_POINT or NEARLY_FEASIBLE_POINT, and you want to assess the solution quality.

> **Warning**
>
> To apply `primal_feasibility_report` to infeasible models, you must also provide a candidate point (solvers generally do not provide one). To diagnose the source of infeasibility, see Conflicts.

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 17

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.

17.1 Set a nonlinear objective

Use `@NLobjective` to set a nonlinear objective.

```julia
julia> model = Model();

julia> @variable(model, x[1:2]);

julia> @NLobjective(model, Min, exp(x[1]) - sqrt(x[2]))
```

To modify a nonlinear objective, call `@NLobjective` again.

17.2 Add a nonlinear constraint

Use `@NLconstraint` to add a nonlinear constraint.

```
julia> model = Model();

julia> @variable(model, x[1:2]);

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.

Delete a nonlinear constraint using delete:

```
julia> delete(model, con[1])
```

17.3   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> model = Model();

julia> @variable(model, x[1:2]);

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 @NLexpressions.

```
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])
```

Use value to query the value of a nonlinear expression:

```
julia> set_start_value(x[1], 1.0)

julia> value(start_value, nested[1])
0.7456241416655579

julia> sin(sin(1.0))
0.7456241416655579
```

17.4 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> model = Model();

julia> @variable(model, x);

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
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
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:

```julia
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.

17.5 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
julia> model = Model();

julia> @variable(model, x);

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

Except for the splatting syntax discussed below, all expressions must be simple scalar operations. You cannot use dot, matrix-vector products, vector slices, etc.

```julia
julia> model = Model();

julia> @variable(model, x[1:2]);

julia> @variable(model, y);

julia> c = [1, 2];

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
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: Unsupported use of the splatting operator. JuMP supports splatting only symbols. For
↪  example, `x...` is ok, but `(x + 1)...`, `[x; y]...` and `g(f(y)...)` are not.
```

17.6 User-defined Functions

JuMP natively supports the set of univariate and multivariate functions recognized by the `MOI.Nonlinear` submodule. In addition to this list of functions, it is possible to register custom user-defined nonlinear functions. User-defined functions can be used anywhere in @NLobjective, @NLconstraint, and @NLexpression.

JuMP will attempt to automatically register functions it detects in your nonlinear expressions, which usually means 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.

Common mistakes when writing a user-defined function

> **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, that is, 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 more readable if it does. Make sure you use `my_f` and not `f` in the macros.

> **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 returns 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

You can also register a function with the second-order derivative information, which is a scalar for univariate functions, and a symmetric matrix for multivariate functions.

Univariate functions

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))
```

Multivariate functions

For multivariate functions, the hessian function ∇²f must take an `AbstractMatrix` as the first argument, the lower-triangular of which is filled in-place:

```
f(x...) = (1 - x[1])^2 + 100 * (x[2] - x[1]^2)^2
function ∇f(g, x...)
    g[1] = 400 * x[1]^3 - 400 * x[1] * x[2] + 2 * x[1] - 2
    g[2] = 200 * (x[2] - x[1]^2)
    return
end
function ∇²f(H, x...)
    H[1, 1] = 1200 * x[1]^2 - 400 * x[2] + 2
    # H[1, 2] = -400 * x[1]   <-- Not needed. Fill the lower-triangular only.
    H[2, 1] = -400 * x[1]
    H[2, 2] = 200.0
    return
end

model = Model()
register(model, :rosenbrock, 2, f, ∇f, ∇²f)
@variable(model, x[1:2])
@NLobjective(model, Min, rosenbrock(x[1], x[2]))
```

> **Warning**
>
> You may assume the Hessian matrix H is initialized with zeros, and because H is symmetric, you need only to fill in the non-zero of the lower-triangular terms. The matrix type passed in as H depends on the automatic differentiation system, so make sure the first argument to the Hessian function supports an `AbstractMatrix` (it may be something other than `Matrix{Float64}`). However, you may assume only that H supports `size(H)` and `setindex!`. Finally, the matrix is treated as dense, so the performance will be poor on functions with high-dimensional input.

User-defined functions with vector inputs

User-defined functions which take vectors as input arguments (for example, `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.

17.7 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. As a conservative bound, JuMP's performance here currently may be expected to be within a factor of 5 of AMPL's. Our paper in SIAM Review has more details.

17.8   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:

```julia
julia> raw_index(v::MOI.VariableIndex) = v.value
raw_index (generic function with 1 method)

julia> model = Model();

julia> @variable(model, x)
x

julia> @variable(model, y)
y

julia> @NLobjective(model, Min, sin(x) + sin(y))

julia> values = zeros(2)
2-element Vector{Float64}:
 0.0
 0.0

julia> x_index = raw_index(JuMP.index(x))
1

julia> y_index = raw_index(JuMP.index(y))
2

julia> values[x_index] = 2.0
2.0

julia> values[y_index] = 3.0
3.0

julia> d = NLPEvaluator(model)
Nonlinear.Evaluator with available features:
  * :Grad
  * :Jac
  * :JacVec
  * :Hess
  * :HessVec
  * :ExprGraph

julia> MOI.initialize(d, [:Grad])

julia> MOI.eval_objective(d, values)
1.0504174348855488

julia> sin(2.0) + sin(3.0)
1.0504174348855488
```

```
julia> ∇f = zeros(2)
2-element Vector{Float64}:
 0.0
 0.0

julia> MOI.eval_objective_gradient(d, ∇f, values)

julia> ∇f[x_index], ∇f[y_index]
(-0.4161468365471424, -0.9899924966004454)

julia> cos(2.0), cos(3.0)
(-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 `MOI.Nonlinear.ConstraintIndex`. 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{GenericModel{Float64},
↪  MathOptInterface.Nonlinear.ConstraintIndex, ScalarShape})

julia> index(cons1)
MathOptInterface.Nonlinear.ConstraintIndex(1)

julia> index(cons2)
MathOptInterface.Nonlinear.ConstraintIndex(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, for example, 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`.

17.9 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_nonlinear_expression, set_nonlinear_objective and add_nonlinear_constraint.

This input form may be useful if the expressions are generated programmatically, or if you experience compilation issues with the macro input (see Known performance issues for more information).

Add a nonlinear expression

Use add_nonlinear_expression to add a nonlinear expression to the model.

```julia
julia> model = Model();

julia> @variable(model, x)
x

julia> expr = :($(x) + sin($(x)^2))
:(x + sin(x ^ 2))

julia> expr_ref = add_nonlinear_expression(model, expr)
subexpression[1]: x + sin(x ^ 2.0)
```

This is equivalent to

```julia
julia> model = Model();

julia> @variable(model, x);

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_nonlinear_objective to set a nonlinear objective.

```julia
julia> model = Model();

julia> @variable(model, x);

julia> expr = :($(x) + $(x)^2)
:(x + x ^ 2)
```

```julia
julia> set_nonlinear_objective(model, MIN_SENSE, expr)
```

This is equivalent to

```julia
julia> model = Model();

julia> @variable(model, x);

julia> @NLobjective(model, Min, x + x^2)
```

> **Note**
>
> You must use MIN_SENSE or MAX_SENSE instead of Min and Max.

Add a constraint

Use add_nonlinear_constraint to add a nonlinear constraint.

```julia
julia> model = Model();

julia> @variable(model, x);

julia> expr = :($(x) + $(x)^2)
:(x + x ^ 2)

julia> add_nonlinear_constraint(model, :($(expr) <= 1))
(x + x ^ 2.0) - 1.0 ≤ 0
```

This is equivalent to

```julia
julia> model = Model();

julia> @variable(model, x);

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
julia> function main(functions::Vector{Function})
           model = Model()
           @variable(model, x)
```

```
            for (i, f) in enumerate(functions)
                f_sym = Symbol("f_$(i)")
                register(model, f_sym, 1, f; autodiff = true)
                add_nonlinear_constraint(model, :($(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)($(x))))
            end
            add_nonlinear_constraint(model, :($(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
```

Registered functions with a variable number of arguments

User defined functions require a fixed number of input arguments. However, sometimes you will want to use a registered function like:

```
julia> f(x...) = sum(exp(x[i]^2) for i in 1:length(x));
```

with different numbers of arguments.

The solution is to register the same function f for each unique number of input arguments, making sure to use a unique name each time. For example:

```
julia> A = [[1], [1, 2], [2, 3, 4], [1, 3, 4, 5]];
```

```julia
julia> model = Model();

julia> @variable(model, x[1:5]);

julia> funcs = Set{Symbol}();

julia> for a in A
           key = Symbol("f$(length(a))")
           if !(key in funcs)
               push!(funcs, key)
               register(model, key, length(a), f; autodiff = true)
           end
           add_nonlinear_constraint(model, :($key($(x[a]...)) <= 1))
       end

julia> print(model)
Feasibility
Subject to
 f1(x[1]) - 1.0 ≤ 0
 f2(x[1], x[2]) - 1.0 ≤ 0
 f3(x[2], x[3], x[4]) - 1.0 ≤ 0
 f4(x[1], x[3], x[4], x[5]) - 1.0 ≤ 0
```

17.10   Known performance issues

The macro-based input to JuMP's nonlinear interface can cause a performance issue if you:

1. write a macro with a large number (hundreds) of terms

2. call that macro from within a function instead of from the top-level in global scope.

The first issue does not depend on the number of resulting terms in the mathematical expression, but rather the number of terms in the Julia Expr representation of that expression. For example, the expression sum(x[i] for i in 1:1_000_000) contains one million mathematical terms, but the Expr representation is just a single sum.

The most common cause, other than a lot of tedious typing, is if you write a program that automatically writes a JuMP model as a text file, which you later execute. One example is MINLPlib.jl which automatically transpiled models in the GAMS scalar format into JuMP examples.

As a rule of thumb, if you are writing programs to automatically generate expressions for the JuMP macros, you should target the Raw expression input instead. For more information, read MathOptInterface Issue#1997.

Chapter 18

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

18.1 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.

18.2 Things you can and cannot do during solver-independent callbacks

There is a 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 (for example, 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.

18.3 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 constraints 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:

```julia
julia> import GLPK

julia> model = Model(GLPK.Optimizer);

julia> @variable(model, x <= 10, Int)
x

julia> @objective(model, Max, x)
x

julia> 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
my_callback_function (generic function with 1 method)

julia> set_attribute(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 many 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
> set_attribute(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, for example, because the earlier lazy constraint was removed during presolve. If this happens, you must re-add the lazy constraint.

18.4 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:

```julia
julia> import GLPK

julia> model = Model(GLPK.Optimizer);

julia> @variable(model, x <= 10.5, Int)
x

julia> @objective(model, Max, x)
x

julia> 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
my_callback_function (generic function with 1 method)

julia> set_attribute(model, MOI.UserCutCallback(), my_callback_function)
```

> **Warning**
>
> User cuts must not change the set of integer feasible solutions. Equivalently, user cuts can only remove fractional solutions. If you add a cut that removes an integer solution (even one that is not optimal), 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.

18.5 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" (for example, 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, for example, you can consistently take fractional solutions and intelligently guess integer solutions from them.

A heuristic solution callback can be set using the following syntax:

```julia
julia> import GLPK

julia> model = Model(GLPK.Optimizer);
```

```julia
julia> @variable(model, x <= 10.5, Int)
x

julia> @objective(model, Max, x)
x

julia> 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
my_callback_function (generic function with 1 method)

julia> set_attribute(model, MOI.HeuristicCallback(), my_callback_function)
```

The third argument to `submit` is a vector of JuMP variables, and the fourth argument is 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.

Chapter 19

Complex number support

This page explains the complex-valued variables and constraints that JuMP supports. For a worked-example using these features, read the Quantum state discrimination tutorial.

19.1 Complex-valued variables

Create a complex-valued variable using ComplexPlane:

```julia
julia> model = Model();

julia> @variable(model, x in ComplexPlane())
real(x) + imag(x) im
```

Note that x is not a VariableRef; instead, it is an affine expression with Complex{Float64}-valued coefficients:

```julia
julia> typeof(x)
GenericAffExpr{ComplexF64, VariableRef}
```

Behind the scenes, JuMP has created two real-valued variables, with names "real(x)" and "imag(x)":

```julia
julia> all_variables(model)
2-element Vector{VariableRef}:
 real(x)
 imag(x)

julia> name.(all_variables(model))
2-element Vector{String}:
 "real(x)"
 "imag(x)"
```

Use the real and imag functions on x to return a real-valued affine expression representing each variable:

```julia
julia> typeof(real(x))
AffExpr (alias for GenericAffExpr{Float64, GenericVariableRef{Float64}})

julia> typeof(imag(x))
AffExpr (alias for GenericAffExpr{Float64, GenericVariableRef{Float64}})
```

To create an anonymous variable, use the `set` keyword argument:

```julia
julia> model = Model();

julia> x = @variable(model, set = ComplexPlane())
_[1] + _[2] im
```

19.2   Complex-valued variable bounds

Because complex-valued variables lack a total ordering, the definition of a variable bound for a complex-valued variable is ambiguous. If you pass a real- or complex-valued argument to keywords such as `lower_bound`, `upper_bound`, and `start_value`, JuMP will apply the real and imaginary parts to the associated real-valued variables.

```julia
julia> model = Model();

julia> @variable(
           model,
           x in ComplexPlane(),
           lower_bound = 1.0,
           upper_bound = 2.0 + 3.0im,
           start = 4im,
       )
real(x) + imag(x) im

julia> vars = all_variables(model)
2-element Vector{VariableRef}:
 real(x)
 imag(x)

julia> lower_bound.(vars)
2-element Vector{Float64}:
 1.0
 0.0

julia> upper_bound.(vars)
2-element Vector{Float64}:
 2.0
 3.0

julia> start_value.(vars)
2-element Vector{Float64}:
 0.0
 4.0
```

19.3   Complex-valued equality constraints

JuMP reformulates complex-valued equality constraints into two real-valued constraints: one representing the real part, and one representing the imaginary part. Thus, complex-valued equality constraints can be solved any solver that supports the real-valued constraint type.

For example:

```
julia> model = Model(HiGHS.Optimizer);

julia> set_silent(model)

julia> @variable(model, x[1:2]);

julia> @constraint(model, (1 + 2im) * x[1] + 3 * x[2] == 4 + 5im)
(1 + 2im) x[1] + 3 x[2] = (4 + 5im)

julia> optimize!(model)

julia> value.(x)
2-element Vector{Float64}:
 2.5
 0.5
```

is equivalent to

```
julia> model = Model(HiGHS.Optimizer);

julia> set_silent(model)

julia> @variable(model, x[1:2]);

julia> @constraint(model, 1 * x[1] + 3 * x[2] == 4)  # real component
x[1] + 3 x[2] = 4

julia> @constraint(model, 2 * x[1] == 5)  # imag component
2 x[1] = 5

julia> optimize!(model)

julia> value.(x)
2-element Vector{Float64}:
 2.5
 0.5
```

This also applies if the variables are complex-valued:

```
julia> model = Model(HiGHS.Optimizer);

julia> set_silent(model)

julia> @variable(model, x in ComplexPlane());

julia> @constraint(model, (1 + 2im) * x + 3 * x == 4 + 5im)
(4 + 2im) real(x) + (-2 + 4im) imag(x) = (4 + 5im)

julia> optimize!(model)

julia> value(x)
1.3 + 0.6000000000000001im
```

which is equivalent to

```julia
julia> model = Model(HiGHS.Optimizer);

julia> set_silent(model)

julia> @variable(model, x_real);

julia> @variable(model, x_imag);

julia> @constraint(model, x_real - 2 * x_imag + 3 * x_real == 4)
4 x_real - 2 x_imag = 4

julia> @constraint(model, x_imag + 2 * x_real + 3 * x_imag == 5)
2 x_real + 4 x_imag = 5

julia> optimize!(model)

julia> value(x_real) + value(x_imag) * im
1.3 + 0.6000000000000001im
```

19.4 Hermitian PSD Cones

JuMP supports creating matrices where are Hermitian.

```julia
julia> model = Model();

julia> @variable(model, H[1:3, 1:3] in HermitianPSDCone())
3×3 LinearAlgebra.Hermitian{GenericAffExpr{ComplexF64, VariableRef},
↪  Matrix{GenericAffExpr{ComplexF64, VariableRef}}}:
 real(H[1,1])                       …  real(H[1,3]) + imag(H[1,3]) im
 real(H[1,2]) - imag(H[1,2]) im        real(H[2,3]) + imag(H[2,3]) im
 real(H[1,3]) - imag(H[1,3]) im        real(H[3,3])
```

Behind the scenes, JuMP has created nine real-valued decision variables:

```julia
julia> all_variables(model)
9-element Vector{VariableRef}:
 real(H[1,1])
 real(H[1,2])
 real(H[2,2])
 real(H[1,3])
 real(H[2,3])
 real(H[3,3])
 imag(H[1,2])
 imag(H[1,3])
 imag(H[2,3])
```

and a `Vector{VariableRef}-in-MOI.HermitianPositiveSemidefiniteConeTriangle` constraint:

```
julia> num_constraints(model, Vector{VariableRef}, MOI.HermitianPositiveSemidefiniteConeTriangle)
1
```

The `MOI.HermitianPositiveSemidefiniteConeTriangle` set can be efficiently bridged to `MOI.PositiveSemidefiniteConeTria`
so it can be solved by any solver that supports PSD constraints.

Each element of H is an affine expression with Complex{Float64}-valued coefficients:

```
julia> typeof(H[1, 1])
GenericAffExpr{ComplexF64, VariableRef}

julia> typeof(H[2, 1])
GenericAffExpr{ComplexF64, VariableRef}
```

19.5 Hermitian PSD constraints

The `HermitianPSDCone` can also be used in the `@constraint` macro:

```
julia> model = Model();

julia> @variable(model, x[1:2])
2-element Vector{VariableRef}:
 x[1]
 x[2]

julia> import LinearAlgebra

julia> H = LinearAlgebra.Hermitian([x[1] 1im; -1im -x[2]])
2×2 LinearAlgebra.Hermitian{GenericAffExpr{ComplexF64, VariableRef},
↪  Matrix{GenericAffExpr{ComplexF64, VariableRef}}}:
 x[1]   im
 -im   -x[2]

julia> @constraint(model, H in HermitianPSDCone())
[x[1]  im;
 -im   -x[2]] ∈ HermitianPSDCone()
```

> **Note**
>
> The matrix H in H in HermitianPSDCone() must be a LinearAlgebra.Hermitian matrix type. A
> build_constraint error will be thrown if the matrix is a different matrix type.

Part IV

API Reference

Chapter 20

Models

More information can be found in the Models section of the manual.

20.1 Constructors

JuMP.GenericModel – Type.

```
GenericModel{T}()
```

Return a new JuMP model of value type T without any optimizer; the model is stored in a cache.

Use set_optimizer to set the optimizer before calling optimize!.

source

```
GenericModel{T}(optimizer_factory; add_bridges::Bool = true) where {T}
```

Return a new JuMP model with the provided optimizer, bridge settings and value_type T.

See set_optimizer for the description of the optimizer_factory and add_bridges arguments.

Example

```julia
julia> import Ipopt

julia> model = Model(Ipopt.Optimizer);

julia> solver_name(model)
"Ipopt"
```

```julia
julia> import HiGHS

julia> import MultiObjectiveAlgorithms as MOA

julia> model = Model(() -> MOA.Optimizer(HiGHS.Optimizer); add_bridges = false);
```

source

JuMP.direct_model – Function.

```
direct_model(backend::MOI.ModelLike, T::Type = Float64)
```

Return a new JuMP model using backend to store the model and solve it.

As opposed to the GenericModel 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 GenericModel 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

```
julia> import HiGHS

julia> optimizer = optimizer_with_attributes(
           HiGHS.Optimizer,
           "presolve" => "off",
           MOI.Silent() => true,
       );

julia> model = direct_model(optimizer)
A JuMP Model
Feasibility problem with:
Variables: 0
Model mode: DIRECT
Solver name: HiGHS
```

is equivalent to:

```
julia> model = direct_model(HiGHS.Optimizer())
A JuMP Model
Feasibility problem with:
Variables: 0
```

```
Model mode: DIRECT
Solver name: HiGHS

julia> set_attribute(model, "presolve", "off")

julia> set_attribute(model, MOI.Silent(), true)
```

source

20.2 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

20.3 Basic functions

JuMP.backend – Function.

```
backend(model::GenericModel)
```

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::GenericModel)
```

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.

```julia
julia> import HiGHS

julia> model = Model(HiGHS.Optimizer)
A JuMP Model
Feasibility problem with:
Variables: 0
Model mode: AUTOMATIC
CachingOptimizer state: EMPTY_OPTIMIZER
Solver name: HiGHS

julia> MOI.Utilities.attach_optimizer(model)

julia> inner = unsafe_backend(model)
A HiGHS model with 0 columns and 0 rows.
```

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:

```julia
julia> import HiGHS

julia> model = Model(HiGHS.Optimizer);

julia> set_silent(model)

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

julia> MOI.Utilities.attach_optimizer(model)

julia> highs = unsafe_backend(model)
A HiGHS model with 1 columns and 0 rows.
```

Use:

```julia
julia> import HiGHS

julia> model = direct_model(HiGHS.Optimizer());

julia> set_silent(model)

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

julia> highs = backend(model)  # No need to call `attach_optimizer`.
A HiGHS model with 1 columns and 0 rows.
```

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::GenericModel)
```

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::GenericModel)::GenericModel
```

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

`Base.isempty` – Method.

```
isempty(model::GenericModel)
```

Verifies whether the model is empty, that is, whether the MOI backend is empty and whether the model is in the same state as at its creation apart from optimizer attributes.

source

`JuMP.mode` – Function.

```
mode(model::GenericModel)
```

Return the ModelMode (DIRECT, AUTOMATIC, or MANUAL) of model.

source

`JuMP.object_dictionary` – Function.

```
object_dictionary(model::GenericModel)
```

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::GenericModel, 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` as well.

See also: `delete`, `object_dictionary`.

Example

```julia
julia> model = Model();

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

JuMP.set_string_names_on_creation – Function.

```
set_string_names_on_creation(model::GenericModel, value::Bool)
```

Set the default argument of the set_string_name keyword in the @variable and @constraint macros to value. This is used to determine whether to assign String names to all variables and constraints in model.

By default, value is true. However, for larger models calling set_string_names_on_creation(model, false) can improve performance at the cost of reducing the readability of printing and solver log messages.

source

JuMP.value_type – Function.

```
value_type(::Type{<:Union{AbstractModel,AbstractVariableRef}})
```

Return the return type of value for variables of that model. It defaults to Float64 if it is not implemented.

source

20.4 Working with attributes

JuMP.set_optimizer – Function.

```
set_optimizer(
    model::GenericModel,
    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_attribute for setting solver-specific parameters of the optimizer.

Example

```
julia> import HiGHS

julia> model = Model();

julia> set_optimizer(model, () -> HiGHS.Optimizer())

julia> set_optimizer(model, HiGHS.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_attribute.

See also: set_attribute, get_attribute.

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.

Example

```julia
julia> import HiGHS

julia> optimizer = optimizer_with_attributes(
           HiGHS.Optimizer, "presolve" => "off", MOI.Silent() => true,
       );

julia> model = Model(optimizer);
```

is equivalent to:

```julia
julia> model = Model(HiGHS.Optimizer);

julia> set_attribute(model, "presolve", "off")

julia> set_attribute(model, MOI.Silent(), true)
```

source

JuMP.get_attribute – Function.

```
get_attribute(model::GenericModel, attr::MOI.AbstractModelAttribute)
get_attribute(x::GenericVariableRef, attr::MOI.AbstractVariableAttribute)
get_attribute(cr::ConstraintRef, attr::MOI.AbstractConstraintAttribute)
```

Get the value of a solver-specifc attribute attr.

This is equivalent to calling MOI.get with the associated MOI model and, for variables and constraints, with the associated MOI.VariableIndex or MOI.ConstraintIndex.

Example

```
julia> model = Model();

julia> @variable(model, x)
x

julia> @constraint(model, c, 2 * x <= 1)
c : 2 x ≤ 1.0

julia> get_attribute(model, MOI.Name())
""

julia> get_attribute(x, MOI.VariableName())
"x"

julia> get_attribute(c, MOI.ConstraintName())
"c"
```

source

```
get_attribute(
    model::Union{GenericModel,MOI.OptimizerWithAttributes},
    attr::Union{AbstractString,MOI.AbstractOptimizerAttribute},
)
```

Get the value of a solver-specifc attribute `attr`.

This is equivalent to calling MOI.get with the associated MOI model.

If `attr` is an AbstractString, it is converted to MOI.RawOptimizerAttribute.

Example

```
julia> import HiGHS

julia> opt = optimizer_with_attributes(HiGHS.Optimizer, "output_flag" => true);

julia> model = Model(opt);

julia> get_attribute(model, "output_flag")
true

julia> get_attribute(model, MOI.RawOptimizerAttribute("output_flag"))
true

julia> get_attribute(opt, "output_flag")
true

julia> get_attribute(opt, MOI.RawOptimizerAttribute("output_flag"))
true
```

source

JuMP.set_attribute – Function.

```
set_attribute(model::GenericModel, attr::MOI.AbstractModelAttribute, value)
set_attribute(x::GenericVariableRef, attr::MOI.AbstractVariableAttribute, value)
set_attribute(cr::ConstraintRef, attr::MOI.AbstractConstraintAttribute, value)
```

Set the value of a solver-specifc attribute `attr` to value.

This is equivalent to calling `MOI.set` with the associated MOI model and, for variables and constraints, with the associated `MOI.VariableIndex` or `MOI.ConstraintIndex`.

Example

```julia
julia> model = Model();

julia> @variable(model, x)
x

julia> @constraint(model, c, 2 * x <= 1)
c : 2 x ≤ 1.0

julia> set_attribute(model, MOI.Name(), "model_new")

julia> set_attribute(x, MOI.VariableName(), "x_new")

julia> set_attribute(c, MOI.ConstraintName(), "c_new")
```

source

```
set_attribute(
    model::Union{GenericModel,MOI.OptimizerWithAttributes},
    attr::Union{AbstractString,MOI.AbstractOptimizerAttribute},
    value,
)
```

Set the value of a solver-specifc attribute `attr` to value.

This is equivalent to calling `MOI.set` with the associated MOI model.

If `attr` is an AbstractString, it is converted to `MOI.RawOptimizerAttribute`.

Example

```julia
julia> import HiGHS

julia> opt = optimizer_with_attributes(HiGHS.Optimizer, "output_flag" => false);

julia> model = Model(opt);

julia> set_attribute(model, "output_flag", false)

julia> set_attribute(model, MOI.RawOptimizerAttribute("output_flag"), true)

julia> set_attribute(opt, "output_flag", true)

julia> set_attribute(opt, MOI.RawOptimizerAttribute("output_flag"), false)
```

JuMP.set_attributes – Function.

```
set_attributes(
    destination::Union{
        GenericModel,
        MOI.OptimizerWithAttributes,
        GenericVariableRef,
        ConstraintRef,
    },
    pairs::Pair...,
)
```

Given a list of attribute => value pairs, calls set_attribute(destination, attribute, value) for each pair.

See also: set_attribute, get_attribute.

Example

```julia
julia> import Ipopt

julia> model = Model(Ipopt.Optimizer);

julia> set_attributes(model, "tol" => 1e-4, "max_iter" => 100)
```

is equivalent to:

```julia
julia> import Ipopt

julia> model = Model(Ipopt.Optimizer);

julia> set_attribute(model, "tol", 1e-4)

julia> set_attribute(model, "max_iter", 100)
```

JuMP.set_silent – Function.

```
set_silent(model::GenericModel)
```

Takes precedence over any other attribute controlling verbosity and requires the solver to produce no output.

See also: unset_silent.

JuMP.unset_silent – Function.

```
unset_silent(model::GenericModel)
```

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::GenericModel, limit::Float64)
```

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::GenericModel)
```

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::GenericModel)
```

Return the time limit (in seconds) of the model.

Returns nothing if unset.

See also: set_time_limit_sec, unset_time_limit_sec.

source

JuMP.set_start_values – Function.

```
set_start_values(
    model::GenericModel;
    variable_primal_start::Union{Nothing,Function} = value,
    constraint_primal_start::Union{Nothing,Function} = value,
    constraint_dual_start::Union{Nothing,Function} = dual,
    nonlinear_dual_start::Union{Nothing,Function} = nonlinear_dual_start_value,
)
```

Set the primal and dual starting values in `model` using the functions provided.

If any keyword argument is `nothing`, the corresponding start value is skipped.

If the optimizer does not support setting the starting value, the value will be skipped.

`variable_primal_start`

This function controls the primal starting solution for the variables. It is equivalent to calling `set_start_value` for each variable, or setting the `MOI.VariablePrimalStart` attribute.

If it is a function, it must have the form `variable_primal_start(x::VariableRef)` that maps each variable x to the starting primal value.

The default is `value`.

`constraint_primal_start`

This function controls the primal starting solution for the constraints. It is equivalent to calling `set_start_value` for each constraint, or setting the `MOI.ConstraintPrimalStart` attribute.

If it is a function, it must have the form `constraint_primal_start(ci::ConstraintRef)` that maps each constraint ci to the starting primal value.

The default is `value`.

`constraint_dual_start`

This function controls the dual starting solution for the constraints. It is equivalent to calling `set_dual_start_value` for each constraint, or setting the `MOI.ConstraintDualStart` attribute.

If it is a function, it must have the form `constraint_dual_start(ci::ConstraintRef)` that maps each constraint ci to the starting dual value.

The default is `dual`.

`nonlinear_dual_start`

This function controls the dual starting solution for the nonlinear constraints It is equivalent to calling `set_nonlinear_dual_start_value`.

If it is a function, it must have the form `nonlinear_dual_start(model::GenericModel)` that returns a vector corresponding to the dual start of the constraints.

The default is `nonlinear_dual_start_value`.

source

JuMP.get_optimizer_attribute – Function.

```
get_optimizer_attribute(
    model::Union{GenericModel,MOI.OptimizerWithAttributes},
    attr::Union{AbstractString,MOI.AbstractOptimizerAttribute},
)
```

Return the value associated with the solver-specific attribute `attr`.

If `attr` is an `AbstractString`, this is equivalent to `get_optimizer_attribute(model, MOI.RawOptimizerAttribute(name))`.

> **Compat**
>
> This method will remain in all v1.X releases of JuMP, but it may be removed in a future v2.0 release. We recommend using `get_attribute` instead.

See also: set_optimizer_attribute, set_optimizer_attributes.

Example

```julia
julia> import Ipopt

julia> model = Model(Ipopt.Optimizer);

julia> get_optimizer_attribute(model, MOI.Silent())
false
```

source

JuMP.set_optimizer_attribute – Function.

```
set_optimizer_attribute(
    model::Union{GenericModel,MOI.OptimizerWithAttributes},
    attr::Union{AbstractString,MOI.AbstractOptimizerAttribute},
    value,
)
```

Set the solver-specific attribute attr in model to value.

If attr is an AbstractString, this is equivalent to set_optimizer_attribute(model, MOI.RawOptimizerAttribute(name),
value).

> **Compat**
>
> This method will remain in all v1.X releases of JuMP, but it may be removed in a future v2.0 release.
> We recommend using set_attribute instead.

See also: set_optimizer_attributes, get_optimizer_attribute.

Example

```julia
julia> model = Model();

julia> set_optimizer_attribute(model, MOI.Silent(), true)
```

source

JuMP.set_optimizer_attributes – Function.

```
set_optimizer_attributes(
    model::Union{GenericModel,MOI.OptimizerWithAttributes},
    pairs::Pair...,
)
```

Given a list of attribute => value pairs, calls set_optimizer_attribute(model, attribute, value)
for each pair.

> **Compat**
>
> This method will remain in all v1.X releases of JuMP, but it may be removed in a future v2.0 release.
> We recommend using set_attributes instead.

See also: set_optimizer_attribute, get_optimizer_attribute.

Example

```julia
julia> import Ipopt

julia> model = Model(Ipopt.Optimizer);

julia> set_optimizer_attributes(model, "tol" => 1e-4, "max_iter" => 100)
```

is equivalent to:

```julia
julia> import Ipopt

julia> model = Model(Ipopt.Optimizer);

julia> set_optimizer_attribute(model, "tol", 1e-4)

julia> set_optimizer_attribute(model, "max_iter", 100)
```

source

20.5 Copying

JuMP.GenericReferenceMap – Type.

```
GenericReferenceMap{T}
```

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::GenericModel; filter_constraints::Union{Nothing, Function}=nothing)
```

Return a copy of the model model and a GenericReferenceMap 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 GenericModel 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.

Example

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.

```julia
julia> model = Model();

julia> @variable(model, x)
x

julia> @constraint(model, cref, x == 2)
cref : x = 2.0

julia> new_model, reference_map = copy_model(model);

julia> x_new = reference_map[x]
x

julia> cref_new = reference_map[cref]
cref : x = 2.0
```

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 GenericModel 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.

Example

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.

```julia
julia> model = Model();

julia> @variable(model, x)
x

julia> @constraint(model, cref, x == 2)
cref : x = 2.0

julia> new_model = copy(model);

julia> x_new = model[:x]
x

julia> cref_new = model[:cref]
cref : x = 2.0
```

source

20.6 I/O

JuMP.write_to_file – Function.

```
write_to_file(
    model::GenericModel,
    filename::String;
    format::MOI.FileFormats.FileFormat = MOI.FileFormats.FORMAT_AUTOMATIC,
    kwargs...,
)
```

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.

Other kwargs are passed to the Model constructor of the chosen format.

source

Base.write – Method.

```
Base.write(
    io::IO,
    model::GenericModel;
    format::MOI.FileFormats.FileFormat = MOI.FileFormats.FORMAT_MOF,
    kwargs...,
)
```

Write the JuMP model model to io in the format format.

Other kwargs are passed to the Model constructor of the chosen format.

source

JuMP.read_from_file – Function.

```
read_from_file(
    filename::String;
    format::MOI.FileFormats.FileFormat = MOI.FileFormats.FORMAT_AUTOMATIC,
    kwargs...,
)
```

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.

Other kwargs are passed to the Model constructor of the chosen format.

source

Base.read – Method.

```
Base.read(
    io::IO,
    ::Type{<:GenericModel};
    format::MOI.FileFormats.FileFormat,
    kwargs...,
)
```

Return a JuMP model read from io in the format format.

Other kwargs are passed to the Model constructor of the chosen format.

source

20.7 Caching Optimizer

MathOptInterface.Utilities.reset_optimizer – Method.

```
MOIU.reset_optimizer(model::GenericModel)
```

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::GenericModel)
```

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::GenericModel)
```

Call `MOIU.attach_optimizer` on the backend of model.

Cannot be called in direct mode.

source

20.8   Bridge tools

`JuMP.add_bridge` – Function.

```
add_bridge(model::AbstractModel, BT::Type{<:MOI.Bridges.AbstractBridge})
```

Add BT to the list of bridges that can be used to transform unsupported constraints into an equivalent formulation using only constraints supported by the optimizer.

See also: remove_bridge.

source

`JuMP.remove_bridge` – Function.

```
remove_bridge(model::GenericModel, BT::Type{<:MOI.Bridges.AbstractBridge})
```

Remove BT to the list of bridges that can be used to transform unsupported constraints into an equivalent formulation using only constraints supported by the optimizer.

See also: add_bridge.

source

`JuMP.bridge_constraints` – Function.

```
bridge_constraints(model::GenericModel)
```

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_active_bridges` – Function.

```
print_active_bridges([io::IO = stdout,] model::GenericModel)
```

Print a list of the variable, constraint, and objective bridges that are currently used in the model.

source

```
print_active_bridges([io::IO = stdout,] model::GenericModel, ::Type{F}) where {F}
```

Print a list of bridges required for an objective function of type F.

source

```
print_active_bridges(
    [io::IO = stdout,]
    model::GenericModel,
    F::Type,
    S::Type{<:MOI.AbstractSet},
)
```

Print a list of bridges required for a constraint of type F-in-S.

source

```
print_active_bridges(
    [io::IO = stdout,]
    model::GenericModel,
    S::Type{<:MOI.AbstractSet},
)
```

Print a list of bridges required to add a variable constrained to the set S.

source

JuMP.`print_bridge_graph` – Function.

```
print_bridge_graph([io::IO,] model::GenericModel)
```

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.

> **Warning**
>
> This function is intended for advanced users. If you want to see only the bridges that are currently used, use `print_active_bridges` instead.

Explanation of output

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

20.9   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::GenericModel)
```

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::GenericModel, 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

JuMP.set_optimize_hook – Function.

```
set_optimize_hook(model::GenericModel, f::Union{Function,Nothing})
```

Set the function f as the optimize hook for model.

f should have a signature f(model::GenericModel; kwargs...), where the kwargs are those passed to optimize!.

Notes

- The optimize hook should generally modify the model, or some external state in some way, and then call optimize!(model; ignore_optimize_hook = true) to optimize the problem, bypassing the hook.

- Use set_optimize_hook(model, nothing) to unset an optimize hook.

Example

```
julia> model = Model();

julia> function my_hook(model::Model; kwargs...)
           println(kwargs)
           println("Calling with `ignore_optimize_hook = true`")
           optimize!(model; ignore_optimize_hook = true)
           return
       end
my_hook (generic function with 1 method)

julia> set_optimize_hook(model, my_hook)
my_hook (generic function with 1 method)

julia> optimize!(model; test_arg = true)
Base.Iterators.Pairs{Symbol, Bool, Tuple{Symbol}, NamedTuple{(:test_arg,),
↪  Tuple{Bool}}}(:test_arg => 1)
Calling with `ignore_optimize_hook = true`
ERROR: NoOptimizer()
Stacktrace:
[...]
```

source

Chapter 21

Variables

More information can be found in the section of the manual.

21.1 Macros

JuMP.@variable – Macro.

```
@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`.

Anonymous and named variables

`expr` must be one of the forms:

- Omitted, like `@variable(model)`, which creates an anonymous variable
- A single symbol like `@variable(model, x)`
- A container expression like `@variable(model, x[i=1:3])`
- An anoymous container expression like `@variable(model, [i=1:3])`

Bounds

In addition, the expression can have bounds, such as:

- `@variable(model, x >= 0)`
- `@variable(model, x <= 0)`
- `@variable(model, x == 0)`
- `@variable(model, 0 <= x <= 1)`

and bounds can depend on the indices of the container expressions:

- `@variable(model, -i <= x[i=1:3] <= i)`

Sets

You can explicitly specify the set to which the variable belongs:

- `@variable(model, x in MOI.Interval(0.0, 1.0))`

For more information on this syntax, read Variables constrained on creation.

Positional arguments

The recognized positional arguments in `args` are the following:

- `Bin`: restricts the variable to the `MOI.ZeroOne` set, that is, {0, 1}. For example, `@variable(model, x, Bin)`. Note: you cannot use `@variable(model, Bin)`, use the `binary` keyword instead.
- `Int`: restricts the variable to the set of integers, that is, ..., -2, -1, 0, 1, 2, ... For example, `@variable(model, x, Int)`. Note: you cannot use `@variable(model, Int)`, use the `integer` keyword instead.
- `Symmetric`: Only available when creating a square matrix of variables, i.e., when `expr` is of the form `varname[1:n,1:n]` or `varname[i=1:n,j=1:n]`, it creates a symmetric matrix of variables.
- `PSD`: A restrictive extension to `Symmetric` which constraints a square matrix of variables to `Symmetric` and constrains to be positive semidefinite.

Keyword arguments

Four keyword arguments are useful in all cases:

- `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.
- `start::Float64`: specify the value passed to `set_start_value` for each variable
- `container`: specify the container type. See Forcing the container type for more information.
- `set_string_name::Bool = true`: control whether to set the `MOI.VariableName` attribute. Passing `set_string_name = false` can improve performance.

Other keyword arguments are needed to disambiguate sitations with anonymous variables:

- `lower_bound::Float64`: an alternative to `x >= lb`, sets the value of the variable lower bound.
- `upper_bound::Float64`: an alternative to `x <= ub`, sets the value of the variable upper bound.
- `binary::Bool`: an alternative to passing `Bin`, sets whether the variable is binary or not.
- `integer::Bool`: an alternative to passing `Int`, sets whether the variable is integer or not.
- `set::MOI.AbstractSet`: an alternative to using `x in set`
- `variable_type`: used by JuMP extensions. See Extend @variable for more information.

Example

The following are equivalent ways of creating a variable x of name x with lower bound 0:

```julia
julia> model = Model();

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

```
julia> model = Model();

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

```
julia> model = Model();

julia> x = @variable(model, base_name = "x", lower_bound = 0)
x
```

Other examples:

```
julia> model = Model();

julia> @variable(model, x[i=1:3] <= i, Int, start = sqrt(i), lower_bound = -i)
3-element Vector{VariableRef}:
 x[1]
 x[2]
 x[3]

julia> @variable(model, y[i=1:3], container = DenseAxisArray, set = MOI.ZeroOne())
1-dimensional DenseAxisArray{VariableRef,1,...} with index sets:
    Dimension 1, Base.OneTo(3)
And data, a 3-element Vector{VariableRef}:
 y[1]
 y[2]
 y[3]

julia> @variable(model, z[i=1:3], set_string_name = false)
3-element Vector{VariableRef}:
 _[7]
 _[8]
 _[9]
```

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.

The macro returns a tuple containing the variables that were defined.

Example

```julia
julia> model = Model();

julia> @variables(model, begin
           x
           y[i = 1:2] >= 0, (start = i)
           z, Bin, (start = 0, base_name = "Z")
       end)
(x, VariableRef[y[1], y[2]], Z)
```

> **Note**
>
> Keyword arguments must be contained within parentheses (refer to the example above).

source

21.2 Basic utilities

JuMP.VariableRef – Type.

```
GenericVariableRef <: AbstractVariableRef
```

Holds a reference to the model and the corresponding MOI.VariableIndex.

source

JuMP.GenericVariableRef – Type.

```
GenericVariableRef <: AbstractVariableRef
```

Holds a reference to the model and the corresponding MOI.VariableIndex.

source

JuMP.num_variables – Function.

```
num_variables(model::GenericModel)::Int64
```

Returns number of variables in model.

source

JuMP.all_variables – Function.

```
all_variables(model::GenericModel{T})::Vector{GenericVariableRef{T}} where {T}
```

Returns a list of all variables currently in the model. The variables are ordered by creation time.

Example

```julia
julia> model = Model();

julia> @variable(model, x);

julia> @variable(model, y);

julia> all_variables(model)
2-element Vector{VariableRef}:
 x
 y
```

source

JuMP.owner_model – Function.

```
owner_model(s::AbstractJuMPScalar)
```

Return the model owning the scalar s.

source

JuMP.index – Method.

```
index(v::GenericVariableRef)::MOI.VariableIndex
```

Return the index of the variable that corresponds to v in the MOI backend.

source

JuMP.optimizer_index – Method.

```
optimizer_index(x::GenericVariableRef)::MOI.VariableIndex
optimizer_index(x::ConstraintRef{<:GenericModel})::MOI.ConstraintIndex
```

Return the index that corresponds to x in the optimizer model.

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::GenericModel,
    variable::VariableConstrainedOnCreation,
    names,
)
    var_ref = add_variable(model, variable.scalar_variable, name)
    add_constraint(model, 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:

```julia
function JuMP.add_variable(
    model::GenericModel,
    variable::VariablesConstrainedOnCreation,
    names,
)
    v_names = vectorize(names, variable.shape)
    var_refs = add_variable.(model, variable.scalar_variables, v_names)
    add_constraint(model, VectorConstraint(var_refs, variable.set))
    return 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

`JuMP.ComplexPlane` – Type.

```
ComplexPlane
```

Complex plane object that can be used to create a complex variable in the `@variable` macro.

Example

Consider the following example:

```julia
julia> model = Model();

julia> @variable(model, x in ComplexPlane())
real(x) + (0.0 + 1.0im) imag(x)

julia> all_variables(model)
2-element Vector{VariableRef}:
 real(x)
 imag(x)
```

We see in the output of the last command that two real variables were created. The Julia variable x binds to an affine expression in terms of these two variables that parametrize the complex plane.

source

`JuMP.ComplexVariable` – Type.

```
ComplexVariable{S,T,U,V} <: AbstractVariable
```

A struct used when adding complex variables.

See also: ComplexPlane.

source

21.3 Names

JuMP.name – Method.

```
name(v::GenericVariableRef)::String
```

Get a variable's name attribute.

source

JuMP.set_name – Method.

```
set_name(v::GenericVariableRef, 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.

Examples

```
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(::GenericModel, ::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

21.4   Start values

JuMP.`has_start_value` – Function.

```
has_start_value(variable::AbstractVariableRef)
```

Return `true` if the variable has a start value set otherwise return `false`.

See also set_start_value.

source

JuMP.`set_start_value` – Function.

```
set_start_value(con_ref::ConstraintRef, value)
```

Set the primal start value (`MOI.ConstraintPrimalStart`) of the constraint con_ref to value. To remove a primal start value set it to `nothing`.

See also start_value.

source

```
set_start_value(variable::GenericVariableRef, 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(con_ref::ConstraintRef)
```

Return the primal start value (MOI.ConstraintPrimalStart) of the constraint con_ref.

Note: If no primal start value has been set, start_value will return nothing.

See also set_start_value.

source

```
start_value(v::GenericVariableRef)
```

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

21.5 Lower bounds

JuMP.has_lower_bound – Function.

```
has_lower_bound(v::GenericVariableRef)
```

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::GenericVariableRef)
```

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::GenericVariableRef, 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::GenericVariableRef)
```

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::GenericVariableRef)
```

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

21.6 Upper bounds

JuMP.has_upper_bound – Function.

```
has_upper_bound(v::GenericVariableRef)
```

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::GenericVariableRef)
```

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::GenericVariableRef, 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::GenericVariableRef)
```

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::GenericVariableRef)
```

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

21.7 Fixed bounds

JuMP.is_fixed – Function.

```
is_fixed(v::GenericVariableRef)
```

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::GenericVariableRef)
```

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::GenericVariableRef, 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::GenericVariableRef)
```

Delete the fixing constraint of a variable.

See also FixRef, is_fixed, fix_value, fix.

source

JuMP.FixRef – Function.

```
FixRef(v::GenericVariableRef)
```

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

21.8   Integer variables

JuMP.is_integer – Function.

```
is_integer(v::GenericVariableRef)
```

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::GenericVariableRef)
```

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::GenericVariableRef)
```

Remove the integrality constraint on the variable `variable_ref`.

See also IntegerRef, is_integer, set_integer.

source

JuMP.IntegerRef – Function.

```
IntegerRef(v::GenericVariableRef)
```

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

21.9   Binary variables

JuMP.is_binary – Function.

```
is_binary(v::GenericVariableRef)
```

Return `true` if v is constrained to be binary.

See also BinaryRef, set_binary, unset_binary.

source

JuMP.set_binary – Function.

```
set_binary(v::GenericVariableRef)
```

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::GenericVariableRef)
```

Remove the binary constraint on the variable `variable_ref`.

See also BinaryRef, is_binary, set_binary.

source

JuMP.BinaryRef – Function.

```
BinaryRef(v::GenericVariableRef)
```

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

21.10   Integrality utilities

JuMP.relax_integrality – Function.

```
relax_integrality(model::GenericModel)
```

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

JuMP.fix_discrete_variables – Function.

```
fix_discrete_variables([var_value::Function = value,] model::GenericModel)
```

Modifies model to convert all binary and integer variables to continuous variables with fixed bounds of var_value(x).

Return

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.

Notes

- 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.

Example

```
julia> model = Model();

julia> @variable(model, x, Bin, start = 1);

julia> @variable(model, 1 <= y <= 10, Int, start = 2);

julia> @objective(model, Min, x + y);

julia> undo_relax = fix_discrete_variables(start_value, model);

julia> print(model)
```

```
Min x + y
Subject to
 x = 1.0
 y = 2.0

julia> undo_relax()

julia> print(model)
Min x + y
Subject to
 y ≥ 1.0
 y ≤ 10.0
 y integer
 x binary
```

source

21.11 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 22

Expressions

More information can be found in the Expressions section of the manual.

22.1 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.

Example

```julia
julia> model = Model();

julia> @variable(model, x[1:5]);

julia> @variable(model, y);

julia> @variable(model, z);

julia> @expression(model, shared, sum(i * x[i] for i in 1:5))
x[1] + 2 x[2] + 3 x[3] + 4 x[4] + 5 x[5]

julia> @constraint(model, shared + y >= 5)
x[1] + 2 x[2] + 3 x[3] + 4 x[4] + 5 x[5] + y ≥ 5.0

julia> @constraint(model, shared + z <= 10)
x[1] + 2 x[2] + 3 x[3] + 4 x[4] + 5 x[5] + z ≤ 10.0
```

The ref accepts index sets in the same way as @variable, and those indices can be used in the construction of the expressions:

```julia
julia> @expression(model, expr[i = 1:3], i * sum(x[j] for j in 1:3))
3-element Vector{AffExpr}:
 x[1] + x[2] + x[3]
```

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

Anonymous syntax is also supported:

```
julia> expr = @expression(model, [i in 1:3], i * sum(x[j] for j in 1:3))
3-element Vector{AffExpr}:
 x[1] + x[2] + x[3]
 2 x[1] + 2 x[2] + 2 x[3]
 3 x[1] + 3 x[2] + 3 x[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 expressions can be added on multiple lines wrapped in a begin ... end block.

The macro returns a tuple containing the expressions that were defined.

Example

```
julia> model = Model();

julia> @variable(model, x);

julia> @variable(model, y);

julia> @variable(model, z[1:2]);

julia> a = [4, 5];

julia> @expressions(model, begin
           my_expr, x^2 + y^2
           my_expr_1[i = 1:2], a[i] - z[i]
       end)
(x² + y², AffExpr[-z[1] + 4, -z[2] + 5])
```

source

22.2 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

22.3 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: $\sum q_{i,j}x_ix_j + \sum a_ix_i + c$.

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

22.4 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::GenericVariableRef{T}, v2::GenericVariableRef{T}) where {T}
```

Return `one(T)` if `v1 == v2`, and `zero(T)` 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, ::GenericVariableRef, ::GenericVariableRef) 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
julia> model = Model();

julia> @variable(model, x);

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> model = Model();

julia> @variable(model, x);

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> model = Model();

julia> @variable(model, x);

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> model = Model();

julia> @variable(model, x);

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

22.5 JuMP-to-MOI converters

JuMP.variable_ref_type – Function.

```
variable_ref_type(::Union{F,Type{F}}) where {F}
```

A helper function used internally by JuMP and some JuMP extensions. Returns the variable type associated with the model or expression type F.

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 23

Objectives

More information can be found in the Objectives section of the manual.

23.1 Objective functions

JuMP.@objective – Macro.

```
@objective(model::GenericModel, sense, func)
```

Set the objective sense to `sense` and objective function to `func`. The objective sense can be either Min, Max, MOI.MIN_SENSE, MOI.MAX_SENSE or MOI.FEASIBILITY_SENSE; see MOI.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 MOI.ObjectiveSense values should be used.

Example

To minimize the value of the variable x, do as follows:

```
julia> model = Model();

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 = 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::GenericModel,
    T::Type = objective_function_type(model),
)
```

Return an object of type T representing the objective function.

Error if the objective is not convertible to type T.

Example

```
julia> model = Model();

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::GenericModel, func::MOI.AbstractFunction)
set_objective_function(model::GenericModel, func::AbstractJuMPScalar)
set_objective_function(model::GenericModel, func::Real)
set_objective_function(model::GenericModel, func::Vector{<:AbstractJuMPScalar})
```

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::GenericModel, variable::GenericVariableRef, 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 calling set_objective_sense and set_objective_function separately.

Example

```
julia> model = Model();

julia> @variable(model, x)
x

julia> set_objective(model, MIN_SENSE, x)
```

source

JuMP.objective_function_type – Function.

```
objective_function_type(model::GenericModel)::AbstractJuMPScalar
```

Return the type of the objective function.

source

JuMP.objective_function_string – Function.

```
objective_function_string(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

23.2 Objective sense

JuMP.objective_sense – Function.

```
objective_sense(model::GenericModel)::MOI.OptimizationSense
```

Return the objective sense.

source

JuMP.set_objective_sense – Function.

```
set_objective_sense(model::GenericModel, sense::MOI.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 24

Constraints

More information can be found in the Constraints section of the manual.

24.1 Macros

JuMP.@constraint – Macro.

```
@constraint(m::GenericModel, expr, kw_args...)
```

Add a constraint described by the expression expr.

```
@constraint(m::GenericModel, 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.

- `set_string_name::Bool = true`: control whether to set the MOI.ConstraintName attribute. Passing `set_string_name = false` can improve performance.

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.

The macro returns a tuple containing the constraints that were defined.

Example

```julia
julia> model = Model();

julia> @variable(model, w);

julia> @variable(model, x);

julia> @variable(model, y);

julia> @variable(model, z[1:3]);

julia> @constraints(model, begin
```

```
            x >= 1
            y - w <= 2
            sum_to_one[i=1:3], z[i] + y == 1
       end);

julia> print(model)
Feasibility
Subject to
 sum_to_one[1] : y + z[1] = 1.0
 sum_to_one[2] : y + z[2] = 1.0
 sum_to_one[3] : y + z[3] = 1.0
 x ≥ 1.0
 -w + y ≤ 2.0
```

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 ConstraintRefs 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

24.2 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> model = Model();

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

24.3 Modification

JuMP.normalized_coefficient – Function.

```
normalized_coefficient(con_ref::ConstraintRef, variable::GenericVariableRef)
```

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::GenericVariableRef, 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 $2x + 3x \leq 2$, set_normalized_coefficient(con, x, 4) will create the constraint $4x \leq 2$.

```
julia> model = Model();

julia> @variable(model, x)
x

julia> @constraint(model, con, 2x + 3x <= 2)
con : 5 x ≤ 2.0

julia> set_normalized_coefficient(con, x, 4)

julia> con
con : 4 x ≤ 2.0
```

source

JuMP.set_normalized_coefficients – Function.

```
set_normalized_coefficients(
    con_ref::ConstraintRef,
    variable,
    new_coefficients::Vector{Tuple{Int64,T}},
)
```

Set the coefficients of variable in the constraint con_ref to new_coefficients, where each element in new_coefficients is a tuple which maps the row to a new coefficient.

Note that prior to this step, during constraint creation, JuMP will aggregate multiple terms containing the same variable.

```
julia> model = Model();

julia> @variable(model, x)
x

julia> @constraint(model, con, [2x + 3x, 4x] in MOI.Nonnegatives(2))
con : [5 x, 4 x] ∈ MathOptInterface.Nonnegatives(2)

julia> set_normalized_coefficients(con, x, [(1, 2.0), (2, 5.0)])

julia> con
con : [2 x, 5 x] ∈ MathOptInterface.Nonnegatives(2)
```

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> model = Model();

julia> @variable(model, x);

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.

Example

For scalar constraints, the set is translated by -value:

```
julia> model = Model();

julia> @variable(model, x);

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> model = Model();

julia> @variable(model, x);

julia> @variable(model, y);

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

JuMP.relax_with_penalty! – Function.

```
relax_with_penalty!(
    model::GenericModel{T},
    [penalties::Dict{ConstraintRef,T}];
    [default::Union{Nothing,Real} = nothing,]
) where {T}
```

Destructively modify the model in-place to create a penalized relaxation of the constraints.

> **Warning**
>
> This is a destructive routine that modifies the model in-place. If you don't want to modify the original model, use copy_model to create a copy before calling relax_with_penalty!.

Reformulation

See MOI.Utilities.ScalarPenaltyRelaxation for details of the reformulation.

For each constraint ci, the penalty passed to MOI.Utilities.ScalarPenaltyRelaxation is get(penalties, ci, default). If the value is nothing, because ci does not exist in penalties and default = nothing, then the constraint is skipped.

Return value

This function returns a Dict{ConstraintRef,AffExpr} that maps each constraint index to the corresponding y + z as an AffExpr. In an optimal solution, query the value of these functions to compute the violation of each constraint.

Relax a subset of constraints

To relax a subset of constraints, pass a penalties dictionary and set default = nothing.

Example

```
julia> function new_model()
           model = Model()
           @variable(model, x)
           @objective(model, Max, 2x + 1)
           @constraint(model, c1, 2x - 1 <= -2)
           @constraint(model, c2, 3x >= 0)
           return model
       end
new_model (generic function with 1 method)

julia> model_1 = new_model();

julia> relax_with_penalty!(model_1; default = 2.0)
```

```
Dict{ConstraintRef{Model, C, ScalarShape} where C, AffExpr} with 2 entries:
  c1 : 2 x - _[3] ≤ -1.0 => _[3]
  c2 : 3 x + _[2] ≥ 0.0  => _[2]

julia> print(model_1)
Max 2 x - 2 _[2] - 2 _[3] + 1
Subject to
 c2 : 3 x + _[2] ≥ 0.0
 c1 : 2 x - _[3] ≤ -1.0
 _[2] ≥ 0.0
 _[3] ≥ 0.0

julia> model_2 = new_model();

julia> relax_with_penalty!(model_2, Dict(model_2[:c2] => 3.0))
Dict{ConstraintRef{Model,
↪  MathOptInterface.ConstraintIndex{MathOptInterface.ScalarAffineFunction{Float64},
↪  MathOptInterface.GreaterThan{Float64}}, ScalarShape}, AffExpr} with 1 entry:
  c2 : 3 x + _[2] ≥ 0.0 => _[2]

julia> print(model_2)
Max 2 x - 3 _[2] + 1
Subject to
 c2 : 3 x + _[2] ≥ 0.0
 c1 : 2 x ≤ -1.0
 _[2] ≥ 0.0
```

source

24.4 Deletion

JuMP.delete – Function.

```
delete(model::GenericModel, 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.

Example

```
julia> model = Model();

julia> @variable(model, x);

julia> @constraint(model, c, 2x <= 1)
c : 2 x ≤ 1.0

julia> delete(model, c)

julia> unregister(model, :c)
```

```
julia> print(model)
Feasibility
Subject to

julia> model[:c]
ERROR: KeyError: key :c not found
Stacktrace:
[...]
```

source

```
delete(model::GenericModel, 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::GenericModel, variable_ref::GenericVariableRef)
```

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.

Example

```
julia> model = Model();

julia> @variable(model, x)
x

julia> delete(model, x)

julia> unregister(model, :x)

julia> print(model)
Feasibility
Subject to

julia> model[:x]
ERROR: KeyError: key :x not found
Stacktrace:
[...]
```

source

```
delete(model::GenericModel, variable_refs::Vector{<:GenericVariableRef})
```

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

```
delete(model::Model, c::NonlinearConstraintRef)
```

Delete the nonlinear constraint c from `model`.

source

JuMP.is_valid – Function.

```
is_valid(model::GenericModel, con_ref::ConstraintRef{<:AbstractModel})
```

Return `true` if `constraint_ref` refers to a valid constraint in `model`.

source

```
is_valid(model::GenericModel, variable_ref::GenericVariableRef)
```

Return `true` if `variable` refers to a valid variable in `model`.

source

```
is_valid(model::Model, c::NonlinearConstraintRef)
```

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

24.5 Query constraints

JuMP.list_of_constraint_types – Function.

```
list_of_constraint_types(model::GenericModel)::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
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)
```

Performance considerations

Iterating over the list of function and set types is a type-unstable operation. Consider using a function barrier. See the Performance tips for extensions section of the documentation for more details.

source

JuMP.all_constraints – Function.

```
all_constraints(model::GenericModel, 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
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
```

```
all_constraints(
    model::GenericModel;
    include_variable_in_set_constraints::Bool,
)::Vector{ConstraintRef}
```

Return a list of all constraints in model.

If include_variable_in_set_constraints == true, then VariableRef constraints such as VariableRef-in-Integer are included. To return only the structural constraints (e.g., the rows in the constraint matrix of a linear program), pass include_variable_in_set_constraints = false.

Example

```
julia> model = Model();

julia> @variable(model, x >= 0, Int);

julia> @constraint(model, 2x <= 1);

julia> @NLconstraint(model, x^2 <= 1);

julia> all_constraints(model; include_variable_in_set_constraints = true)
4-element Vector{ConstraintRef}:
 2 x ≤ 1.0
 x ≥ 0.0
 x integer
 x ^ 2.0 - 1.0 ≤ 0

julia> all_constraints(model; include_variable_in_set_constraints = false)
2-element Vector{ConstraintRef}:
 2 x ≤ 1.0
 x ^ 2.0 - 1.0 ≤ 0
```

Performance considerations

Note that this function is type-unstable because it returns an abstractly typed vector. If performance is a problem, consider using list_of_constraint_types and a function barrier. See the Performance tips for extensions section of the documentation for more details.

JuMP.num_constraints – Function.

```
num_constraints(model::GenericModel, 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
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

```
num_constraints(model::GenericModel; count_variable_in_set_constraints::Bool)
```

Return the number of constraints in model.

If count_variable_in_set_constraints == true, then VariableRef constraints such as VariableRef-in-Integer are included. To count only the number of structural constraints (e.g., the rows in the constraint matrix of a linear program), pass count_variable_in_set_constraints = false.

Example

```julia
julia> model = Model();

julia> @variable(model, x >= 0, Int);

julia> @constraint(model, 2x <= 1);

julia> num_constraints(model; count_variable_in_set_constraints = true)
3

julia> num_constraints(model; count_variable_in_set_constraints = false)
1
```

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(x::GenericVariableRef)::MOI.VariableIndex
optimizer_index(x::ConstraintRef{<:GenericModel})::MOI.ConstraintIndex
```

Return the index that corresponds to x in the optimizer model.

Throws NoOptimizer if no optimizer is set, and throws an ErrorException if the optimizer is set but is not attached.

source

JuMP.constraint_object – Function.

```
constraint_object(con_ref::ConstraintRef)
```

Return the underlying constraint data for the constraint referenced by ref.

source

24.6 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

24.7 Special sets

JuMP.Nonnegatives – Type.

```
Nonnegatives()
```

The JuMP equivalent of the MOI.Nonnegatives set, in which the dimension is inferred from the corresponding function.

Example

```julia
julia> model = Model();

julia> @variable(model, x[1:2])
2-element Vector{VariableRef}:
 x[1]
 x[2]

julia> @constraint(model, x in Nonnegatives())
[x[1], x[2]] ∈ MathOptInterface.Nonnegatives(2)

julia> A = [1 2; 3 4];

julia> b = [5, 6];

julia> @constraint(model, A * x >= b)
[x[1] + 2 x[2] - 5, 3 x[1] + 4 x[2] - 6] ∈ MathOptInterface.Nonnegatives(2)
```

source

JuMP.Nonpositives – Type.

```
Nonpositives()
```

The JuMP equivalent of the MOI.Nonpositives set, in which the dimension is inferred from the corresponding function.

Example

```julia
julia> model = Model();

julia> @variable(model, x[1:2])
2-element Vector{VariableRef}:
    x[1]
    x[2]

julia> @constraint(model, x in Nonpositives())
[x[1], x[2]] ∈ MathOptInterface.Nonpositives(2)

julia> A = [1 2; 3 4];
```

```
julia> b = [5, 6];

julia> @constraint(model, A * x <= b)
[x[1] + 2 x[2] - 5, 3 x[1] + 4 x[2] - 6] ∈ MathOptInterface.Nonpositives(2)
```

source

JuMP.Zeros – Type.

```
Zeros()
```

The JuMP equivalent of the MOI.Zeros set, in which the dimension is inferred from the corresponding function.

Example

```
julia> model = Model();

julia> @variable(model, x[1:2])
2-element Vector{VariableRef}:
    x[1]
    x[2]

julia> @constraint(model, x in Zeros())
[x[1], x[2]] ∈ MathOptInterface.Zeros(2)

julia> A = [1 2; 3 4];

julia> b = [5, 6];

julia> @constraint(model, A * x == b)
[x[1] + 2 x[2] - 5, 3 x[1] + 4 x[2] - 6] ∈ MathOptInterface.Zeros(2)
```

source

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.

Example

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.

Example

The following constrains $\|(x - 1, x - 2)\|_2^2 \le 2tx$ and $t, x \ge 0$:

```
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.

Example

Consider the following example:

```julia
julia> model = Model();

julia> @variable(model, x)
x

julia> a = [ x 2x
             2x  x];

julia> b = [1 2
            2 4];

julia> cref = @constraint(model, a >= b, PSDCone())
[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 vectorization of the matrix is constrained to belong to the PositiveSemidefiniteConeSquare.

```julia
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.HermitianPSDCone – Type.

```
HermitianPSDCone
```

Hermitian positive semidefinite cone object that can be used to create a Hermitian positive semidefinite square matrix in the @variable and @constraint macros.

Example

Consider the following example:

```julia
julia> model = Model();

julia> @variable(model, H[1:3, 1:3] in HermitianPSDCone())
3×3 Matrix{GenericAffExpr{ComplexF64, VariableRef}}:
 real(H[1,1])                               real(H[1,2]) + (0.0 + 1.0im) imag(H[1,2])
↪   real(H[1,3]) + (0.0 + 1.0im) imag(H[1,3])
 real(H[1,2]) + (-0.0 - 1.0im) imag(H[1,2])  real(H[2,2])
↪   real(H[2,3]) + (0.0 + 1.0im) imag(H[2,3])
 real(H[1,3]) + (-0.0 - 1.0im) imag(H[1,3])  real(H[2,3]) + (-0.0 - 1.0im) imag(H[2,3])
↪   real(H[3,3])

 julia> v = all_variables(model)
 9-element Vector{VariableRef}:
  real(H[1,1])
  real(H[1,2])
  real(H[2,2])
  real(H[1,3])
  real(H[2,3])
  real(H[3,3])
  imag(H[1,2])
  imag(H[1,3])
  imag(H[2,3])

julia> all_constraints(model, Vector{VariableRef},
↪   MOI.HermitianPositiveSemidefiniteConeTriangle)
1-element Vector{ConstraintRef{Model,
↪   MathOptInterface.ConstraintIndex{MathOptInterface.VectorOfVariables,
↪   MathOptInterface.HermitianPositiveSemidefiniteConeTriangle}}}:
 [real(H[1,1]), real(H[1,2]), real(H[2,2]), real(H[1,3]), real(H[2,3]), real(H[3,3]),
↪   imag(H[1,2]), imag(H[1,3]), imag(H[2,3])] in
↪   MathOptInterface.HermitianPositiveSemidefiniteConeTriangle(3)
```

We see in the output of the last commands that 9 real variables were created. The matrix H contrains affine expressions in terms of these 9 variables that parametrize a Hermitian matrix.

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.

Example

```julia
julia> model = Model();

julia> @variable(model, Q[1:2, 1:2] in SkewSymmetricMatrixSpace())
2×2 Matrix{AffExpr}:
 0       Q[1,2]
 -Q[1,2]  0
```

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.SymmetricMatrixSpace – Type.

```
SymmetricMatrixSpace()
```

Use in the @variable macro to constrain a matrix of variables to be symmetric.

Example

```
julia> model = Model();

julia> @variable(model, Q[1:2, 1:2] in SymmetricMatrixSpace())
2×2 LinearAlgebra.Symmetric{VariableRef,Array{VariableRef,2}}:
 Q[1,1]  Q[1,2]
 Q[1,2]  Q[2,2]
```

source

JuMP.HermitianMatrixSpace – Type.

```
HermitianMatrixSpace()
```

Use in the @variable macro to constrain a matrix of variables to be hermitian.

Example

```
julia> model = Model();

julia> @variable(model, Q[1:2, 1:2] in HermitianMatrixSpace())
2×2 LinearAlgebra.Hermitian{GenericAffExpr{ComplexF64, VariableRef},
↪  Matrix{GenericAffExpr{ComplexF64, VariableRef}}}:
 real(Q[1,1])                           real(Q[1,2]) + (0.0 + 1.0im) imag(Q[1,2])
 real(Q[1,2]) + (0.0 - 1.0im) imag(Q[1,2])  real(Q[2,2])
```

source

JuMP.HermitianMatrixShape – Type.

```
HermitianMatrixShape
```

Shape object for a Hermitian square matrix of side_dimension rows and columns. The vectorized form corresponds to MOI.HermitianPositiveSemidefiniteConeTriangle.

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.

```
moi_set(s::AbstractScalarSet)
```

Returns the MOI set corresponding to the JuMP set s.

source

JuMP.Semicontinuous – Type.

```
Semicontinuous(lower, upper)
```

A short-cut for the MOI.Semicontinuous set.

This short-cut is useful because it automatically promotes lower and upper to the same type, and converts them into the element type supported by the JuMP model.

Example

```julia
julia> model = Model();

julia> @variable(model, x in Semicontinuous(1, 2))
x

julia> print(model)
Feasibility
Subject to
 x ∈ MathOptInterface.Semicontinuous{Float64}(1.0, 2.0)
```

source

JuMP.Semiinteger – Type.

```
Semiinteger(lower, upper)
```

A short-cut for the MOI.Semiinteger set.

This short-cut is useful because it automatically promotes lower and upper to the same type, and converts them into the element type supported by the JuMP model.

Example

```julia
julia> model = Model();

julia> @variable(model, x in Semiinteger(3, 5))
x

julia> print(model)
Feasibility
Subject to
 x ∈ MathOptInterface.Semiinteger{Float64}(3.0, 5.0)
```

source

24.8 Printing

JuMP.function_string – Function.

```
function_string(
    mode::MIME,
    func::Union{JuMP.AbstractJuMPScalar,Vector{<:JuMP.AbstractJuMPScalar}},
)
```

Return a `String` representing the function `func` using print mode mode.

source

JuMP.constraints_string – Function.

```
constraints_string(mode, model::AbstractModel)::Vector{String}
```

Return a list of `Strings` describing each constraint of the model.

source

JuMP.in_set_string – Function.

```
in_set_string(mode::MIME, set)
```

Return a `String` representing the membership to the set set using print mode 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 25

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
julia> array = 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 = 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.

Example

```
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 = 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} = AutoContainerType], names])
```

Create a container of type C with index names names, indices `indices` and values at given indices given by f.

If the method with names is not specialized on Type{C}, it falls back to calling container(f, indices, c) for backwards compatibility with containers not supporting index names.

Example

```
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
```

JuMP.Containers.rowtable – Function.

```
rowtable([f::Function=identity,] x; [header::Vector{Symbol} = Symbol[]])
```

Applies the function f to all elements of the variable container x, returning the result as a Vector of NamedTuples, where header is a vector containing the corresponding axis names.

If x is an N-dimensional array, there must be N+1 names, so that the last name corresponds to the result of f(x[i]).

If header is left empty, then the default header is [:x1, :x2, ..., :xN, :y].

> **Info**
>
> A Vector of NamedTuples implements the Tables.jl interface, and so the result can be used as input for any function that consumes a 'Tables.jl' compatible source.

Example

```julia
julia> model = Model();

julia> @variable(model, x[i=1:2, j=i:2] >= 0, start = i+j);

julia> Containers.rowtable(start_value, x; header = [:i, :j, :start])
3-element Vector{NamedTuple{(:i, :j, :start), Tuple{Int64, Int64, Float64}}}:
 (i = 1, j = 2, start = 3.0)
 (i = 1, j = 1, start = 2.0)
 (i = 2, j = 2, start = 4.0)

julia> Containers.rowtable(x)
3-element Vector{NamedTuple{(:x1, :x2, :y), Tuple{Int64, Int64, VariableRef}}}:
 (x1 = 1, x2 = 2, y = x[1,2])
 (x1 = 1, x2 = 1, y = x[1,1])
 (x1 = 2, x2 = 2, y = x[2,2])
```

JuMP.Containers.AutoContainerType – Type.

```
AutoContainerType
```

Pass AutoContainerType to container to let the container type be chosen based on the type of the indices using default_container.

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.

Example

```julia
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.

Example

```
julia> iterator = Containers.vectorized_product(1:2, ["A", "B"]);

julia> collect(iterator)
2×2 Matrix{Tuple{Int64, String}}:
 (1, "A")  (1, "B")
 (2, "A")  (2, "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

```julia
julia> iterators = (() -> 1:2, (i,) -> ["A", "B"]);

julia> condition = (i, j) -> isodd(i) || j == "B";

julia> x = Containers.NestedIterator(iterators, condition);

julia> for (i, j) in x
           println((i, j))
       end
(1, "A")
(1, "B")
(2, "B")
```

is the same as

```julia
julia> for i in iterators[1]()
           for j in iterators[2](i)
               if condition(i, j)
                   println((i, j))
               end
           end
       end
(1, "A")
(1, "B")
(2, "B")
```

source

JuMP.Containers.nested – Function.

```
nested(iterators...; condition = (args...) -> true)
```

Create a NestedIterator.

Example

```julia
julia> iterator = Containers.nested(
           () -> 1:2,
           (i,) -> ["A", "B"];
           condition = (i, j) -> isodd(i) || j == "B",
       );

julia> collect(iterator)
3-element Vector{Tuple{Int64, String}}:
 (1, "A")
 (1, "B")
 (2, "B")
```

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, for example, Containers.NestedIterator

Example

See container_code for a worked example.

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`.

Example

```julia
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 26

Solutions

More information can be found in the Solutions section of the manual.

26.1 Basic utilities

JuMP.optimize! – Function.

```
optimize!(
    model::GenericModel;
    ignore_optimize_hook = (model.optimize_hook === nothing),
    _differentiation_backend::MOI.Nonlinear.AbstractAutomaticDifferentiation =
        MOI.Nonlinear.SparseReverseMode(),
    kwargs...,
)
```

Optimize the model.

If an optimizer has not been set yet (see set_optimizer), a NoOptimizer error is thrown.

If ignore_optimize_hook == true, the optimize hook is ignored and the model is solved as if the hook was not set. Keyword arguments kwargs are passed to the optimize_hook. An error is thrown if optimize_hook is nothing and keyword arguments are provided.

Experimental features

These features may change or be removed in any future version of JuMP.

Pass _differentiation_backend to set the MOI.Nonlinear.AbstractAutomaticDifferentiation backend used to compute derivatives of nonlinear programs.

If you require only :ExprGraph, it is more efficient to pass _differentiation_backend = MOI.Nonlinear.ExprGraphOnly().

source

JuMP.NoOptimizer – Type.

```
struct NoOptimizer <: Exception end
```

No optimizer is set. The optimizer can be provided to the GenericModel 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::GenericModel; result::Int = 1, verbose::Bool = false)
```

Return a struct that can be used print a summary of the solution in result `result`.

If `verbose=true`, write out the primal solution for every variable and the dual solution for every constraint, excluding those with empty names.

Example

When called at the REPL, the summary is automatically printed:

```
julia> model = Model();

julia> solution_summary(model)
* Solver : No optimizer attached.

* Status
  Result count       : 0
  Termination status : OPTIMIZE_NOT_CALLED
  Message from the solver:
  "optimize not called"

* Candidate solution (result #1)
  Primal status      : NO_SOLUTION
  Dual status        : NO_SOLUTION

* Work counters
```

Use `print` to force the printing of the summary from inside a function:

```
julia> model = Model();

julia> function foo(model)
           print(solution_summary(model))
           return
       end
foo (generic function with 1 method)

julia> foo(model)
* Solver : No optimizer attached.

* Status
```

```
  Result count      : 0
  Termination status : OPTIMIZE_NOT_CALLED
  Message from the solver:
  "optimize not called"

* Candidate solution (result #1)
  Primal status     : NO_SOLUTION
  Dual status       : NO_SOLUTION

* Work counters
```

source

26.2 Termination status

JuMP.termination_status – Function.

```
termination_status(model::GenericModel)
```

Return a MOI.TerminationStatusCode describing why the solver stopped (i.e., the MOI.TerminationStatus attribute).

source

JuMP.raw_status – Function.

```
raw_status(model::GenericModel)
```

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::GenericModel)
```

Return the number of results available to query after a call to optimize!.

source

26.3 Primal solutions

JuMP.primal_status – Function.

```
primal_status(model::GenericModel; 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::GenericModel; 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(::GenericVariableRef)`).

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::GenericVariableRef; 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::GenericVariableRef)
```

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

```
julia> model = Model();

julia> @NLparameter(model, p == 10)
p == 10.0

julia> value(p)
10.0
```

```
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.

```
value(var_value::Function, ex::NonlinearExpression)
```

Evaluate ex using var_value(v) as the value for each variable v.

```
value(c::NonlinearConstraintRef; result::Int = 1)
```

Return the value of the `NonlinearConstraintRef` c associated with result index `result` of the most-recent solution returned by the solver.

See also: result_count.

```
value(var_value::Function, c::NonlinearConstraintRef)
```

Evaluate c using var_value(v) as the value for each variable v.

26.4   Dual solutions

JuMP.dual_status – Function.

```
dual_status(model::GenericModel; 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.

JuMP.has_duals – Function.

```
has_duals(model::GenericModel; 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::NonlinearConstraintRef)
```

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::GenericVariableRef{T})::T where {T}
```

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

26.5 Basic attributes

JuMP.objective_value – Function.

```
objective_value(model::GenericModel; result::Int = 1)
```

Return the objective value associated with result index `result` of the most-recent solution returned by the solver.

For scalar-valued objectives, this function returns a Float64. For vector-valued objectives, it returns a Vector{Float64}.

See also: result_count.

source

JuMP.objective_bound – Function.

```
objective_bound(model::GenericModel)
```

Return the best known bound on the optimal objective value after a call to `optimize!(model)`.

For scalar-valued objectives, this function returns a Float64. For vector-valued objectives, it returns a Vector{Float64}.

In the case of a vector-valued objective, this returns the ideal point, that is, the point obtained if each objective was optimized independently.

source

JuMP.dual_objective_value – Function.

```
dual_objective_value(model::GenericModel; 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::GenericModel)
```

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 at-
tribute is not implemented.

source

JuMP.relative_gap – Function.

```
relative_gap(model::GenericModel)
```

Return the final relative optimality gap after a call to `optimize!(model)`. Exact value depends upon im-
plementation of MathOptInterface.RelativeGap() by the particular solver used for optimization.

source

JuMP.simplex_iterations – Function.

```
simplex_iterations(model::GenericModel)
```

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::GenericModel)
```

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::GenericModel)
```

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

26.6 Conflicts

JuMP.compute_conflict! – Function.

```
compute_conflict!(model::GenericModel)
```

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::GenericModel)
```

Return a copy of the current conflict for the model model and a GenericReferenceMap 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 GenericModel 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.

Example

In the following example, a model model is constructed with a variable x and two constraints c1 and c2. 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 iis_model.

```
julia> using JuMP

julia> import Gurobi

julia> model = Model(Gurobi.Optimizer);

julia> set_silent(model)

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

julia> @constraint(model, c1, x >= 2)
c1 : x ≥ 2.0

julia> @constraint(model, c2, x <= 1)
c2 : x ≤ 1.0
```

```
julia> optimize!(model)

julia> compute_conflict!(model)

julia> if get_attribute(model, MOI.ConflictStatus()) == MOI.CONFLICT_FOUND
           iis_model, reference_map = copy_conflict(model)
           print(iis_model)
       end
Feasibility
Subject to
 c1 : x ≥ 2.0
 c2 : x ≤ 1.0
```

source

26.7 Sensitivity

JuMP.lp_sensitivity_report – Function.

```
lp_sensitivity_report(model::GenericModel{T}; atol::T =
↪  Base.rtoldefault(T))::SensitivityReport{T} where {T}
```

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{T,T}, 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{T,T}, 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

```
julia> model = Model(HiGHS.Optimizer);

julia> set_silent(model)

julia> @variable(model, -1 <= x <= 2)
x

julia> @objective(model, Min, x)
x

julia> optimize!(model)
```

```julia
julia> report = lp_sensitivity_report(model; atol = 1e-7);

julia> dx_lo, dx_hi = report[x]
(-1.0, Inf)

julia> println(
           "The objective coefficient of `x` can decrease by $dx_lo or " *
           "increase by $dx_hi."
       )
The objective coefficient of `x` can decrease by -1.0 or increase by Inf.

julia> dRHS_lo, dRHS_hi = report[LowerBoundRef(x)]
(-Inf, 3.0)

julia> println(
           "The lower bound of `x` can decrease by $dRHS_lo or increase " *
           "by $dRHS_hi."
       )
The lower bound of `x` can decrease by -Inf or increase by 3.0.
```

source

JuMP.SensitivityReport – Type.

```
SensitivityReport
```

See lp_sensitivity_report.

source

26.8 Feasibility

JuMP.primal_feasibility_report – Function.

```julia
primal_feasibility_report(
    model::GenericModel{T},
    point::AbstractDict{GenericVariableRef{T},T} = _last_primal_solution(model),
    atol::T = zero(T),
    skip_missing::Bool = false,
)::Dict{Any,T}
```

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.

Example

```julia
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::GenericModel{T};
    atol::T = zero(T),
    skip_missing::Bool = false,
) where {T}
```

A form of `primal_feasibility_report` where a function is passed as the first argument instead of a dictionary as the second argument.

Example

```julia
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 27

Nonlinear Modeling

More information can be found in the Nonlinear Modeling section of the manual.

27.1 Models

JuMP.nonlinear_model – Function.

```
nonlinear_model(
    model::GenericModel;
    force::Bool = false,
)::Union{MOI.Nonlinear.Model,Nothing}
```

If model has nonlinear components, return a MOI.Nonlinear.Model, otherwise return nothing.

If force, always return a MOI.Nonlinear.Model, and if one does not exist for the model, create an empty one.

source

27.2 Constraints

JuMP.@NLconstraint – Macro.

```
@NLconstraint(model::GenericModel, expr)
```

Add a constraint described by the nonlinear expression expr. See also @constraint. For example:

```
julia> model = Model();

julia> @variable(model, x)
x

julia> @NLconstraint(model, sin(x) <= 1)
sin(x) - 1.0 ≤ 0

julia> @NLconstraint(model, [i = 1:3], sin(i * x) <= 1 / i)
```

```
3-element Vector{NonlinearConstraintRef{ScalarShape}}:
 (sin(1.0 * x) - 1.0 / 1.0) - 0.0 ≤ 0
 (sin(2.0 * x) - 1.0 / 2.0) - 0.0 ≤ 0
 (sin(3.0 * x) - 1.0 / 3.0) - 0.0 ≤ 0
```

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 constraints can be added on multiple lines wrapped in a begin ... end block.

The macro returns a tuple containing the constraints that were defined.

Example

```julia
julia> model = Model();

julia> @variable(model, x);

julia> @variable(model, y);

julia> @variable(model, t);

julia> @variable(model, z[1:2]);

julia> a = [4, 5];

julia> @NLconstraints(model, begin
           t >= sqrt(x^2 + y^2)
           [i = 1:2], z[i] <= log(a[i])
       end)
((t - sqrt(x ^ 2.0 + y ^ 2.0)) - 0.0 ≥ 0, NonlinearConstraintRef{ScalarShape}[(z[1] - log(4.0))
↪  - 0.0 ≤ 0, (z[2] - log(5.0)) - 0.0 ≤ 0])
```

source

JuMP.num_nonlinear_constraints – Function.

```
num_nonlinear_constraints(model::GenericModel)
```

Returns the number of nonlinear constraints associated with the model.

source

JuMP.add_nonlinear_constraint – Function.

```
add_nonlinear_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.

Example

```
julia> model = Model();

julia> @variable(model, x);

julia> add_nonlinear_constraint(model, :($(x) + $(x)^2 <= 1))
(x + x ^ 2.0) - 1.0 ≤ 0
```

source

JuMP.all_nonlinear_constraints – Function.

```
all_nonlinear_constraints(model::GenericModel)
```

Return a vector of all nonlinear constraint references in the model in the order they were added to the model.

source

JuMP.nonlinear_dual_start_value – Function.

```
nonlinear_dual_start_value(model::Model)
```

Return the current value of the MOI attribute MOI.NLPBlockDualStart.

source

JuMP.set_nonlinear_dual_start_value – Function.

```
set_nonlinear_dual_start_value(
    model::Model,
    start::Union{Nothing,Vector{Float64}},
)
```

Set the value of the MOI attribute MOI.NLPBlockDualStart.

The start vector corresponds to the Lagrangian duals of the nonlinear constraints, in the order given by all_nonlinear_constraints. That is, you must pass a single start vector corresponding to all of the nonlinear constraints in a single function call; you cannot set the dual start value of nonlinear constraints one-by-one. The example below demonstrates how to use all_nonlinear_constraints to create a mapping between the nonlinear constraint references and the start vector.

Pass nothing to unset a previous start.

Example

```julia
julia> model = Model();

julia> @variable(model, x[1:2]);

julia> nl1 = @NLconstraint(model, x[1] <= sqrt(x[2]));

julia> nl2 = @NLconstraint(model, x[1] >= exp(x[2]));

julia> start = Dict(nl1 => -1.0, nl2 => 1.0);

julia> start_vector = [start[con] for con in all_nonlinear_constraints(model)]
2-element Vector{Float64}:
 -1.0
  1.0

julia> set_nonlinear_dual_start_value(model, start_vector)

julia> nonlinear_dual_start_value(model)
2-element Vector{Float64}:
 -1.0
  1.0
```

source

27.3 Expressions

JuMP.@NLexpression – Macro.

```julia
@NLexpression(args...)
```

Efficiently build a nonlinear expression which can then be inserted in other nonlinear constraints and the objective. See also [@expression]. For example:

```julia
julia> model = Model();

julia> @variable(model, x)
x

julia> @variable(model, y)
y

julia> @NLexpression(model, my_expr, sin(x)^2 + cos(x^2))
```

```
subexpression[1]: sin(x) ^ 2.0 + cos(x ^ 2.0)

julia> @NLconstraint(model, my_expr + y >= 5)
(subexpression[1] + y) - 5.0 ≥ 0

julia> @NLobjective(model, Min, my_expr)
```

Indexing over sets and anonymous expressions are also supported:

```
julia> @NLexpression(model, my_expr_1[i=1:3], sin(i * x))
3-element Vector{NonlinearExpression}:
 subexpression[2]: sin(1.0 * x)
 subexpression[3]: sin(2.0 * x)
 subexpression[4]: sin(3.0 * x)

julia> my_expr_2 = @NLexpression(model, log(1 + sum(exp(my_expr_1[i]) for i in 1:2)))
subexpression[5]: log(1.0 + (exp(subexpression[2]) + exp(subexpression[3])))
```

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 expressions can be added on multiple lines wrapped in a begin ... end block.

The macro returns a tuple containing the expressions that were defined.

Example

```
julia> model = Model();

julia> @variable(model, x);

julia> @variable(model, y);

julia> @variable(model, z[1:2]);

julia> a = [4, 5];

julia> @NLexpressions(model, begin
           my_expr, sqrt(x^2 + y^2)
           my_expr_1[i = 1:2], log(a[i]) - z[i]
       end)
(subexpression[1]: sqrt(x ^ 2.0 + y ^ 2.0), NonlinearExpression[subexpression[2]: log(4.0) -
↪  z[1], subexpression[3]: log(5.0) - z[2]])
```

source

JuMP.NonlinearExpression – Type.

```
NonlinearExpression <: AbstractJuMPScalar
```

A struct to represent a nonlinear expression.

Create an expression using @NLexpression.

source

JuMP.add_nonlinear_expression – Function.

```
add_nonlinear_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.

Example

```
julia> model = Model();

julia> @variable(model, x);

julia> add_nonlinear_expression(model, :($(x) + $(x)^2))
subexpression[1]: x + x ^ 2.0
```

source

27.4 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

```
julia> model = Model();

julia> @variable(model, x)
x
```

```
julia> @NLobjective(model, Max, 2x + 1 + sin(x))

julia> print(model)
Max 2.0 * x + 1.0 + sin(x)
Subject to
```

JuMP.set_nonlinear_objective – Function.

```
set_nonlinear_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 MIN_SENSE or MAX_SENSE instead of Min and Max.

Example

```
julia> model = Model();

julia> @variable(model, x);

julia> set_nonlinear_objective(model, MIN_SENSE, :($(x) + $(x)^2))
```

27.5 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

```
julia> model = Model();

julia> @NLparameter(model, x == 10)
x == 10.0

julia> value(x)
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

```
julia> model = Model();

julia> x = @NLparameter(model, value = 10)
parameter[1] == 10.0

julia> value(x)
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

```
julia> model = Model();

julia> @NLparameter(model, y[i = 1:3] == 2 * i)
3-element Vector{NonlinearParameter}:
 parameter[1] == 2.0
 parameter[2] == 4.0
 parameter[3] == 6.0

julia> value(y[2])
4.0
```

```
@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

```
julia> model = Model();

julia> y = @NLparameter(model, [i = 1:3] == 2 * i)
3-element Vector{NonlinearParameter}:
 parameter[1] == 2.0
 parameter[2] == 4.0
 parameter[3] == 6.0

julia> value(y[2])
4.0
```

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.

The macro returns a tuple containing the parameters that were defined.

Example

```
julia> model = Model();

julia> @NLparameters(model, begin
           x == 10
           b == 156
       end);

julia> value(x)
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

```
julia> model = Model();

julia> @NLparameter(model, p == 10)
p == 10.0

julia> value(p)
10.0
```

source

JuMP.set_value – Method.

```
set_value(p::NonlinearParameter, v::Number)
```

Store the value v in the nonlinear parameter p.

Example

```
julia> model = Model();

julia> @NLparameter(model, p == 0)
p == 0.0

julia> set_value(p, 5)
5

julia> value(p)
5.0
```

source

27.6 User-defined functions

JuMP.register – Function.

```
register(
    model::Model,
    op::Symbol,
    dimension::Integer,
    f::Function;
    autodiff:Bool = false,
)
```

Register the user-defined function f that takes `dimension` arguments in `model` as the symbol op.

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`.
- op does not have to be the same symbol as f, but it is generally more readable if it is.

Example

```julia
julia> model = Model();

julia> @variable(model, x)
x

julia> f(x::T) where {T<:Real} = x^2
f (generic function with 1 method)

julia> register(model, :foo, 1, f; autodiff = true)

julia> @NLobjective(model, Min, foo(x))
```

```julia
julia> model = Model();

julia> @variable(model, x[1:2])
2-element Vector{VariableRef}:
 x[1]
 x[2]

julia> g(x::T, y::T) where {T<:Real} = x * y
g (generic function with 1 method)

julia> register(model, :g, 2, g; autodiff = true)

julia> @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.

Example

```julia
julia> model = Model();

julia> @variable(model, x)
x

julia> f(x::T) where {T<:Real} = x^2
f (generic function with 1 method)

julia> ∇f(x::T) where {T<:Real} = 2 * x
∇f (generic function with 1 method)

julia> register(model, :foo, 1, f, ∇f; autodiff = true)

julia> @NLobjective(model, Min, foo(x))
```

```julia
julia> model = Model();

julia> @variable(model, x[1:2])
2-element Vector{VariableRef}:
 x[1]
 x[2]

julia> g(x::T, y::T) where {T<:Real} = x * y
g (generic function with 1 method)

julia> function ∇g(g::AbstractVector{T}, x::T, y::T) where {T<:Real}
           g[1] = y
           g[2] = x
           return
       end
∇g (generic function with 1 method)

julia> register(model, :g, 2, g, ∇g)

julia> @NLobjective(model, Min, g(x[1], x[2]))
```

```
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.

Example

```
julia> model = Model();

julia> @variable(model, x)
x

julia> f(x::Float64) = x^2
f (generic function with 1 method)

julia> ∇f(x::Float64) = 2 * x
∇f (generic function with 1 method)

julia> ∇²f(x::Float64) = 2.0
∇²f (generic function with 1 method)

julia> register(model, :foo, 1, f, ∇f, ∇²f)

julia> @NLobjective(model, Min, foo(x))
```

27.7 Derivatives

JuMP.NLPEvaluator – Function.

```
NLPEvaluator(
    model::Model,
```

```
    _differentiation_backend::MOI.Nonlinear.AbstractAutomaticDifferentiation =
        MOI.Nonlinear.SparseReverseMode(),
)
```

Return an `MOI.AbstractNLPEvaluator` constructed from model

> **Warning**
>
> Before using, you must initialize the evaluator using `MOI.initialize`.

Experimental

These features may change or be removed in any future version of JuMP.

Pass _differentiation_backend to specify the differentiation backend used to compute derivatives.

source

Chapter 28

Callbacks

More information can be found in the Callbacks section of the manual.

28.1 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.

Example

```julia
julia> model = Model();

julia> @variable(model, x);

julia> @build_constraint(2x >= 1)
ScalarConstraint{AffExpr, MathOptInterface.GreaterThan{Float64}}(2 x,
↪   MathOptInterface.GreaterThan{Float64}(1.0))
```

source

28.2 Callback variable primal

JuMP.callback_value – Function.

```
callback_value(cb_data, x::GenericVariableRef)
```

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.

```
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.

28.3   Callback node status

JuMP.callback_node_status – Function.

```
callback_node_status(cb_data, model::GenericModel)
```

Return an MOI.CallbackNodeStatusCode enum, indicating if the current primal solution available from callback_value is integer feasible.

Chapter 29

Extensions

More information can be found in the [Extensions](#) section of the manual.

29.1 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](#)

JuMP.`AbstractScalarSet` – Type.

```
AbstractScalarSet
```

An abstract type for defining new scalar sets in JuMP.

Implement `moi_set(::AbstractScalarSet)` to convert the type into an MOI set.

See also: [moi_set](#).

[source](#)

29.2 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::GenericModel, v::AbstractVariable, name::String="")
```

Add a variable v to Model m and sets its name.

source

JuMP.build_variable – Function.

```
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.

Example

```
@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

```
build_variable(_error::Function, variables, ::SymmetricMatrixSpace)
```

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:

```
julia> model = Model();

julia> @variable(model, Q[1:2, 1:2], Symmetric)
2×2 LinearAlgebra.Symmetric{VariableRef, Matrix{VariableRef}}:
 Q[1,1]  Q[1,2]
 Q[1,2]  Q[2,2]
```

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:

```
julia> model = Model();

julia> @variable(model, Q[1:2, 1:2] in SkewSymmetricMatrixSpace())
2×2 Matrix{AffExpr}:
 0       Q[1,2]
 -Q[1,2]  0
```

source

```
build_variable(_error::Function, variables, ::HermitianMatrixSpace)
```

Return a VariablesConstrainedOnCreation of shape HermitianMatrixShape 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:

```
julia> model = Model();

julia> @variable(model, Q[1:2, 1:2] in HermitianMatrixSpace())
2×2 LinearAlgebra.Hermitian{GenericAffExpr{ComplexF64, VariableRef},
↪  Matrix{GenericAffExpr{ComplexF64, VariableRef}}}:
 real(Q[1,1])                         real(Q[1,2]) + (0.0 + 1.0im) imag(Q[1,2])
 real(Q[1,2]) + (0.0 - 1.0im) imag(Q[1,2])  real(Q[2,2])
```

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:

```
julia> model = Model();

julia> @variable(model, Q[1:2, 1:2], PSD)
2×2 LinearAlgebra.Symmetric{VariableRef, Matrix{VariableRef}}:
 Q[1,1]  Q[1,2]
 Q[1,2]  Q[2,2]
```

source

29.3 Extend @constraint

JuMP.build_constraint – Function.

```
build_constraint(
    _error::Function,
    Q::LinearAlgebra.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:

```
julia> import LinearAlgebra

julia> model = Model();

julia> @variable(model, Q[1:2, 1:2]);

julia> @constraint(model, LinearAlgebra.Symmetric(Q) in PSDCone())
[Q[1,1]  Q[1,2];
 Q[1,2]  Q[2,2]] ∈ 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.,

```
julia> model = Model();

julia> @variable(model, Q[1:2, 1:2], Symmetric)
2×2 LinearAlgebra.Symmetric{VariableRef, Matrix{VariableRef}}:
 Q[1,1]  Q[1,2]
 Q[1,2]  Q[2,2]

julia> @constraint(model, Q in PSDCone())
[Q[1,1]  Q[1,2];
 Q[1,2]  Q[2,2]] ∈ PSDCone()
```

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:

```
julia> model = Model();

julia> @variable(model, Q[1:2, 1:2]);
```

```julia
julia> @constraint(model, Q in PSDCone())
[Q[1,1]  Q[1,2];
 Q[2,1]  Q[2,2]] ∈ PSDCone()
```

source

```
build_constraint(
    _error::Function,
    Q::LinearAlgebra.Hermitian{V,M},
    ::HermitianPSDCone,
) where {V<:AbstractJuMPScalar,M<:AbstractMatrix{V}}
```

Return a VectorConstraint of shape HermitianMatrixShape constraining the matrix Q to be Hermitian positive semidefinite.

This function is used by the @constraint macros as follows:

```julia
julia> import LinearAlgebra

julia> model = Model();

julia> @variable(model, Q[1:2, 1:2]);

julia> @constraint(model, LinearAlgebra.Hermitian(Q) in HermitianPSDCone())
[Q[1,1]  Q[1,2];
 Q[1,2]  Q[2,2]] ∈ HermitianPSDCone()
```

source

```
build_constraint(
    _error::Function,
    f::AbstractVector{<:AbstractJuMPScalar},
    ::Nonnegatives,
    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},
    ::Nonpositives,
    extra::Union{MOI.AbstractVectorSet,AbstractVectorSet},
)
```

A helper method that re-writes

```
@constraint(model, Y <= X, extra)
```

into

```
@constraint(model, X - Y in extra)
```

source

JuMP.add_constraint – Function.

```
add_constraint(model::GenericModel, con::AbstractConstraint, name::String="")
```

Add a constraint con to Model model and sets its name.

source

JuMP.model_convert – Function.

```
model_convert(
    model::AbstractModel,
    rhs::Union{
        AbstractConstraint,
        Number,
        AbstractJuMPScalar,
        MOI.AbstractSet,
    },
)
```

Convert the coefficients and constants of functions and sets in the rhs to the coefficient type value_type(typeof(model)).

Purpose

Creating and adding a constraint is a two-step process. The first step calls build_constraint, and the result of that is passed to add_constraint.

However, because build_constraint does not take the model as an argument, the coefficients and constants of the function or set might be different than value_type(typeof(model)).

Therefore, the result of build_constraint is converted in a call to model_convert before the result is passed to add_constraint.

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.

Example

Given a SymmetricMatrixShape of vectorized form [1, 2, 3], the following code returns the matrix Symmetric(Matrix[1 2; 2 3]):

```julia
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.

Example

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
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.

Example

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
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 30

Algebraic modeling languages

JuMP is an algebraic modeling language for mathematical optimization written in the Julia language. In this page, we explain what an algebraic modeling language actually is.

30.1   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, HiGHS 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.

Despite the textbook view of a linear program, you probably formulated problems algebraically like so:

$$\max \sum_{i=1}^{n} c_i x_i$$

$$\text{s.t.} \sum_{i=1}^{n} w_i x_i \leq b$$

$$x_i \geq 0 \quad \forall i = 1, \ldots, n$$

$$x_i \in \mathbb{Z} \quad \forall i = 1, \ldots, n.$$

> **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 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:

$$\max \sum_{i=1}^{n} c_i x_i$$

$$\text{s.t.} x_0 + \sum_{i=1}^{n} w_i x_i = b$$

$$x_i \geq 0 \quad \forall i = 0, \ldots, n$$

$$x_i \in \mathbb{Z} \quad \forall i = 1, \ldots, n.$$

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).

Part 2 is less trivial than it might seem, because each solver has a unique application programming interface (API) and data structure 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.

30.2 From user to solver

This section provides a brief summary of the steps that happen in order to translate the model that the user writes into a model that the solver understands.

Step 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 write the knapsack problem in JuMP:

```julia
julia> using JuMP, HiGHS

julia> function algebraic_knapsack(c, w, b)
           n = length(c)
           model = Model(HiGHS.Optimizer)
           set_silent(model)
           @variable(model, x[1:n] >= 0, Int)
           @objective(model, Max, sum(c[i] * x[i] for i = 1:n))
           @constraint(model, sum(w[i] * x[i] for i = 1:n) <= b)
           optimize!(model)
           return value.(x)
       end
algebraic_knapsack (generic function with 1 method)

julia> algebraic_knapsack([1, 2], [0.5, 0.5], 1.25)
2-element Vector{Float64}:
 0.0
 2.0
```

This formulation is compact, and it closely matches the algebraic formulation of the model we wrote out above.

Step II: algebraic to functional

For the next step, JuMP's macros re-write the variables and constraints into a functional form. Here's what the JuMP code looks like after this step:

```julia
julia> using JuMP, HiGHS

julia> function nonalgebraic_knapsack(c, w, b)
           n = length(c)
           model = Model(HiGHS.Optimizer)
           set_silent(model)
```

```julia
            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, MAX_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)
            optimize!(model)
            return value.(x)
        end
nonalgebraic_knapsack (generic function with 1 method)

julia> nonalgebraic_knapsack([1, 2], [0.5, 0.5], 1.25)
2-element Vector{Float64}:
 0.0
 2.0
```

Hopefully you agree that the macro version is much easier to read.

Part III: JuMP to MathOptInterface

In the third step, JuMP converts the functional form of the problem, that is, `nonalgebraic_knapsack`, into the MathOptInterface API:

```julia
julia> import MathOptInterface as MOI

julia> import HiGHS

julia> function mathoptinterface_knapsack(optimizer, c, w, b)
            n = length(c)
            model = MOI.instantiate(optimizer)
            MOI.set(model, MOI.Silent(), true)
            x = MOI.add_variables(model, n)
            for i in 1:n
                MOI.add_constraint(model, x[i], MOI.GreaterThan(0.0))
                MOI.add_constraint(model, x[i], MOI.Integer())
                MOI.set(model, MOI.VariableName(), x[i], "x[$i]")
            end
            MOI.set(model, MOI.ObjectiveSense(), MOI.MAX_SENSE)
            obj = MOI.ScalarAffineFunction(MOI.ScalarAffineTerm.(c, x), 0.0)
            MOI.set(model, MOI.ObjectiveFunction{typeof(obj)}(), obj)
            MOI.add_constraint(
                model,
                MOI.ScalarAffineFunction(MOI.ScalarAffineTerm.(w, x), 0.0),
```

```
                MOI.LessThan(b),
            )
            MOI.optimize!(model)
            return MOI.get.(model, MOI.VariablePrimal(), x)
        end
mathoptinterface_knapsack (generic function with 1 method)

julia> mathoptinterface_knapsack(HiGHS.Optimizer, [1.0, 2.0], [0.5, 0.5], 1.25)
2-element Vector{Float64}:
 0.0
 2.0
```

The code is becoming more verbose and looking less like the mathematical formulation that we started with.

Step IV: MathOptInterface to HiGHS

As a final step, the HiGHS.jl package converts the MathOptInterface form, that is, `mathoptinterface_knapsack`, into a HiGHS-specific API:

```
julia> using HiGHS

julia> function highs_knapsack(c, w, b)
            n = length(c)
            model = Highs_create()
            Highs_setBoolOptionValue(model, "output_flag", false)
            for i in 1:n
                Highs_addCol(model, c[i], 0.0, Inf, 0, C_NULL, C_NULL)
                Highs_changeColIntegrality(model, i-1, 1)
            end
            Highs_changeObjectiveSense(model, -1)
            Highs_addRow(
                model,
                -Inf,
                b,
                Cint(length(w)),
                collect(Cint(0):Cint(n-1)),
                w,
            )
            Highs_run(model)
            x = fill(NaN, 2)
            Highs_getSolution(model, x, C_NULL, C_NULL, C_NULL)
            Highs_destroy(model)
            return x
        end
highs_knapsack (generic function with 1 method)

julia> highs_knapsack([1.0, 2.0], [0.5, 0.5], 1.25)
2-element Vector{Float64}:
 0.0
 2.0
```

We've now gone from a algebraic model that looked identical to the mathematical model we started with, to a verbose function that uses HiGHS-specific functionality.

The difference between `algebraic_knapsack` and `highs_knapsack` highlights the benefit that algebraic modeling languages provide to users. Moreover, if we used a different solver, the solver-specific function would be entirely different. A key benefit of an algebraic modeling language is that you can change the solver without needing to rewrite the model.

Part VI

Developer Docs

Chapter 31

Contributing

31.1 How to contribute to JuMP

Welcome, this document explains some 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 similar to 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 and run JuliaFormatter
- 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 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, 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.

Chapter 32

Extensions

32.1 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

Compatibility

When writing JuMP extensions, you should carefully consider the compatibility guarantees that JuMP makes. In particular:

- All functions, structs, and constants which do not begin with an underscore (_) are public. These are always safe to use, and they should all have corresponding documentation.

- All identifiers beginning with an underscore (_) are private. These are not safe to use, because they may break in any JuMP release, including patch releases.

- Unless explicitly mentioned in the documentation, all fields of a struct are private. These are not safe to use, because they may break in any JuMP release, including patch releases. An example of a field which is safe to use is the `model.ext` extension dictionary, which is documented in The extension dictionary.

In general, we strongly encourage you to use only the public API of JuMP. If you are missing a feature, please open a GitHub issue.

However, if you do use the private API (for example, because your feature request has not been implemented yet), then you must carefully restrict the versions of JuMP that your package is compatible with in the `Project.toml` file. The easiest way to do this is via the hyphen specifiers. For example, if your package supports all JuMP versions between v1.0.0 and v1.1.1, do:

```
JuMP = "1.0.0 - 1.1.1"
```

Then, whenever JuMP releases a new version, you should check if your package is still compatible and update the bound accordingly.

Define a new set

To define a new set for JuMP, subtype `MOI.AbstractScalarSet` or `MOI.AbstractVectorSet` and implement `Base.copy` for the set.

```
julia> struct NewMOIVectorSet <: MOI.AbstractVectorSet
           dimension::Int
       end

julia> Base.copy(x::NewMOIVectorSet) = x

julia> model = Model();

julia> @variable(model, x[1:2]);

julia> @constraint(model, x in NewMOIVectorSet(2))
[x[1], x[2]] ∈ NewMOIVectorSet(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.

```
julia> struct NewVectorSet <: JuMP.AbstractVectorSet end

julia> JuMP.moi_set(::NewVectorSet, dim::Int) = NewMOIVectorSet(dim)

julia> @constraint(model, x in NewVectorSet())
[x[1], x[2]] ∈ NewMOIVectorSet(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
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
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
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 similar to Bin or Int, or through the variable_type keyword. However, now it adds two variables instead of one.

```julia
julia> model = Model();

julia> @variable(model, x[i=1:2], variable_type = 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
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
```

`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
```

> **Tip**
>
> When parsing a constraint you can recurse into sub-constraint (for example, 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, for example, 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
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
```

> **Note**
>
> Only a single positional argument can be given to a particular constraint. Extensions that seek to pass multiple arguments (for example, Foo and Bar) should combine them into one argument type (for example, 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
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,
```

```julia
            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
```

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
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
```

If you define extension data, implement copy_extension_data to support copy_model.

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 AbstractVariableRefs 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.

Testing JuMP extensions

The JuMP test suite contains a large number of tests for JuMP extensions. You can run these tests by copying the MIT-licensed Kokako.jl file in the JuMP tests into your /test folder, and then adding this snippet to your /test/runtests.jl file:

```julia
using MyJuMPExtension
import JuMP
include("Kokako.jl")
const MODULES_TO_TEST = Kokako.include_modules_to_test(JuMP)
Kokako.run_tests(
    MODULES_TO_TEST,
    MyJuMPExtension.MyModel,
    MyJuMPExtension.MyVariableRef;
    test_prefix = "test_extension_",
)
```

Set an optimize! hook

Some extensions require modification to the problem after the user has finished constructing the problem, but before optimize! is called. For these situations, JuMP provides set_optimize_hook, which lets you intercept the optimize! call.

Here's a simple example of adding an optimize hook that extends optimize! to take a keyword argument silent:

```julia
julia> using JuMP, HiGHS

julia> model = Model(HiGHS.Optimizer);

julia> @variable(model, x >= 1.5, Int);

julia> @objective(model, Min, x);
```

```
julia> function silent_hook(model; silent::Bool)
           if silent
               set_silent(model)
           else
               unset_silent(model)
           end
           ## Make sure you set ignore_optimize_hook = true, or we'll
           ## recursively enter the optimize hook!
           return optimize!(model; ignore_optimize_hook = true)
       end
silent_hook (generic function with 1 method)

julia> set_optimize_hook(model, silent_hook)
silent_hook (generic function with 1 method)

julia> optimize!(model; silent = true)

julia> optimize!(model; silent = false)
Solution has                  num          max          sum
Col     infeasibilities       0            0            0
Integer infeasibilities       0            0            0
Row     infeasibilities       0            0            0
Row     residuals             0            0            0
Presolving model
0 rows, 0 cols, 0 nonzeros
0 rows, 0 cols, 0 nonzeros
Presolve: Optimal

Solving report
  Status            Optimal
  Primal bound      2
  Dual bound        2
  Gap               0% (tolerance: 0.01%)
  Solution status   feasible
                    2 (objective)
                    0 (bound viol.)
                    0 (int. viol.)
                    0 (row viol.)
  Timing            0.00 (total)
                    0.00 (presolve)
                    0.00 (postsolve)
  Nodes             0
  LP iterations     0 (total)
                    0 (strong br.)
                    0 (separation)
                    0 (heuristics)
```

Creating new container types

JuMP macros (for example, @variable) accept a container keyword argument to force the type of container that is chosen. By default, JuMP supports container = Array, container = DenseAxisArray, container = SparseAxisArray and container = Auto. You can extend support to user-defined types by implementing Containers.container.

For example, here is a container that reverses the order of the indices:

```julia
julia> struct Foo end

julia> function Containers.container(f::Function, indices, ::Type{Foo})
           return reverse([f(i...) for i in indices])
       end

julia> model = Model();

julia> @variable(model, x[1:3], container = Foo)
3-element Vector{VariableRef}:
 x[3]
 x[2]
 x[1]

julia> x[1]
x[3]

julia> @variable(model, y[1:3, 1:2], container = Foo)
3×2 Matrix{VariableRef}:
 y[3,2]  y[3,1]
 y[2,2]  y[2,1]
 y[1,2]  y[1,1]

julia> y[1, 1]
y[3,2]

julia> @variable(model, z[i=1:3; isodd(i)], container = Foo)
2-element Vector{VariableRef}:
 z[3]
 z[1]

julia> z[2]
z[1]
```

> **Warning**
>
> If you are a general user, you should not need to create a new container type. Instead, consider following User-defined containers and create a new container using standard Julia syntax. For example:
>
> ```julia
> julia> model = Model();
>
> julia> @variable(model, x[1:3])
> 3-element Vector{VariableRef}:
>  x[1]
>  x[2]
>  x[3]
>
> julia> y = reverse(x)
> 3-element Vector{VariableRef}:
>  x[3]
>  x[2]
>  x[1]
> ```

Performance tips for extensions

The function-in-set design of MathOptInterface causes type stability issues in Julia if you try to iterate over all of the constraints in a model. The easiest way to fix this is to use a function barrier.

For example, instead of:

```julia
function all_names_slow(model)
    names = Set{String}()
    for ci in all_constraints(model)
        push!(names, name(ci))
    end
    return names
end
```

use:

```julia
function _function_barrier(names, model, ::Type{F}, ::Type{S}) where {F,S}
    for ci in all_constraints(model, F, S)
        push!(names, name(ci))
    end
    return
end

function all_names_fast(model)
    names = Set{String}()
    for (F, S) in list_of_constraint_types(model)
        _function_barrier(names, model, F, S)
    end
    return names
end
```

> **Note**
>
> It is important to explicitly type the F and S arguments. If you leave them untyped, for example, `function _function_barrier(names, model, F, S)`, Julia will not specialize the function calls and performance will not be improved.

Chapter 33

Custom binaries

33.1   How to use a custom binary

Many solvers are not written in Julia, but instead in languages like C or C++. JuMP interacts with these solvers through binary dependencies.

For many open-source solvers, we automatically install the appropriate binary when you run `Pkg.add("Solver")`. For example, `Pkg.add("ECOS")` will also install the ECOS binary.

This page explains how this installation works, and how you can use a custom binary.

> **Compat**
>
> These instructions require Julia 1.6 or later.

Background

Each solver that JuMP supports is structured as a Julia package. For example, the interface for the ECOS solver is provided by the ECOS.jl package.

> **Tip**
>
> This page uses the example of ECOS.jl because it is simple to compile. Other solvers follow similar conventions. For example, the interface to the Clp solver is provided by Clp.jl.

The ECOS.jl package provides an interface between the C API of ECOS and MathOptInterface. However, it does not handle the installation of the solver binary; that is the job for a JLL package.

A JLL is a Julia package that wraps a pre-compiled binary. Binaries are built using Yggdrasil (for example, ECOS) and hosted in the JuliaBinaryWrappers GitHub repository (for example, ECOS_jll.jl).

JLL packages contain little code. Their only job is to `dlopen` a dynamic library, along with any dependencies.

JLL packages manage their binary dependencies using Julia's artifact system. Each JLL package has an `Artifacts.toml` file which describes where to find each binary artifact for each different platform that it might be installed on. Here is the Artifacts.toml file for ECOS_jll.jl.

The binaries installed by the JLL package should be sufficient for most users. In rare cases, however, you may require a custom binary. The two main reasons to use a custom binary are:

- You want a binary with custom compilation settings (for example, debugging)

- You want a binary with a set of dependencies that are not available on Yggdrasil (for example, a commercial solver like Gurobi or CPLEX).

The following sections explain how to replace the binaries provided by a JLL package with the custom ones you have compiled. As a reminder, we use ECOS as an example for simplicity, but the steps are the same for other solvers.

Explore the JLL you want to override

The first step is to explore the structure and filenames of the JLL package we want to override.

Find the location of the files using `.artifact_dir`:

```julia
julia> using ECOS_jll

julia> ECOS_jll.artifact_dir
"/Users/oscar/.julia/artifacts/2addb75332eff5a1657b46bb6bf30d2410bc7ecf"
```

> **Tip**
>
> This path may be different on other machines.

Here is what it contains:

```julia
julia> readdir(ECOS_jll.artifact_dir)
4-element Vector{String}:
 "include"
 "lib"
 "logs"
 "share"

julia> readdir(joinpath(ECOS_jll.artifact_dir, "lib"))
1-element Vector{String}:
 "libecos.dylib"
```

Other solvers may have a `bin` directory containing executables. To use a custom binary of ECOS, we need to replace `/lib/libecos.dylib` with our custom binary.

Compile a custom binary

The next step is to compile a custom binary. Because ECOS is written in C with no dependencies, this is easy to do if you have a C compiler:

```
oscar@Oscars-MBP jll_example % git clone https://github.com/embotech/ecos.git
[... lines omitted ...]
oscar@Oscars-MBP jll_example % cd ecos
oscar@Oscars-MBP ecos % make shared
[... many lines omitted...]
oscar@Oscars-MBP ecos % mkdir lib
oscar@Oscars-MBP ecos % cp libecos.dylib lib
```

> **Warning**
>
> Compiling custom solver binaries is an advanced operation. Due to the complexities of compiling various solvers, the JuMP community is unable to help you diagnose and fix compilation issues.

After this compilation step, we now have a folder /tmp/jll_example/ecos that contains lib and include directories with the same files as ECOS_jll:

```julia
julia> readdir(joinpath("ecos", "lib"))
1-element Vector{String}:
 "libecos.dylib"
```

Overriding a single library

To override the libecos library, we need to know what ECOS_jll calls it. (In most cases, it will also be libecos, but not always.)

There are two ways you can check.

1. Check the bottom of the JLL's GitHub README. For example, ECOS_jll has a single LibraryProduct called libecos.

2. Type ECOS_jll. and the press the [TAB] key twice to auto-complete available options:

   ```julia
   julia> ECOS_jll.
   LIBPATH          PATH_list        best_wrapper     get_libecos_path  libecos_handle
   LIBPATH_list     __init__         dev_jll          is_available      libecos_path
   PATH             artifact_dir     find_artifact_dir libecos
   ```

   Here you can see there is libecos, and more usefully for us, libecos_path.

Once you know the name of the variable to override (the one that ends in _path), use Preferences.jl to specify a new path:

```julia
using Preferences
set_preferences!(
    "LocalPreferences.toml",
    "ECOS_jll",
    "libecos_path" => "/tmp/jll_example/ecos/lib/libecos"
)
```

This will create a file in your current directory called LocalPreferences.toml with the contents:

```toml
[ECOS_jll]
libecos_path = "/tmp/jll_example/ecos/lib/libecos"
```

Now if you restart Julia, you will see:

```
julia> using ECOS_jll

julia> ECOS_jll.libecos
"/tmp/jll_example/ecos/lib/libecos"
```

To go back to using the default library, just delete the LocalPreferences.toml file.

Overriding an entire artifact

Sometimes a solver may provide a number of libraries and executables, and specifying the path for each of the becomes tedious. In this case, we can use Julia's Override.toml to replace an entire artifact.

Overriding an entire artifact requires you to replicate the structure and contents of the JLL package that we explored above.

In most cases you need only reproduce the include, lib, and bin directories (if they exist). You can safely ignore any logs or share directories. Take careful note of what files each directory contains and what they are called.

For our ECOS example, we already reproduced the structure when we compiled ECOS.

So, now we need to tell Julia to use our custom installation instead of the default. We can do this by making an override file at ~/.julia/artifacts/Overrides.toml.

Overrides.toml has the following content:

```
# Override for ECOS_jll
2addb75332eff5a1657b46bb6bf30d2410bc7ecf = "/tmp/jll_example/ecos"
```

where 2addb75332eff5a1657b46bb6bf30d2410bc7ecf is the folder from the original ECOS_jll.artifact_dir and "/tmp/jll_example/ecos" is the location of our new installation. Replace these as appropriate for your system.

If you restart Julia after creating the override file, you will see:

```
julia> using ECOS_jll

julia> ECOS_jll.artifact_dir
"/tmp/jll_example/ecos"
```

Now when we use ECOS it will use our custom binary.

Using Cbc with a custom binary

As a second example, we demonstrate how to use Cbc.jl with a custom binary.

Explore the JLL you want to override

First, let's check where Cbc_jll is installed:

```
julia> using Cbc_jll

julia> Cbc_jll.artifact_dir
```

```
"/Users/oscar/.julia/artifacts/e481bc81db5e229ba1f52b2b4bd57484204b1b06"

julia> readdir(Cbc_jll.artifact_dir)
5-element Vector{String}:
 "bin"
 "include"
 "lib"
 "logs"
 "share"

julia> readdir(joinpath(Cbc_jll.artifact_dir, "bin"))
1-element Vector{String}:
 "cbc"

julia> readdir(joinpath(Cbc_jll.artifact_dir, "lib"))
10-element Vector{String}:
 "libCbc.3.10.5.dylib"
 "libCbc.3.dylib"
 "libCbc.dylib"
 "libCbcSolver.3.10.5.dylib"
 "libCbcSolver.3.dylib"
 "libCbcSolver.dylib"
 "libOsiCbc.3.10.5.dylib"
 "libOsiCbc.3.dylib"
 "libOsiCbc.dylib"
 "pkgconfig"
```

Compile a custom binary

Next, we need to compile Cbc. Cbc can be difficult to compile (it has a lot of dependencies), but for macOS users there is a homebrew recipe:

```
(base) oscar@Oscars-MBP jll_example % brew install cbc
[ ... lines omitted ... ]
(base) oscar@Oscars-MBP jll_example % brew list cbc
/usr/local/Cellar/cbc/2.10.5/bin/cbc
/usr/local/Cellar/cbc/2.10.5/include/cbc/ (76 files)
/usr/local/Cellar/cbc/2.10.5/lib/libCbc.3.10.5.dylib
/usr/local/Cellar/cbc/2.10.5/lib/libCbcSolver.3.10.5.dylib
/usr/local/Cellar/cbc/2.10.5/lib/libOsiCbc.3.10.5.dylib
/usr/local/Cellar/cbc/2.10.5/lib/pkgconfig/ (2 files)
/usr/local/Cellar/cbc/2.10.5/lib/ (6 other files)
/usr/local/Cellar/cbc/2.10.5/share/cbc/ (59 files)
/usr/local/Cellar/cbc/2.10.5/share/coin/ (4 files)
```

Override single libraries

To use Preferences.jl to override specific libraries we first check the names of each library in Cbc_jll:

```
julia> Cbc_jll.
LIBPATH                 cbc                     get_libcbcsolver_path   libOsiCbc_path
LIBPATH_list            cbc_path                is_available            libcbcsolver
PATH                    dev_jll                 libCbc                  libcbcsolver_handle
```

```
PATH_list              find_artifact_dir       libCbc_handle            libcbcsolver_path
__init__               get_cbc_path            libCbc_path
artifact_dir           get_libCbc_path         libOsiCbc
best_wrapper           get_libOsiCbc_path      libOsiCbc_handle
```

Then we add the following to LocalPreferences.toml:

```
[Cbc_jll]
cbc_path = "/usr/local/Cellar/cbc/2.10.5/bin/cbc"
libCbc_path = "/usr/local/Cellar/cbc/2.10.5/lib/libCbc.3.10.5"
libOsiCbc_path = "/usr/local/Cellar/cbc/2.10.5/lib/libOsiCbc.3.10.5"
libcbcsolver_path = "/usr/local/Cellar/cbc/2.10.5/lib/libCbcSolver.3.10.5"
```

> **Info**
>
> Note that capitalization matters, so libcbcsolver_path corresponds to libCbcSolver.3.10.5.

Override entire artifact

To use the homebrew install as our custom binary we add the following to ~/.julia/artifacts/Overrides.toml:

```
# Override for Cbc_jll
e481bc81db5e229ba1f52b2b4bd57484204b1b06 = "/usr/local/Cellar/cbc/2.10.5"
```

Chapter 34

Style Guide

34.1   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 (for example, 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 (that is, 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 auto-formatting tool.

We use the options contained in `.JuliaFormatter.toml`.

To format code, cd to the JuMP directory, then run:

```
] add JuliaFormatter@1
using JuliaFormatter
format("docs")
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.

Abstract types and composition

Specifying types for method arguments is mostly optional in Julia. The benefit of abstract method arguments is that it enables functions and types from one package to be used with functions and types from another package via multiple dispatch.

However, abstractly typed methods have two main drawbacks:

1. It's possible to find out that you are working with unexpected types deep in the call chain, potentially leading to hard-to-diagnose `MethodErrors`.

2. Untyped function arguments can lead to correctness problems if the user's choice of input type does not satisfy the assumptions made by the author of the function.

As a motivating example, consider the following function:

```julia
julia> function my_sum(x)
           y = 0.0
           for i in 1:length(x)
               y += x[i]
           end
           return y
       end
my_sum (generic function with 1 method)
```

This function contains a number of implicit assumptions about the type of x:

- x supports 1-based `getindex` and implements `length`

- The element type of x supports addition with `0.0`, and then with the result of `x + 0.0`.

> **Info**
>
> As a motivating example for the second point, `VariableRef` plus `Float64` produces an `AffExpr`. Do not assume that `+(::A, ::B)` produces an instance of the type A or B.

`my_sum` works as expected if the user passes in `Vector{Float64}`:

```julia
julia> my_sum([1.0, 2.0, 3.0])
6.0
```

but it doesn't respect input types, for example returning a `Float64` if the user passes `Vector{Int}`:

```
julia> my_sum([1, 2, 3])
6.0
```

but it throws a `MethodError` if the user passes `String`:

```
julia> my_sum("abc")
ERROR: MethodError: no method matching +(::Float64, ::Char)
[...]
```

This particular `MethodError` is hard to debug, particularly for new users, because it mentions +, `Float64`, and `Char`, none of which were called or passed by the user.

Dealing with `MethodErrors`

This section diverges from the Julia style guide, as well as other common guides like SciML. The following suggestions are intended to provide a friendlier experience for novice Julia programmers, at the cost of limiting the power and flexibility of advanced Julia programmers.

Code should follow the `MethodError` principle:

> **The MethodError principle**
>
> 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()`s with the wrong type.",
    )
end
public_function(x) = _internal_function(x)
```

Dealing with correctness

Dealing with correctness is harder, because Julia has no way of formally specifying interfaces that abstract types must implement. Instead, here are two options that you can use when writing and interacting with generic code:

Option 1: use concrete types and let users extend new methods.

In this option, explicitly restrict input arguments to concrete types that are tested and have been validated for correctness. For example:

```julia
julia> function my_sum_option_1(x::Vector{Float64})
           y = 0.0
           for i in 1:length(x)
               y += x[i]
           end
           return y
       end
my_sum_option_1 (generic function with 1 method)

julia> my_sum_option_1([1.0, 2.0, 3.0])
6.0
```

Using concrete types satisfies the `MethodError` principle:

```julia
julia> my_sum_option_1("abc")
ERROR: MethodError: no method matching my_sum_option_1(::String)
```

and it allows other types to be supported in future by defining new methods:

```julia
julia> function my_sum_option_1(x::Array{T,N}) where {T<:Number,N}
           y = zero(T)
           for i in eachindex(x)
               y += x[i]
           end
           return y
       end
my_sum_option_1 (generic function with 2 methods)
```

Importantly, these methods do not have to be defined in the original package.

> **Info**
>
> Some usage of abstract types is okay. For example, in `my_sum_option_1`, we allowed the element type, `T`, to be a subtype of `Number`. This is fairly safe, but it still has an implicit assumption that T supports `zero(T)` and `+(::T, ::T)`.

Option 2: program defensively, and validate all assumptions.

An alternative is to program defensively, and to rigorously document and validate all assumptions that the code makes. In particular:

1. All assumptions on abstract types that aren't guaranteed by the definition of the abstract type (for example, optional methods without a fallback) should be documented.

2. If practical, the assumptions should be checked in code, and informative error messages should be provided to the user if the assumptions are not met. In general, these checks may be expensive, so you should prefer to do this once, at the highest level of the call-chain.

3. Tests should cover for a range of corner cases and argument types.

For example:

```julia
"""
    test_my_sum_defensive_assumptions(x::AbstractArray{T}) where {T}

Test the assumptions made by `my_sum_defensive`.
"""
function test_my_sum_defensive_assumptions(x::AbstractArray{T}) where {T}
    try
        # Some types may not define zero.
        @assert zero(T) isa T
        # Check iteration supported
        @assert iterate(x) isa Union{Nothing,Tuple{T,Int}}
        # Check that + is defined
        @assert +(zero(T), zero(T)) isa Any
    catch err
        error(
            "Unable to call my_sum_defensive(::$(typeof(x))) because " *
            "it failed an internal assumption",
        )
    end
    return
end


"""
    my_sum_defensive(x::AbstractArray{T}) where {T}

Return the sum of the elements in the abstract array `x`.

## Assumptions

This function makes the following assumptions:

 * That `zero(T)` is defined
 * That `x` supports the iteration interface
 * That  `+(::T, ::T)` is defined
"""
function my_sum_defensive(x::AbstractArray{T}) where {T}
    test_my_sum_defensive_assumptions(x)
    y = zero(T)
    for xi in x
        y += xi
    end
    return y
end

# output

my_sum_defensive
```

This function works on `Vector{Float64}`:

```julia
julia> my_sum_defensive([1.0, 2.0, 3.0])
6.0
```

as well as `Matrix{Rational{Int}}`:

```julia
julia> my_sum_defensive([(1//2) + (4//3)im; (6//5) + (7//11)im])
17//10 + 65//33*im
```

and it throws an error when the assumptions aren't met:

```julia
julia> my_sum_defensive(['a', 'b', 'c'])
ERROR: Unable to call my_sum_defensive(::Vector{Char}) because it failed an internal assumption
[...]
```

As an alternative, you may choose not to call `test_my_sum_defensive_assumptions` within `my_sum_defensive`, and instead ask users of `my_sum_defensive` to call it in their tests.

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, for example, "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, for example, 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 (for example, indx). Use abbreviations consistently within a body of code (for example, 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.

`@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++). Symbols 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 @enums when applicable, for example, for reporting statuses. Use strings to provide long-form additional information like error messages.

Use of Symbol should typically be reserved for identifiers, for example, 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, for example, (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 (for example, 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

function runtests()
    for name in names(@__MODULE__; all = true)
        if startswith("$(name)", "test_")
            @testset "$(name)" begin
                getfield(@__MODULE__, name)()
            end
        end
    end
    return
end

_helper_function() = 2
```

```
function test_addition()
    @test 1 + 1 == _helper_function()
    return
end

end # module 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.

Chapter 35

Roadmap

35.1 Development roadmap

The JuMP developers have compiled this roadmap document to share their plans and goals with the JuMP community. Contributions to roadmap issues are especially invited.

Most of these issues will require changes to both JuMP and MathOptInterface, and are non-trivial in their implementation. They are in no particular order, but represent broad themes that we see as areas in which JuMP could be improved.

- Make nonlinear programming a first-class citizen. There have been many issues and discussions about this: currently nonlinear constraints are handled through a `MOI.NLPBlock` and have various limitations and restrictions.

    - https://github.com/jump-dev/JuMP.jl/issues/1185
    - https://github.com/jump-dev/JuMP.jl/issues/1198
    - https://github.com/jump-dev/JuMP.jl/issues/2788
    - https://github.com/jump-dev/MathOptInterface.jl/issues/846
    - https://github.com/jump-dev/MathOptInterface.jl/issues/1397

- Add support for coefficient types other than `Float64`: https://github.com/jump-dev/JuMP.jl/issues/2025 Since the very beginning, JuMP has hard-coded the coefficient type as `Float64`. This has made it impossible to support solvers which can use other types such as `BigFloat` or `Rational{BigInt}`.

- Add support for constraint programming: https://github.com/jump-dev/JuMP.jl/issues/2227 JuMP has a strong focus on linear, conic and nonlinear optimization problems. We want to add better support for constraint programming.

- **Done #3176** Add support for multiobjective problems: https://github.com/jump-dev/JuMP.jl/issues/2099 JuMP is restricted to problems with scalar-valued objectives. We want to extend this to vector-valued problems.

- Refactor the internal code of JuMP's macros. The code in `src/macros.jl` is some of the oldest part of JuMP and is difficult to read, modify, and extend. We should overhaul the internals of JuMP's macros–without making user-visible breaking changes–to improve their long-term maintainability.

Chapter 36

Checklists

36.1 Checklists

The purpose of this page is to collate a series of checklists for commonly performed changes to the source code of JuMP.

In each case, copy the checklist into the description of the pull request.

Making a release

In preparation for a release, use the following checklist. These steps can be done in the same commit, or separately. The last commit should have the message "Prep for vX.Y.Z."

```
## Pre-release

 - [ ] Check that the pinned packages in `docs/Project.toml` are updated. We pin
       the versions so that changes in the solvers (changes in printing, small
       numeric changes) do not break the printing of the JuMP docs in arbitrary
       commits.
 - [ ] Check that the `rev` fields in `docs/packages.toml` are updated. We pin
       the versions of solvers and extensions to ensure that changes to their
       READMEs do not break the JuMP docs in arbitrary commits, and to ensure
       that the versions are compatible with the latest JuMP and
       MathOptInterface releases.
 - [ ] Update `docs/src/changelog.md`
 - [ ] Run https://github.com/jump-dev/JuMP.jl/actions/workflows/extension-tests.yml
       using a `workflow_dispatch` trigger to check for any changes in JuMP that
       broke extensions.
 - [ ] Change the version number in `Project.toml`
 - [ ] Update the links in README.md
 - [ ] The commit messages in this PR do not contain `[ci skip]`
       (See [Documenter#965](https://github.com/JuliaDocs/Documenter.jl/issues/965).)

## The release

 - [ ] After merging this pull request, comment `@JuliaRegistrator register` in
       the GitHub commit. This should automatically publish a new version to the
       Julia registry, as well as create a tag, and rebuild the documentation
       for this tag.

       These steps can take quite a bit of time (1 hour or more), so don't be
```

```
        surprised if the new documentation takes a while to appear. In addition,
        the links in the README will be broken until JuliaHub fetches the new
        version on their servers.

## Post-release

 - [ ] Once the tag is created, update the relevant `release-` branch. The latest
        release branch at the time of writing is `release-1.0` (we haven't
        back-ported any patches that needed to create a `release-1.Y` branch). To
        to update the release branch with the v1.10.0 tag, do:
        ```
        git checkout release-1.0
        git pull
        git merge v1.10.0
        git push
        ```
```

Adding a new solver to the documentation

Use the following checklist when adding a new solver to the JuMP documentation.

```
## Basic

 - [ ] Check that the solver is a registered Julia package
 - [ ] Check that the solver supports the long-term support release of Julia
 - [ ] Check that the solver has a MathOptInterface wrapper
 - [ ] Check that the tests call `MOI.Test.runtests`. Some test excludes are
        permissible, but the reason for skipping a particular test should be
        documented.
 - [ ] Check that the README and/or documentation provides an example of how to
        use the solver with JuMP

## Documentation

 - [ ] Add a new row to the table in `docs/src/installation.md`

## Optional

 - [ ] Add package metadata to `docs/packages.toml`
```

Part VII

MathOptInterface

Chapter 37

Introduction

37.1 Introduction

> **Warning**
>
> This documentation in this section is a copy of the official MathOptInterface documentation available at https://jump.dev/MathOptInterface.jl/v1.16.0. 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 **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 **Background** section contains articles on the theory behind MathOptInterface. Look here if you want to understand why, rather than how.

- 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:

```
@article{legat2021mathoptinterface,
    title={{MathOptInterface}: a data structure for mathematical optimization problems},
    author={Legat, Beno{\^\i}t and Dowson, Oscar and Garcia, Joaquim Dias and Lubin, Miles},
    journal={INFORMS Journal on Computing},
    year={2021},
    doi={10.1287/ijoc.2021.1067},
    publisher={INFORMS}
}
```

37.2 Motivation

MathOptInterface (MOI) is a replacement for MathProgBase, the first-generation abstraction layer for mathematical optimization previously used by JuMP and Convex.jl.

To address a number of limitations of MathProgBase, MOI is designed to:

- Be simple and extensible

    - unifying linear, quadratic, and conic optimization,

    - seamlessly facilitating extensions to essentially arbitrary constraints and functions (for example, 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)

    - by using multiple dispatch and avoiding requiring containers of non-concrete types

- Allow a solver to return multiple results (for example, a pool of solutions)

- Allow a solver to return extra arbitrary information via attributes (for example, 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

Chapter 38

Tutorials

38.1 Solving a problem using MathOptInterface

In this tutorial we demonstrate how to use MathOptInterface to solve the binary-constrained knapsack problem:

$$
\begin{aligned}
\max\ & c^\top x \\
s.t.\ & w^\top x \le C \\
& x_i \in \{0, 1\}, \quad \forall i = 1, \dots, n
\end{aligned}
$$

Required packages

Load the MathOptInterface module and define the shorthand MOI:

```
import MathOptInterface as MOI
```

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, for example, did it stop because of a time limit or did it stop because it found the optimal solution?

```julia
julia> MOI.get(optimizer, MOI.TerminationStatus())
OPTIMAL::TerminationStatusCode = 1
```

Looks like we found an optimal solution.

Understand what solution was returned

```julia
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
julia> MOI.get(optimizer, MOI.ObjectiveValue())
6.0
```

Query the primal solution

And what is the value of the variables x?

```julia
julia> MOI.get(optimizer, MOI.VariablePrimal(), x)
3-element Vector{Float64}:
 1.0
 1.0
 1.0
```

38.2   Implementing a solver interface

This guide outlines the basic steps to implement an interface to MathOptInterface for a new solver.

> **Danger**
>
> 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 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

Before starting on your wrapper, you should do some background research and make the solver accessible via Julia.

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, for example, 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.

Documentation

Your package must include documentation explaining how to use the package. The easiest approach is to include documentation in your `README.md`. A more involved option is to use Documenter.jl.

Examples of packages with README-based documentation include:

- Cbc.jl

- HiGHS.jl

- SCS.jl

Examples of packages with Documenter-based documentation include:

- Alpine.jl

- COSMO.jl

- Juniper.jl

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 (for example, 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`.

```julia
import MathOptInterface as MOI

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:

```julia
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 `Optimizer`s must implement the following methods:

- empty!

- is_empty

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
AbsoluteGapTolerance	Yes	Yes	Yes
RelativeGapTolerance	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 Unions:

```julia
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: incremental modification?

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 optimize! or copy_to function call. Because these functions see 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 non-incremental solvers are AmplNLWriter.jl and SCS.jl

It is possible for a solver 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 the one-shot optimize!.

The non-incremental interface

There are two ways to implement the non-incremental interface. The first uses a two-argument version of optimize!, the second implements copy_to followed by the one-argument version of optimize!.

If your solver does not support modification, and requires all data to solve the problem in a single function call, you should implement the "one-shot" optimize!.

- optimize!(::ModelLike, ::ModelLike)

If your solver separates data loading and the actual optimization into separate steps, implement the `copy_to` interface.

- `copy_to(::ModelLike, ::ModelLike)`

- `optimize!(::ModelLike)`

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`

- `optimize!(::ModelLike)`

> **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, for example, 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
```

Attribute	get	set	supports
VariableName	Yes	Yes	Yes
ConstraintName	Yes	Yes	Yes

Names

Regardless of which interface you implement, you have the option of implementing the Name attribute for variables and constraints:

If you implement names, you must also implement the following three methods:

```julia
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_bounds(model, 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

Here are some other points to be aware of when writing your wrapper.

Unsupported constraints at runtime

In some cases, your solver may support a particular type of constraint (for example, quadratic constraints), but only if the data meets some condition (for example, it is convex).

In this case, declare that you `support` 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 must expect that functions such as ScalarAffineFunction and VectorQuadraticFunction may contain duplicate coefficients.

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 must be copied immediately to internal data structures. Solvers may not modify any input vectors and must assume that input vectors will not be modified by users in the future.

This applies, for example, to the terms vector in ScalarAffineFunction. Vectors returned to the user, for example, via ObjectiveFunction or ConstraintFunction attributes, must 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.

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)
```

38.3   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 the simplify implementation process.

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:

```julia
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:

```julia
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:

```julia
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
```

38.4   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 must 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 maps 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 is 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:

```julia
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:

```julia
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 must 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 must 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 must 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 must 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
```

In order for the user to be able to access the function and set of the original constraint, the bridge needs to implement getters for the `ConstraintFunction` and `ConstraintSet` attributes:

```
function get(
    model::MOI.ModelLike,
    attr::MOI.ConstraintFunction,
    bridge::SignBridge,
)
    return -MOI.get(model, attr, bridge.constraint)
end

function get(
    model::MOI.ModelLike,
    attr::MOI.ConstraintSet,
    bridge::SignBridge,
)
    set = MOI.get(model, attr, bridge.constraint)
    return MOI.LessThan(-set.lower)
end
```

> **Warning**
>
> Alternatively, one could store the original function and set in `SignBridge` during `Bridges.Constraint.bridge_constraint` to make these getters simpler and more efficient. On the other hand, this will increase the memory footprint of the bridges as the garbage collector won't be able to delete that object. The convention is to not store the function in the bridge and not care too much about the efficiency of these getters. If the user needs efficient getters for `ConstraintFunction` then they should use a `Utilities.CachingOptimizer`.

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 (for example, replacing the value of the coefficient of a variable in the affine function) must 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 must be deleted too.

```
function delete(model::ModelLike, bridge::SignBridge)
    delete(model, bridge.constraint)
    return
end
```

38.5 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
MOI.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
(2) + (1) MOI.VariableIndex(1)
```

However, you can also use operators to build the same scalar function:

```
julia> f = x + 2
(2) + (1) MOI.VariableIndex(1)
```

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
(0) + 1.0 MOI.VariableIndex(1)²

julia> f * f  # (x + 2)²
(4) + (2) MOI.VariableIndex(1) + (2) MOI.VariableIndex(1) + 1.0 MOI.VariableIndex(1)²

julia> f^2  # (x + 2)² too
(4) + (2) MOI.VariableIndex(1) + (2) MOI.VariableIndex(1) + 1.0 MOI.VariableIndex(1)²
```

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])
⌈                            ⌉
│(2) + (1) MOI.VariableIndex(1)│
│(4) + (2) MOI.VariableIndex(1)│
⌊                            ⌋
```

> **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
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
julia> MOI.Utilities.canonical(f + f) # Returns 2x + 4
(4) + (2) MOI.VariableIndex(1)
```

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
julia> MOI.Utilities.scalarize(g) # Returns a vector [f, 2 * f].
2-element Vector{MathOptInterface.ScalarAffineFunction{Int64}}:
 (2) + (1) MOI.VariableIndex(1)
 (4) + (2) MOI.VariableIndex(1)
```

> **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
julia> MOI.output_dimension(g)
2
```

38.6 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 (for example, the `Optimizers`), which implement methods of functions defined in `MOI` (for example, `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 (that is, 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):

```
import GLPK
import MathOptInterface as MOI

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),
```
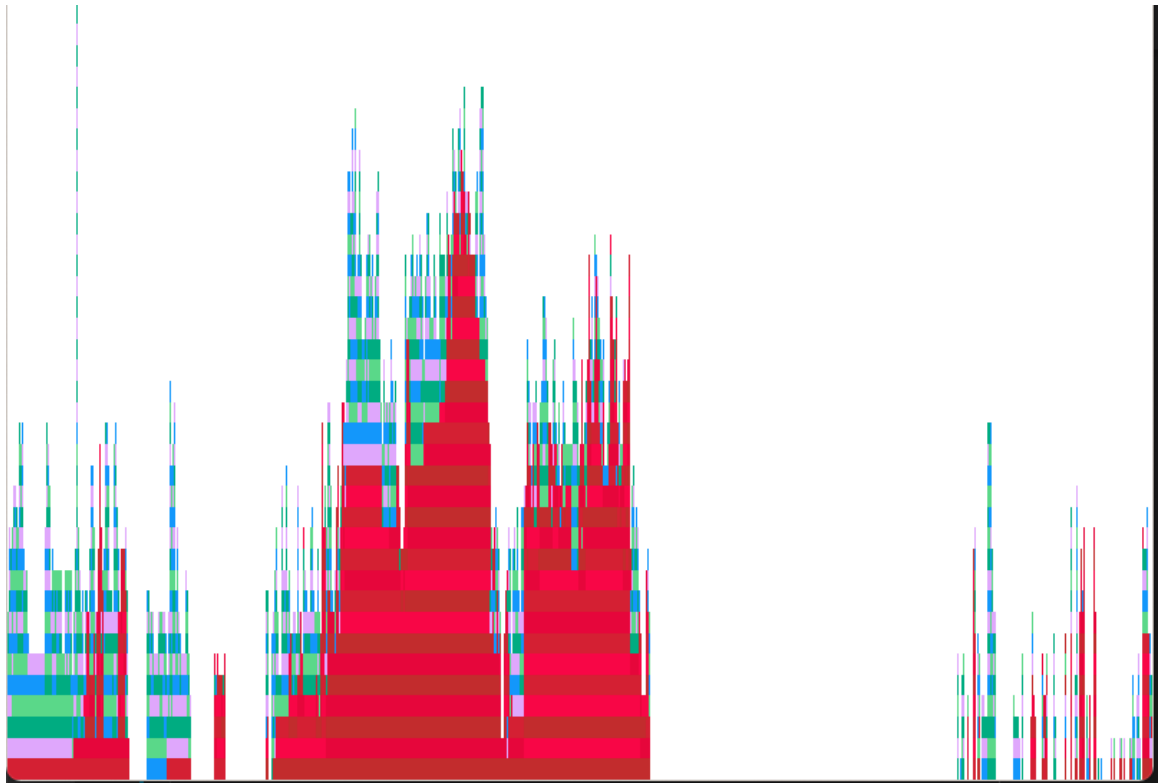
Figure 38.1: flamegraph

```
    )
    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.

Chapter 39

Manual

39.1 Standard form problem

MathOptInterface represents optimization problems in the standard form:

$$\min_{x \in \mathbb{R}^n} \quad f_0(x) \tag{39.1}$$

$$\text{s.t.} \quad f_i(x) \in \mathcal{S}_i \qquad\qquad i = 1 \ldots m \tag{39.2}$$

where:

- the functions $f_0, f_1, \ldots, f_m$ are specified by `AbstractFunction` objects
- the sets $\mathcal{S}_1, \ldots, \mathcal{S}_m$ 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:

Function	Description
`VariableIndex`	x_j, the projection onto a single coordinate defined by a variable index j.
`VectorOfVariables`	The projection onto multiple coordinates (that is, extracting a sub-vector).
`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:

Set	Description
LessThan(u)	$(-\infty, u]$
GreaterThan(l)	$[l, \infty)$
EqualTo(v)	$\{v\}$
Interval(l, u)	$[l, u]$
Integer()	$\mathbb{Z}$
ZeroOne()	$\{0, 1\}$
Semicontinuous(l, u)	$\{0\} \cup [l, u]$
Semiinteger(l, u)	$\{0\} \cup \{l, l+1, \ldots, u-1, u\}$

Vector cones

The vector-valued set types implemented in MathOptInterface.jl are:

Set	Description
Reals(d)	$\mathbb{R}^d$
Zeros(d)	0^d
Nonnegatives(d)	$\{x \in \mathbb{R}^d : x \geq 0\}$
Nonpositives(d)	$\{x \in \mathbb{R}^d : x \leq 0\}$
SecondOrderCone(d)	$\{(t, x) \in \mathbb{R}^d : t \geq \|x\|_2\}$
RotatedSecondOrderCone(d)	$\{(t, u, x) \in \mathbb{R}^d : 2tu \geq \|x\|_2^2, t \geq 0, u \geq 0\}$
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(d)	$\{(t, x) \in \mathbb{R}^{1+n} : x \geq 0, t \leq \sqrt[n]{x_1 x_2 \cdots x_n}\}$ where n is $d - 1$
PowerCone(α)	$\{(x, y, z) \in \mathbb{R}^3 : x^\alpha y^{1-\alpha} \geq |z|, x \geq 0, y \geq 0\}$
DualPowerCone(α)	$\{(u, v, w) \in \mathbb{R}^3 : \left(\frac{u}{\alpha}\right(^\alpha \left(\frac{v}{1-\alpha}\right)^{1-\alpha} \geq |w|, u, v \geq 0\}$
NormOneCone(d)	$\{(t, x) \in \mathbb{R}^d : t \geq \sum_i |x_i|\}$
NormInfinityCone(d)	$\{(t, x) \in \mathbb{R}^d : t \geq \max_i |x_i|\}$
RelativeEntropyCone(d)	$\{(u, v, w) \in \mathbb{R}^d : u \geq \sum_i w_i \log(\frac{w_i}{v_i}), v_i \geq 0, w_i \geq 0\}$
HyperRectangle(l, u)	$\{x \in \bar{\mathbb{R}}^d : x_i \in [l_i, u_i] \forall i = 1, \ldots, d\}$
NormCone(p, d)	"{ (t,x) \in \mathbb{R}^{d} : t \ge \left(\sum\limits_i

Matrix cones

The matrix-valued set types implemented in MathOptInterface.jl are:

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.

Set	Description
RootDetConeTriangle(d)	$\{(t, X) \in \mathbb{R}^{1+d(1+d)/2} : t \leq \det(X)^{1/d}, X$ is the upper triangle of a PSD matrix$\}$
RootDetConeSquare(d)	$\{(t, X) \in \mathbb{R}^{1+d^2} : t \leq \det(X)^{1/d}, X$ is a PSD matrix$\}$
PositiveSemidefiniteConeTriangle(d)	$\{X \in \mathbb{R}^{d(d+1)/2} : X$ is the upper triangle of a PSD matrix$\}$
PositiveSemidefiniteConeSquare(d)	$\{X \in \mathbb{R}^{d^2} : X$ is a PSD matrix$\}$
ScaledPositiveSemidefiniteConeTriangle(d)	$\{X \in \mathbb{R}^{d(d+1)/2} : X$ is a PSD matrix$\}$
LogDetConeTriangle(d)	$\{(t, u, X) \in \mathbb{R}^{2+d(1+d)/2} : t \leq u \log(\det(X/u)), X$ is the upper triangle of a PSD matrix, $u > 0\}$
LogDetConeSquare(d)	$\{(t, u, X) \in \mathbb{R}^{2+d^2} : t \leq u \log(\det(X/u)), X$ is a PSD matrix, $u > 0\}$
NormSpectralCone(r, c)	$\{(t, X) \in \mathbb{R}^{1+r \times c} : t \geq \sigma_1(X), X$ is a $r \times c$ matrix$\}$
NormNuclearCone(r, c)	$\{(t, X) \in \mathbb{R}^{1+r \times c} : t \geq \sum_i \sigma_i(X), X$ is a $r \times c$ matrix$\}$
HermitianPositiveSemidefiniteConeTriangle(d)	The cone of Hermitian positive semidefinite matrices, with side_dimension rows and columns.

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

Other sets are vector-valued, with a particular combinatorial structure. Read their docstrings for more information on how to interpret them.

Set	Description
SOS1	A Special Ordered Set (SOS) of Type I
SOS2	A Special Ordered Set (SOS) of Type II
Indicator	A set to specify an indicator constraint
Complements	A set to specify a mixed complementarity constraint
AllDifferent	The all_different global constraint
BinPacking	The bin_packing global constraint
Circuit	The circuit global constraint
CountAtLeast	The at_least global constraint
CountBelongs	The nvalue global constraint
CountDistinct	The distinct global constraint
CountGreaterThan	The count_gt global constraint
Cumulative	The cumulative global constraint
Path	The path global constraint
Table	The table global constraint

39.2 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 (for example, by adding variables and constraints). Objects that implement the model API must 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 (for example, 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
>
> ```
> import MathOptInterface as MOI
> ```
>
> 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 docstrings 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

39.3   Variables

Add a variable

Use add_variable to add a single variable.

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

add_variable returns a VariableIndex type, which is used to refer to the added variable in other calls.

Check if a VariableIndex is valid using is_valid.

```julia
julia> MOI.is_valid(model, x)
true
```

Use add_variables to add a number of variables.

```julia
julia> y = MOI.add_variables(model, 2)
2-element Vector{MathOptInterface.VariableIndex}:
 MOI.VariableIndex(2)
 MOI.VariableIndex(3)
```

> **Warning**
>
> The integer does not necessarily correspond to the column inside an optimizer.

Delete a variable

Delete a variable using delete.

```julia
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
julia> MOI.set(model, MOI.VariableName(), x, "var_x")

julia> MOI.get(model, MOI.VariableName(), x)
"var_x"
```

39.4 Constraints

Add a constraint

Use add_constraint to add a single constraint.

```julia
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 is used to refer to the added constraint in other calls.

Check if a ConstraintIndex is valid using is_valid.

```julia
julia> MOI.is_valid(model, c)
true
```

Use add_constraints to add a number of constraints of the same type.

```julia
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
julia> MOI.supports_constraint(
           model,
           MOI.VariableIndex,
           MOI.GreaterThan{Float64},
       )
true
```

Delete a constraint

Use delete to delete a constraint.

```julia
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
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 non-negative (resp. non-positive) real numbers.

Linear constraints

By convention, solvers are not expected to support nonzero constant terms in the ScalarAffineFunctions the first four rows of the preceding table because they are redundant with the parameters of the sets. For example, encode $2x + 1 \leq 2$ 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 will throw respectively LowerBoundAlreadySet or UpperBoundAlreadySet.

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

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
$\frac{1}{2} x^T Q x + a^T x + b \geq 0$	ScalarQuadraticFunction	GreaterThan
$\frac{1}{2} x^T Q x + a^T x + b \leq 0$	ScalarQuadraticFunction	LessThan
$\frac{1}{2} x^T Q x + a^T x + b = 0$	ScalarQuadraticFunction	EqualTo
Bilinear matrix inequality	VectorQuadraticFunction	PositiveSemidefiniteCone...

> **Note**
>
> For more details on the internal format of the quadratic functions see ScalarQuadraticFunction or VectorQuadraticFunction.

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, that is:

- `@variable(m, 0 <= x <= 1)` becomes `VariableIndex-in-LessThan` and `VariableIndex-in-GreaterThan`.

39.5 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 stop 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, 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 infeasibility, 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.

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

> **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.

Local search solvers

Nonlinear programming solvers fall into this category. It also includes non-global tree search solvers like Juniper.

Querying solution attributes

Some solvers will not implement every solution attribute. Therefore, a call like MOI.get(model, MOI.SolveTimeSec()) may throw an UnsupportedAttribute error.

If you need to write code that is agnostic to the solver (for example, you are writing a library that an end-user passes their choice of solver to), you can work-around this problem using a try-catch:

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	*	*

```julia
function get_solve_time(model)
    try
        return MOI.get(model, MOI.SolveTimeSec())
    catch err
        if err isa MOI.UnsupportedAttribute
            return NaN  # Solver doesn't support. Return a placeholder value.
        end
        rethrow(err)  # Something else went wrong. Rethrow the error
    end
end
```

If, after careful profiling, you find that the `try-catch` is taking a significant portion of your runtime, you can improve performance by caching the result of the `try-catch`:

```julia
mutable struct CachedSolveTime{M}
    model::M
    supports_solve_time::Bool
    CachedSolveTime(model::M) where {M} = new(model, true)
end

function get_solve_time(model::CachedSolveTime)
    if !model.supports_solve_time
        return NaN
    end
    try
        return MOI.get(model, MOI.SolveTimeSec())
    catch err
        if err isa MOI.UnsupportedAttribute
            model.supports_solve_time = false
            return NaN
        end
        rethrow(err)  # Something else went wrong. Rethrow the error
    end
end
```

39.6   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
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
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 (that is, c) is no longer valid.

```julia
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
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
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
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
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
```

ScalarConstantChange can also be used to modify the objective function by passing an instance of ObjectiveFunction:

```
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
```

ScalarCoefficientChange can also be used to modify the objective function by passing an instance of ObjectiveFunction.

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 40

Background

40.1 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:

$$\min_{x \in \mathbb{R}^n} \quad a_0^T x + b_0 \tag{40.1}$$

$$\text{s.t.} \quad A_i x + b_i \in \mathcal{C}_i \qquad i = 1 \ldots m \tag{40.2}$$

and the dual is a maximization problem in standard conic form:

$$\max_{y_1, \ldots, y_m} \quad -\sum_{i=1}^{m} b_i^T y_i + b_0 \tag{40.3}$$

$$\text{s.t.} \quad a_0 - \sum_{i=1}^{m} A_i^T y_i = 0 \tag{40.4}$$

$$y_i \in \mathcal{C}_i^* \qquad i = 1 \ldots m \tag{40.5}$$

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:

$$\max_{x \in \mathbb{R}^n} \quad a_0^T x + b_0 \tag{40.6}$$

$$\text{s.t.} \quad A_i x + b_i \in \mathcal{C}_i \qquad i = 1 \ldots m \tag{40.7}$$

and the dual is a minimization problem in standard conic form:

$$\min_{y_1,\ldots,y_m} \qquad \sum_{i=1}^{m} b_i^T y_i + b_0 \tag{40.8}$$

$$\text{s.t.} \qquad a_0 + \sum_{i=1}^{m} A_i^T y_i = 0 \tag{40.9}$$

$$y_i \in \mathcal{C}_i^* \qquad\qquad i = 1 \ldots m \tag{40.10}$$

A linear inequality constraint $a^T x + b \geq c$ is equivalent to $a^T x + b - c \in \mathbb{R}_+$, and $a^T x + b \leq c$ is equivalent to $a^T x + b - c \in \mathbb{R}_-$. Variable-wise constraints are 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:

$$\min_{x \in \mathbb{R}^n} \qquad a_0^T x + b_0 \tag{40.11}$$

$$\text{s.t.} \qquad A_1 x \geq b_1 \tag{40.12}$$

$$A_2 x \leq b_2 \tag{40.13}$$

$$A_3 x = b_3 \tag{40.14}$$

By applying the stated transformations to conic form, taking the dual, and transforming back into linear inequality form, one obtains the following dual:

$$\max_{y_1,y_2,y_3} \qquad b_1^T y_1 + b_2^T y_2 + b_3^T y_3 + b_0 \tag{40.15}$$

$$\text{s.t.} \qquad A_1^T y_1 + A_2^T y_2 + A_3^T y_3 = a_0 \tag{40.16}$$

$$y_1 \geq 0 \tag{40.17}$$

$$y_2 \leq 0 \tag{40.18}$$

For maximization LPs, the MOI primal form can be stated as:

$$\max_{x \in \mathbb{R}^n} \qquad a_0^T x + b_0 \tag{40.19}$$

$$\text{s.t.} \qquad A_1 x \geq b_1 \tag{40.20}$$

$$A_2 x \leq b_2 \tag{40.21}$$

$$A_3 x = b_3 \tag{40.22}$$

and similarly, the dual is:

$$\min_{y_1,y_2,y_3} \qquad -b_1^T y_1 - b_2^T y_2 - b_3^T y_3 + b_0 \tag{40.23}$$

$$\text{s.t.} \qquad A_1^T y_1 + A_2^T y_2 + A_3^T y_3 = -a_0 \tag{40.24}$$

$$y_1 \geq 0 \tag{40.25}$$

$$y_2 \leq 0 \tag{40.26}$$

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

Quadratic Programs (QPs)

For quadratic programs with only affine conic constraints,

$$\min_{x \in \mathbb{R}^n} \quad \frac{1}{2} x^T Q_0 x + a_0^T x + b_0$$
$$\text{s.t.} \quad A_i x + b_i \in \mathcal{C}_i \qquad i = 1 \ldots m.$$

with cones $\mathcal{C}_i \subseteq \mathbb{R}^{m_i}$ for $i = 1, \ldots, m$, consider the Lagrangian function

$$L(x, y) = \frac{1}{2} x^T Q_0 x + a_0^T x + b_0 - \sum_{i=1}^{m} y_i^T (A_i x + b_i).$$

Let $z(y)$ denote $\sum_{i=1}^{m} A_i^T y_i - a_0$, the Lagrangian can be rewritten as

$$L(x, y) = \frac{1}{2} x^T Q_0 x - z(y)^T x + b_0 - \sum_{i=1}^{m} y_i^T b_i.$$

The condition $\nabla_x L(x, y) = 0$ gives

$$0 = \nabla_x L(x, y) = Q_0 x + a_0 - \sum_{i=1}^{m} y_i^T b_i$$

which gives $Q_0 x = z(y)$. This allows to obtain that

$$\min_{x \in \mathbb{R}^n} L(x, y) = -\frac{1}{2} x^T Q_0 x + b_0 - \sum_{i=1}^{m} y_i^T b_i$$

so the dual problem is

$$\max_{y_i \in \mathcal{C}_i^*} \min_{x \in \mathbb{R}^n} -\frac{1}{2} x^T Q_0 x + b_0 - \sum_{i=1}^m y_i^T b_i.$$

If Q_0 is invertible, we have $x = Q_0^{-1} z(y)$ hence

$$\min_{x \in \mathbb{R}^n} L(x, y) = -\frac{1}{2} z(y)^T Q_0^{-1} z(y) + b_0 - \sum_{i=1}^m y_i^T b_i$$

so the dual problem is

$$\max_{y_i \in \mathcal{C}_i^*} -\frac{1}{2} z(y)^T Q_0^{-1} z(y) + b_0 - \sum_{i=1}^m y_i^T b_i.$$

Quadratically Constrained Quadratic Programs (QCQPs)

Given a problem with both quadratic function and quadratic objectives:

$$\min_{x \in \mathbb{R}^n} \qquad \frac{1}{2} x^T Q_0 x + a_0^T x + b_0$$
$$\text{s.t.} \qquad \frac{1}{2} x^T Q_i x + a_i^T x + b_i \in \mathcal{C}_i \qquad\qquad i = 1 \dots m.$$

with cones $\mathcal{C}_i \subseteq \mathbb{R}$ for $i = 1 \dots m$, consider the Lagrangian function

$$L(x, y) = \frac{1}{2} x^T Q_0 x + a_0^T x + b_0 - \sum_{i=1}^m y_i \left(\frac{1}{2} x^T Q_i x + a_i^T x + b_i \right)$$

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 $\{z \in \mathbb{R} : z \leq 0\}$, 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.

Dual for square semidefinite matrices

The set `PositiveSemidefiniteConeTriangle` is a self-dual. That is, querying `ConstraintDual` of a `PositiveSemidefiniteConeT` constraint returns a vector that is itself a member of `PositiveSemidefiniteConeTriangle`.

However, the dual of `PositiveSemidefiniteConeSquare` is not so straight forward. This section explains the duality convention we use, and how it is derived.

> **Info**
>
> If you have a `PositiveSemidefiniteConeSquare` constraint, the result matrix A from `ConstraintDual` is not positive semidefinite. However, $A + A^\top$ is positive semidefinite.

Let $\mathcal{S}_+$ be the cone of symmetric semidefinite matrices in the $\frac{n(n+1)}{2}$ dimensional space of symmetric $\mathbb{R}^{n \times n}$ matrices. That is, $\mathcal{S}_+$ is the set `PositiveSemidefiniteConeTriangle`. It is well known that $\mathcal{S}_+$ is a self-dual proper cone.

Let $\mathcal{P}_+$ be the cone of symmetric semidefinite matrices in the n^2 dimensional space of $\mathbb{R}^{n \times n}$ matrices. That is $\mathcal{P}_+$ is the set `PositiveSemidefiniteConeSquare`.

In addition, let $\mathcal{D}_+$ be the cone of matrices A such that $A + A^\top \in \mathcal{P}_+$.

$\mathcal{P}_+$ is not proper because it is not solid (it is not n^2 dimensional), so it is not necessarily true that $\mathcal{P}_+^{**} = \mathcal{P}_+$.

However, this is the case, because we will show that $\mathcal{P}_+^* = \mathcal{D}_+$ and $\mathcal{D}_+^* = \mathcal{P}_+$.

First, let us see why $\mathcal{P}_+^* = \mathcal{D}_+$.

If B is symmetric, then

$$\langle A, B \rangle = \langle A^\top, B^\top \rangle = \langle A^\top, B \rangle$$

so

$$2\langle A, B \rangle = \langle A, B \rangle + \langle A^\top, B \rangle = \langle A + A^\top, B \rangle.$$

Therefore, $\langle A, B \rangle \geq 0$ for all $B \in \mathcal{P}_+$ if and only if $\langle A + A^\top, B \rangle \geq 0$ for all $B \in \mathcal{P}_+$. Since $A + A^\top$ is symmetric, and we know that $\mathcal{S}_+$ is self-dual, we have shown that $\mathcal{P}_+^*$ is the set of matrices A such that $A + A^\top \in \mathcal{P}_+$.

Second, let us see why $\mathcal{D}_+^* = \mathcal{P}_+$.

Since $A \in \mathcal{D}_+$ implies that $A^\top \in \mathcal{D}_+$, $B \in \mathcal{D}_+^*$ means that $\langle A + A^\top, B \rangle \geq 0$ for all $A \in \mathcal{D}_+$, and hence $B \in$
$mathcalP_+$.

To see why it should be symmetric, simply notice that if $B_{i,j} < B_{j,i}$, then $\langle A, B \rangle$ can be made arbitrarily small by setting $A_{i,j} = A_{i,j} + s$ and $A_{j,i} = A_{j,i} - s$, with s arbitrarily large, and A stays in $\mathcal{D}_+$ because $A + A^\top$ does not change.

Typically, the primal/dual pair for semidefinite programs is presented as:

$$\min \langle C, X \rangle \tag{40.27}$$
$$\text{s.t.} \ \ \langle A_k, X \rangle = b_k \forall k \tag{40.28}$$
$$X \in \mathcal{S}_+ \tag{40.29}$$

with the dual

$$\max \sum_k b_k y_k \tag{40.30}$$
$$\text{s.t.} \ \ C - \sum A_k y_k \in \mathcal{S}_+ \tag{40.31}$$

If we allow A_k to be non-symmetric, we should instead use:

$$\min \langle C, X \rangle \tag{40.32}$$
$$\text{s.t.} \ \ \langle A_k, X \rangle = b_k \forall k \tag{40.33}$$
$$X \in \mathcal{D}_+ \tag{40.34}$$

with the dual

$$\max \sum b_k y_k \tag{40.35}$$
$$\text{s.t.} \ \ C - \sum A_k y_k \in \mathcal{P}_+ \tag{40.36}$$

This is implemented as:

$$\min \langle C, Z \rangle + \langle C - C^\top, S \rangle \tag{40.37}$$
$$\text{s.t.} \ \ \langle A_k, Z \rangle + \langle A_k - A_k^\top, S \rangle = b_k \forall k \tag{40.38}$$
$$Z \in \mathcal{S}_+ \tag{40.39}$$

with the dual

$$\max \sum b_k y_k \tag{40.40}$$
$$\text{s.t.} \ \ C + C^\top - \sum (A_k + A_k^\top) y_k \in \mathcal{S}_+ \tag{40.41}$$
$$C - C^\top - \sum (A_k - A_k^\top) y_k = 0 \tag{40.42}$$

and we recover $Z = X + X^\top$.

40.2   Infeasibility certificates

When given a conic problem that is infeasible or unbounded, some solvers can produce a certificate of infeasibility. This page explains what a certificate of infeasibility is, and the related conventions that MathOptInterface adopts.

Conic duality

MathOptInterface uses conic duality to define infeasibility certificates. A full explanation is given in the section Duality, but here is a brief overview.

Minimization problems

For a minimization problem in geometric conic form, the primal is:

$$\min_{x \in \mathbb{R}^n} \quad a_0^\top x + b_0 \tag{40.43}$$

$$\text{s.t.} \quad A_i x + b_i \in \mathcal{C}_i \qquad i = 1 \ldots m, \tag{40.44}$$

and the dual is a maximization problem in standard conic form:

$$\max_{y_1, \ldots, y_m} \quad -\sum_{i=1}^m b_i^\top y_i + b_0 \tag{40.45}$$

$$\text{s.t.} \quad a_0 - \sum_{i=1}^m A_i^\top y_i = 0 \tag{40.46}$$

$$y_i \in \mathcal{C}_i^* \qquad i = 1 \ldots m, \tag{40.47}$$

where each $\mathcal{C}_i$ is a closed convex cone and $\mathcal{C}_i^*$ is its dual cone.

Maximization problems

For a maximization problem in geometric conic form, the primal is:

$$\max_{x \in \mathbb{R}^n} \quad a_0^\top x + b_0 \tag{40.48}$$

$$\text{s.t.} \quad A_i x + b_i \in \mathcal{C}_i \qquad i = 1 \ldots m, \tag{40.49}$$

and the dual is a minimization problem in standard conic form:

$$\min_{y_1, \ldots, y_m} \quad \sum_{i=1}^m b_i^\top y_i + b_0 \tag{40.50}$$

$$\text{s.t.} \quad a_0 + \sum_{i=1}^m A_i^\top y_i = 0 \tag{40.51}$$

$$y_i \in \mathcal{C}_i^* \qquad i = 1 \ldots m. \tag{40.52}$$

Unbounded problems

A problem is unbounded if and only if:

1. there exists a feasible primal solution

2. the dual is infeasible.

A feasible primal solution—if one exists—can be obtained by setting ObjectiveSense to FEASIBILITY_SENSE before optimizing. Therefore, most solvers stop after they prove the dual is infeasible via a certificate of dual infeasibility, but before they have found a feasible primal solution. This is also the reason that MathOptInterface defines the DUAL_INFEASIBLE status instead of UNBOUNDED.

A certificate of dual infeasibility is an improving ray of the primal problem. That is, there exists some vector d such that for all $\eta > 0$:

$$A_i(x + \eta d) + b_i \in \mathcal{C}_i, \ \ i = 1 \ldots m,$$

and (for minimization problems):

$$a_0^\top (x + \eta d) + b_0 < a_0^\top x + b_0,$$

for any feasible point x. The latter simplifies to $a_0^\top d < 0$. For maximization problems, the inequality is reversed, so that $a_0^\top d > 0$.

If the solver has found a certificate of dual infeasibility:

- TerminationStatus must be DUAL_INFEASIBLE

- PrimalStatus must be INFEASIBILITY_CERTIFICATE

- VariablePrimal must be the corresponding value of d

- ConstraintPrimal must be the corresponding value of $A_i d$

- ObjectiveValue must be the value $a_0^\top d$. Note that this is the value of the objective function at d, ignoring the constant b_0.

> **Note**
>
> The choice of whether to scale the ray d to have magnitude 1 is left to the solver.

Infeasible problems

A certificate of primal infeasibility is an improving ray of the dual problem. However, because infeasibility is independent of the objective function, we first homogenize the primal problem by removing its objective.

For a minimization problem, a dual improving ray is some vector d such that for all $\eta > 0$:

$$-\sum_{i=1}^{m} A_i^\top (y_i + \eta d_i) = 0 \tag{40.53}$$

$$(y_i + \eta d_i) \in \mathcal{C}_i^* \qquad\qquad i = 1 \ldots m, \tag{40.54}$$

and:

$$-\sum_{i=1}^{m} b_i^\top (y_i + \eta d_i) > -\sum_{i=1}^{m} b_i^\top y_i,$$

for any feasible dual solution y. The latter simplifies to $-\sum_{i=1}^{m} b_i^\top d_i > 0$. For a maximization problem, the inequality is $\sum_{i=1}^{m} b_i^\top d_i < 0$. (Note that these are the same inequality, modulo a - sign.)

If the solver has found a certificate of primal infeasibility:

- `TerminationStatus` must be INFEASIBLE

- `DualStatus` must be INFEASIBILITY_CERTIFICATE

- `ConstraintDual` must be the corresponding value of d

- `DualObjectiveValue` must be the value $-\sum_{i=1}^{m} b_i^\top d_i$ for minimization problems and $\sum_{i=1}^{m} b_i^\top d_i$ for maximization problems.

> **Note**
>
> The choice of whether to scale the ray d to have magnitude 1 is left to the solver.

Infeasibility certificates of variable bounds

Many linear solvers (for example, Gurobi) do not provide explicit access to the primal infeasibility certificate of a variable bound. However, given a set of linear constraints:

$$l_A \leq Ax \leq u_A \tag{40.55}$$

$$l_x \leq x \leq u_x, \tag{40.56}$$

the primal certificate of the variable bounds can be computed using the primal certificate associated with the affine constraints, d. (Note that d will have one element for each row of the A matrix, and that some or all of the elements in the vectors l_A and u_A may be $\pm\infty$. If both l_A and u_A are finite for some row, the corresponding element in 'd must be 0.)

Given d, compute $\bar{d} = d^\top A$. If the bound is finite, a certificate for the lower variable bound of x_i is $\max\{\bar{d}_i, 0\}$, and a certificate for the upper variable bound is $\min\{\bar{d}_i, 0\}$.

40.3   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`, for example, `AbstractScalarSet`, `AbstractVectorSet`.

- Vector-valued conic sets should end with `Cone`, for example, `NormInfinityCone`, `SecondOrderCone`.

- Vector-valued Cartesian products should be plural and not end in Cone, for example, `Nonnegatives`, not `NonnegativeCone`.

- Matrix-valued conic sets should provide two representations: `ConeSquare` and `ConeTriangle`, for example, `RootDetConeTriangle` and `RootDetConeSquare`. See `Matrix cones` for more details.

- Scalar sets should be singular, not plural, for example, `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 41

API Reference

41.1 Standard form

Functions

MathOptInterface.AbstractFunction – Type.

```
AbstractFunction
```

Abstract supertype for function objects.

Required methods

All functions must implement:

- Base.copy
- Base.isapprox
- constant

Abstract subtypes of AbstractFunction may require additional methods to be implemented.

MathOptInterface.output_dimension – Function.

```
output_dimension(f::AbstractFunction)
```

Return 1 if f is an AbstractScalarFunction, or the number of output components if f is an AbstractVectorFunction.

MathOptInterface.constant – Function.

```
constant(f::AbstractFunction[, ::Type{T}]) where {T}
```

Returns the constant term of a scalar-valued function, or the constant vector of a vector-valued function.

If f is untyped and T is provided, returns zero(T).

```
constant(set::Union{EqualTo,GreaterThan,LessThan,Parameter})
```

Returns the constant term of the set set.

Example

```
julia> import MathOptInterface as MOI

julia> MOI.constant(MOI.GreaterThan(1.0))
1.0

julia> MOI.constant(MOI.LessThan(2.5))
2.5

julia> MOI.constant(MOI.EqualTo(3))
3

julia> MOI.constant(MOI.Parameter(4.5))
4.5
```

Scalar functions

MathOptInterface.AbstractScalarFunction – Type.

```
abstract type AbstractScalarFunction <: AbstractFunction
```

Abstract supertype for scalar-valued AbstractFunctions.

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.ScalarAffineTerm – Type.

```
ScalarAffineTerm{T}(coefficient::T, variable::VariableIndex) where {T}
```

Represents the scalar-valued term coefficient * variable.

Example

```
julia> import MathOptInterface as MOI

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

julia> MOI.ScalarAffineTerm(2.0, x)
MathOptInterface.ScalarAffineTerm{Float64}(2.0, MOI.VariableIndex(1))
```

MathOptInterface.ScalarAffineFunction – Type.

```
ScalarAffineFunction{T}(terms::ScalarAffineTerm{T}, constant::T) where {T}
```

Represents the scalar-valued affine function $a^\top x + b$, where:

- $a^\top x$ is represented by the vector of ScalarAffineTerms
- b is a scalar constant::T

Duplicates

Duplicate variable indices in terms are accepted, and the corresponding coefficients are summed together.

Example

```julia
julia> import MathOptInterface as MOI

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

julia> terms = [MOI.ScalarAffineTerm(2.0, x), MOI.ScalarAffineTerm(3.0, x)]
2-element Vector{MathOptInterface.ScalarAffineTerm{Float64}}:
 MathOptInterface.ScalarAffineTerm{Float64}(2.0, MOI.VariableIndex(1))
 MathOptInterface.ScalarAffineTerm{Float64}(3.0, MOI.VariableIndex(1))

julia> f = MOI.ScalarAffineFunction(terms, 4.0)
4.0 + 2.0 MOI.VariableIndex(1) + 3.0 MOI.VariableIndex(1)
```

MathOptInterface.ScalarQuadraticTerm – Type.

```
ScalarQuadraticTerm{T}(
    coefficient::T,
    variable_1::VariableIndex,
    variable_2::VariableIndex,
) where {T}
```

Represents the scalar-valued term cx_ix_j where c is coefficient, x_i is variable_1 and x_j is variable_2.

Example

```julia
julia> import MathOptInterface as MOI

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

julia> MOI.ScalarQuadraticTerm(2.0, x, x)
MathOptInterface.ScalarQuadraticTerm{Float64}(2.0, MOI.VariableIndex(1), MOI.VariableIndex(1))
```

MathOptInterface.ScalarQuadraticFunction – Type.

```
ScalarQuadraticFunction{T}(
    quadratic_terms::Vector{ScalarQuadraticTerm{T}},
    affine_terms::Vector{ScalarAffineTerm{T}},
    constant::T,
) wher {T}
```

The scalar-valued quadratic function $\frac{1}{2}x^\top Qx + a^\top x + b$, where:

- Q is the symmetric matrix given by the vector of ScalarQuadraticTerms
- $a^\top x$ is a sparse vector given by the vector of ScalarAffineTerms
- b is the scalar constant::T.

Duplicates

Duplicate indices in quadratic_terms or affine_terms are accepted, and the corresponding coefficients are summed together.

In quadratic_terms, "mirrored" indices, (q, r) and (r, q) where r and q are VariableIndexes, are considered duplicates; only one needs to be specified.

The 0.5 factor

Coupled with the interpretation of mirrored indices, the 0.5 factor in front of the Q matrix is a common source of bugs.

As a rule, to represent $a * x^2 + b * x * y$:

- The coefficient a in front of squared variables (diagonal elements in Q) must be doubled when creating a ScalarQuadraticTerm
- The coefficient b in front of off-diagonal elements in Q should be left as b, be cause the mirrored index $b * y * x$ will be implicitly added.

Example

To represent the function $f(x, y) = 2 * x^2 + 3 * x * y + 4 * x + 5$, do:

```
julia> import MathOptInterface as MOI

julia> x = MOI.VariableIndex(1);

julia> y = MOI.VariableIndex(2);

julia> constant = 5.0;

julia> affine_terms = [MOI.ScalarAffineTerm(4.0, x)];

julia> quadratic_terms = [
           MOI.ScalarQuadraticTerm(4.0, x, x),  # Note the changed coefficient
           MOI.ScalarQuadraticTerm(3.0, x, y),
       ]
2-element Vector{MathOptInterface.ScalarQuadraticTerm{Float64}}:
 MathOptInterface.ScalarQuadraticTerm{Float64}(4.0, MOI.VariableIndex(1), MOI.VariableIndex(1))
 MathOptInterface.ScalarQuadraticTerm{Float64}(3.0, MOI.VariableIndex(1), MOI.VariableIndex(2))
```

```julia
julia> f = MOI.ScalarQuadraticFunction(quadratic_terms, affine_terms, constant)
5.0 + 4.0 MOI.VariableIndex(1) + 2.0 MOI.VariableIndex(1)² + 3.0
↪  MOI.VariableIndex(1)*MOI.VariableIndex(2)
```

Vector functions

MathOptInterface.AbstractVectorFunction – Type.

```julia
abstract type AbstractVectorFunction <: AbstractFunction
```

Abstract supertype for vector-valued AbstractFunctions.

Required methods

All subtypes of AbstractVectorFunction must implement:

- output_dimension

MathOptInterface.VectorOfVariables – Type.

```julia
VectorOfVariables(variables::Vector{VariableIndex}) <: AbstractVectorFunction
```

The vector-valued function f(x) = variables, where variables is a subset of VariableIndexes in the model.

The list of variables may contain duplicates.

Example

```julia
julia> import MathOptInterface as MOI

julia> x = MOI.VariableIndex.(1:2)
2-element Vector{MathOptInterface.VariableIndex}:
 MOI.VariableIndex(1)
 MOI.VariableIndex(2)

julia> f = MOI.VectorOfVariables([x[1], x[2], x[1]])
⌈                    ⌉
|MOI.VariableIndex(1)|
|MOI.VariableIndex(2)|
|MOI.VariableIndex(1)|
⌊                    ⌋

julia> MOI.output_dimension(f)
3
```

MathOptInterface.VectorAffineTerm – Type.

```
VectorAffineTerm{T}(
    output_index::Int64,
    scalar_term::ScalarAffineTerm{T},
) where {T}
```

A VectorAffineTerm is a scalar_term that appears in the output_index row of the vector-valued VectorAffineFunction or VectorQuadraticFunction.

Example

```
julia> import MathOptInterface as MOI

julia> x = MOI.VariableIndex(1);

julia> MOI.VectorAffineTerm(Int64(2), MOI.ScalarAffineTerm(3.0, x))
MathOptInterface.VectorAffineTerm{Float64}(2, MathOptInterface.ScalarAffineTerm{Float64}(3.0,
↪  MOI.VariableIndex(1)))
```

MathOptInterface.VectorAffineFunction – Type.

```
VectorAffineFunction{T}(
    terms::Vector{VectorAffineTerm{T}},
    constants::Vector{T},
) where {T}
```

The vector-valued affine function $Ax + b$, where:

- Ax is the sparse matrix given by the vector of VectorAffineTerms
- b is the vector constants

Duplicates

Duplicate indices in the A are accepted, and the corresponding coefficients are summed together.

Example

```
julia> import MathOptInterface as MOI

julia> x = MOI.VariableIndex(1);

julia> terms = [
           MOI.VectorAffineTerm(Int64(1), MOI.ScalarAffineTerm(2.0, x)),
           MOI.VectorAffineTerm(Int64(2), MOI.ScalarAffineTerm(3.0, x)),
       ];

julia> f = MOI.VectorAffineFunction(terms, [4.0, 5.0])
⌈                          ⌉
│4.0 + 2.0 MOI.VariableIndex(1)│
│5.0 + 3.0 MOI.VariableIndex(1)│
⌊                          ⌋
```

```
julia> MOI.output_dimension(f)
2
```

MathOptInterface.VectorQuadraticTerm – Type.

```
VectorQuadraticTerm{T}(
    output_index::Int64,
    scalar_term::ScalarQuadraticTerm{T},
) where {T}
```

A VectorQuadraticTerm is a ScalarQuadraticTerm scalar_term that appears in the output_index row of the vector-valued VectorQuadraticFunction.

Example

```
julia> import MathOptInterface as MOI

julia> x = MOI.VariableIndex(1);

julia> MOI.VectorQuadraticTerm(Int64(2), MOI.ScalarQuadraticTerm(3.0, x, x))
MathOptInterface.VectorQuadraticTerm{Float64}(2,
↪   MathOptInterface.ScalarQuadraticTerm{Float64}(3.0, MOI.VariableIndex(1),
↪   MOI.VariableIndex(1)))
```

MathOptInterface.VectorQuadraticFunction – Type.

```
VectorQuadraticFunction{T}(
    quadratic_terms::Vector{VectorQuadraticTerm{T}},
    affine_terms::Vector{VectorAffineTerm{T}},
    constants::Vector{T},
) where {T}
```

The vector-valued quadratic function with ith component ("output index") defined as $\frac{1}{2} x^\top Q_i x + a_i^\top x + b_i$, where:

- $\frac{1}{2} x^\top Q_i x$ is the symmetric matrix given by the VectorQuadraticTerm elements in quadratic_terms with output_index == i
- $a_i^\top x$ is the sparse vector given by the VectorAffineTerm elements in affine_terms with output_index == i
- b_i is a scalar given by constants[i]

Duplicates

Duplicate indices in quadratic_terms and affine_terms with the same output_index are handled in the same manner as duplicates in ScalarQuadraticFunction.

Example

```julia
julia> import MathOptInterface as MOI

julia> x = MOI.VariableIndex(1);

julia> y = MOI.VariableIndex(2);

julia> constants = [4.0, 5.0];

julia> affine_terms = [
           MOI.VectorAffineTerm(Int64(1), MOI.ScalarAffineTerm(2.0, x)),
           MOI.VectorAffineTerm(Int64(2), MOI.ScalarAffineTerm(3.0, x)),
       ];

julia> quad_terms = [
        MOI.VectorQuadraticTerm(Int64(1), MOI.ScalarQuadraticTerm(2.0, x, x)),
        MOI.VectorQuadraticTerm(Int64(2), MOI.ScalarQuadraticTerm(3.0, x, y)),
          ];

julia> f = MOI.VectorQuadraticFunction(quad_terms, affine_terms, constants)
⌈                                                                          ⌉
│4.0 + 2.0 MOI.VariableIndex(1) + 1.0 MOI.VariableIndex(1)²                 │
│5.0 + 3.0 MOI.VariableIndex(1) + 3.0 MOI.VariableIndex(1)*MOI.VariableIndex(2)│
⌊                                                                          ⌋

julia> MOI.output_dimension(f)
2
```

Sets

MathOptInterface.AbstractSet – Type.

```
AbstractSet
```

Abstract supertype for set objects used to encode constraints.

Required methods

For sets of type S with isbitstype(S) == false, you must implement:

- Base.copy(set::S)
- Base.:(==)(x::S, y::S)

Subtypes of AbstractSet such as AbstractScalarSet and AbstractVectorSet may prescribe additional required methods.

Optional methods

You may optionally implement:

- dual_set
- dual_set_type

Note for developers

When creating a new set, the set struct must not contain any VariableIndex or ConstraintIndex objects.

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.

Required methods

All AbstractVectorSets of type S must implement:

- dimension, unless the dimension is stored in the set.dimension field
- Utilities.set_dot, unless the dot product between two vectors in the set is equivalent to LinearAlgebra.dot.

Utilities

MathOptInterface.dimension – Function.

```
dimension(set::AbstractSet)
```

Return the output_dimension that an AbstractFunction should have to be used with the set set.

Example

```julia
julia> import MathOptInterface as MOI

julia> MOI.dimension(MOI.Reals(4))
4

julia> MOI.dimension(MOI.LessThan(3.0))
1

julia> MOI.dimension(MOI.PositiveSemidefiniteConeTriangle(2))
3
```

MathOptInterface.dual_set – Function.

```
dual_set(set::AbstractSet)
```

Return the dual set of set, 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.

Example

```
julia> import MathOptInterface as MOI

julia> MOI.dual_set(MOI.Reals(4))
MathOptInterface.Zeros(4)

julia> MOI.dual_set(MOI.SecondOrderCone(5))
MathOptInterface.SecondOrderCone(5)

julia> MOI.dual_set(MOI.ExponentialCone())
MathOptInterface.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.

Example

```
julia> import MathOptInterface as MOI

julia> MOI.dual_set_type(MOI.Reals)
MathOptInterface.Zeros

julia> MOI.dual_set_type(MOI.SecondOrderCone)
MathOptInterface.SecondOrderCone

julia> MOI.dual_set_type(MOI.ExponentialCone)
MathOptInterface.DualExponentialCone
```

MathOptInterface.constant – Method.

```
constant(set::Union{EqualTo,GreaterThan,LessThan,Parameter})
```

Returns the constant term of the set set.

Example

```
julia> import MathOptInterface as MOI

julia> MOI.constant(MOI.GreaterThan(1.0))
```

```
1.0

julia> MOI.constant(MOI.LessThan(2.5))
2.5

julia> MOI.constant(MOI.EqualTo(3))
3

julia> MOI.constant(MOI.Parameter(4.5))
4.5
```

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::Int)
```

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 $[lower, \infty) \subseteq \mathbb{R}$.

Example

```
julia> import MathOptInterface as MOI

julia> model = MOI.Utilities.Model{Float64}()
MOIU.Model{Float64}

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

```julia
julia> MOI.add_constraint(model, x, MOI.GreaterThan(0.0))
MathOptInterface.ConstraintIndex{MathOptInterface.VariableIndex,
↪   MathOptInterface.GreaterThan{Float64}}(1)
```

MathOptInterface.LessThan – Type.

```julia
LessThan{T<:Real}(upper::T)
```

The set $(-\infty, upper] \subseteq \mathbb{R}$.

Example

```julia
julia> import MathOptInterface as MOI

julia> model = MOI.Utilities.Model{Float64}()
MOIU.Model{Float64}

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

julia> MOI.add_constraint(model, x, MOI.LessThan(2.0))
MathOptInterface.ConstraintIndex{MathOptInterface.VariableIndex,
↪   MathOptInterface.LessThan{Float64}}(1)
```

MathOptInterface.EqualTo – Type.

```julia
EqualTo{T<:Number}(value::T)
```

The set containing the single point $\{value\} \subseteq \mathbb{R}$.

Example

```julia
julia> import MathOptInterface as MOI

julia> model = MOI.Utilities.Model{Float64}()
MOIU.Model{Float64}

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

julia> MOI.add_constraint(model, x, MOI.EqualTo(2.0))
MathOptInterface.ConstraintIndex{MathOptInterface.VariableIndex,
↪   MathOptInterface.EqualTo{Float64}}(1)
```

MathOptInterface.Interval – Type.

```julia
Interval{T<:Real}(lower::T, upper::T)
```

The interval $[lower, upper] \subseteq \mathbb{R} \cup \{-\infty, +\infty\}$.

If lower or upper is -Inf or Inf, respectively, the set is interpreted as a one-sided interval.

Example

```julia
julia> import MathOptInterface as MOI

julia> model = MOI.Utilities.Model{Float64}()
MOIU.Model{Float64}

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

julia> MOI.add_constraint(model, x, MOI.Interval(1.0, 2.0))
MathOptInterface.ConstraintIndex{MathOptInterface.VariableIndex,
↪  MathOptInterface.Interval{Float64}}(1)
```

MathOptInterface.Integer – Type.

```julia
Integer()
```

The set of integers, $\mathbb{Z}$.

Example

```julia
julia> import MathOptInterface as MOI

julia> model = MOI.Utilities.Model{Float64}()
MOIU.Model{Float64}

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

julia> MOI.add_constraint(model, x, MOI.Integer())
MathOptInterface.ConstraintIndex{MathOptInterface.VariableIndex, MathOptInterface.Integer}(1)
```

MathOptInterface.ZeroOne – Type.

```julia
ZeroOne()
```

The set $\{0, 1\}$.

Variables belonging to the ZeroOne set are also known as "binary" variables.

Example

```julia
julia> import MathOptInterface as MOI

julia> model = MOI.Utilities.Model{Float64}()
MOIU.Model{Float64}
```

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

julia> MOI.add_constraint(model, x, MOI.ZeroOne())
MathOptInterface.ConstraintIndex{MathOptInterface.VariableIndex, MathOptInterface.ZeroOne}(1)
```

MathOptInterface.Semicontinuous – Type.

```
Semicontinuous{T<:Real}(lower::T, upper::T)
```

The set $\{0\} \cup [lower, upper]$.

Example

```
julia> import MathOptInterface as MOI

julia> model = MOI.Utilities.Model{Float64}()
MOIU.Model{Float64}

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

julia> MOI.add_constraint(model, x, MOI.Semicontinuous(2.0, 3.0))
MathOptInterface.ConstraintIndex{MathOptInterface.VariableIndex,
↪   MathOptInterface.Semicontinuous{Float64}}(1)
```

MathOptInterface.Semiinteger – Type.

```
Semiinteger{T<:Real}(lower::T, upper::T)
```

The set $\{0\} \cup \{lower, lower + 1, \ldots, upper - 1, upper\}$.

Note that if lower and upper are not equivalent to an integer, then the solver may throw an error, or it may round up lower and round down upper to the nearest integers.

Example

```
julia> import MathOptInterface as MOI

julia> model = MOI.Utilities.Model{Float64}()
MOIU.Model{Float64}

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

julia> MOI.add_constraint(model, x, MOI.Semiinteger(2.0, 3.0))
MathOptInterface.ConstraintIndex{MathOptInterface.VariableIndex,
↪   MathOptInterface.Semiinteger{Float64}}(1)
```

MathOptInterface.Parameter – Type.

```
Parameter{T<:Number}(value::T)
```

The set containing the single point $\{value\} \subseteq \mathbb{R}$.

The Parameter set is conceptually similar to the EqualTo set, except that a variable constrained to the Parameter set cannot have other constraints added to it, and the Parameter set can never be deleted. Thus, solvers are free to treat the variable as a constant, and they need not add it as a decision variable to the model.

Because of this behavior, you must add parameters using add_constrained_variable, and solvers should declare supports_add_constrained_variable and not supports_constraint for the Parameter set.

Example

```
julia> import MathOptInterface as MOI

julia> model = MOI.Utilities.Model{Float64}()
MOIU.Model{Float64}

julia> p, ci = MOI.add_constrained_variable(model, MOI.Parameter(2.5))
(MOI.VariableIndex(1), MathOptInterface.ConstraintIndex{MathOptInterface.VariableIndex,
↪  MathOptInterface.Parameter{Float64}}(1))

julia> MOI.set(model, MOI.ConstraintSet(), ci, MOI.Parameter(3.0))

julia> MOI.get(model, MOI.ConstraintSet(), ci)
MathOptInterface.Parameter{Float64}(3.0)
```

Vector sets

List of recognized vector sets.

MathOptInterface.Reals – Type.

```
Reals(dimension::Int)
```

The set $\mathbb{R}^{dimension}$ (containing all points) of non-negative dimension dimension.

Example

```
julia> import MathOptInterface as MOI

julia> model = MOI.Utilities.Model{Float64}()
MOIU.Model{Float64}

julia> x = MOI.add_variables(model, 3);

julia> MOI.add_constraint(model, MOI.VectorOfVariables(x), MOI.Reals(3))
MathOptInterface.ConstraintIndex{MathOptInterface.VectorOfVariables, MathOptInterface.Reals}(1)
```

MathOptInterface.Zeros – Type.

```
Zeros(dimension::Int)
```

The set $\{0\}^{dimension}$ (containing only the origin) of non-negative dimension dimension.

Example

```
julia> import MathOptInterface as MOI

julia> model = MOI.Utilities.Model{Float64}()
MOIU.Model{Float64}

julia> x = MOI.add_variables(model, 3);

julia> MOI.add_constraint(model, MOI.VectorOfVariables(x), MOI.Zeros(3))
MathOptInterface.ConstraintIndex{MathOptInterface.VectorOfVariables, MathOptInterface.Zeros}(1)
```

MathOptInterface.Nonnegatives – Type.

```
Nonnegatives(dimension::Int)
```

The nonnegative orthant $\{x \in \mathbb{R}^{dimension} : x \geq 0\}$ of non-negative dimension dimension.

Example

```
julia> import MathOptInterface as MOI

julia> model = MOI.Utilities.Model{Float64}()
MOIU.Model{Float64}

julia> x = MOI.add_variables(model, 3);

julia> MOI.add_constraint(model, MOI.VectorOfVariables(x), MOI.Nonnegatives(3))
MathOptInterface.ConstraintIndex{MathOptInterface.VectorOfVariables,
↪   MathOptInterface.Nonnegatives}(1)
```

MathOptInterface.Nonpositives – Type.

```
Nonpositives(dimension::Int)
```

The nonpositive orthant $\{x \in \mathbb{R}^{dimension} : x \leq 0\}$ of non-negative dimension dimension.

Example

```
julia> import MathOptInterface as MOI

julia> model = MOI.Utilities.Model{Float64}()
```

```
MOIU.Model{Float64}

julia> x = MOI.add_variables(model, 3);

julia> MOI.add_constraint(model, MOI.VectorOfVariables(x), MOI.Nonpositives(3))
MathOptInterface.ConstraintIndex{MathOptInterface.VectorOfVariables,
↪   MathOptInterface.Nonpositives}(1)
```

MathOptInterface.NormInfinityCone – Type.

```
NormInfinityCone(dimension::Int)
```

The ℓ_∞-norm cone $\{(t, x) \in \mathbb{R}^{dimension} : t \geq \|x\|_\infty = \max_i |x_i|\}$ of dimension `dimension`.

The `dimension` must be at least 1.

Example

```
julia> import MathOptInterface as MOI

julia> model = MOI.Utilities.Model{Float64}()
MOIU.Model{Float64}

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

julia> x = MOI.add_variables(model, 3);

julia> MOI.add_constraint(model, MOI.VectorOfVariables([t; x]), MOI.NormInfinityCone(4))
MathOptInterface.ConstraintIndex{MathOptInterface.VectorOfVariables,
↪   MathOptInterface.NormInfinityCone}(1)
```

MathOptInterface.NormOneCone – Type.

```
NormOneCone(dimension::Int)
```

The ℓ_1-norm cone $\{(t, x) \in \mathbb{R}^{dimension} : t \geq \|x\|_1 = \sum_i |x_i|\}$ of dimension `dimension`.

The `dimension` must be at least 1.

Example

```
julia> import MathOptInterface as MOI

julia> model = MOI.Utilities.Model{Float64}()
MOIU.Model{Float64}

julia> t = MOI.add_variable(model)
MOI.VariableIndex(1)
```

```
julia> x = MOI.add_variables(model, 3);

julia> MOI.add_constraint(model, MOI.VectorOfVariables([t; x]), MOI.NormOneCone(4))
MathOptInterface.ConstraintIndex{MathOptInterface.VectorOfVariables,
↪   MathOptInterface.NormOneCone}(1)
```

MathOptInterface.NormCone – Type.

```
NormCone(p::Float64, dimension::Int)
```

The ℓ_p-norm cone $\{(t, x) \in \mathbb{R}^{dimension} : t \geq \left(\sum_i |x_i|^p \right)^{\frac{1}{p}} \}$ of dimension dimension.

The dimension must be at least 1.

Example

```
julia> import MathOptInterface as MOI

julia> model = MOI.Utilities.Model{Float64}()
MOIU.Model{Float64}

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

julia> x = MOI.add_variables(model, 3);

julia> MOI.add_constraint(model, MOI.VectorOfVariables([t; x]), MOI.NormCone(3, 4))
MathOptInterface.ConstraintIndex{MathOptInterface.VectorOfVariables,
↪   MathOptInterface.NormCone}(1)
```

MathOptInterface.SecondOrderCone – Type.

```
SecondOrderCone(dimension::Int)
```

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.

The dimension must be at least 1.

Example

```
julia> import MathOptInterface as MOI

julia> model = MOI.Utilities.Model{Float64}()
MOIU.Model{Float64}

julia> t = MOI.add_variable(model)
MOI.VariableIndex(1)
```

```
julia> x = MOI.add_variables(model, 3);

julia> MOI.add_constraint(model, MOI.VectorOfVariables([t; x]), MOI.SecondOrderCone(4))
MathOptInterface.ConstraintIndex{MathOptInterface.VectorOfVariables,
↪   MathOptInterface.SecondOrderCone}(1)
```

MathOptInterface.RotatedSecondOrderCone – Type.

```
RotatedSecondOrderCone(dimension::Int)
```

The rotated second-order cone $\{(t, u, x) \in \mathbb{R}^{dimension} : 2tu \geq \|x\|_2^2, t, u \geq 0\}$ of dimension `dimension`.

The `dimension` must be at least 2.

Example

```
julia> import MathOptInterface as MOI

julia> model = MOI.Utilities.Model{Float64}()
MOIU.Model{Float64}

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

julia> u = MOI.add_variable(model)
MOI.VariableIndex(2)

julia> x = MOI.add_variables(model, 3);

julia> MOI.add_constraint(
           model,
           MOI.VectorOfVariables([t; u; x]),
           MOI.RotatedSecondOrderCone(5),
       )
MathOptInterface.ConstraintIndex{MathOptInterface.VectorOfVariables,
↪   MathOptInterface.RotatedSecondOrderCone}(1)
```

MathOptInterface.GeometricMeanCone – Type.

```
GeometricMeanCone(dimension::Int)
```

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`.

Example

```julia
julia> import MathOptInterface as MOI

julia> model = MOI.Utilities.Model{Float64}()
MOIU.Model{Float64}

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

julia> x = MOI.add_variables(model, 3);

julia> MOI.add_constraint(
           model,
           MOI.VectorOfVariables([t; x]),
           MOI.GeometricMeanCone(4),
       )
MathOptInterface.ConstraintIndex{MathOptInterface.VectorOfVariables,
↪   MathOptInterface.GeometricMeanCone}(1)
```

MathOptInterface.ExponentialCone – Type.

```
ExponentialCone()
```

The 3-dimensional exponential cone $\{(x, y, z) \in \mathbb{R}^3 : y \exp(x/y) \leq z, y > 0\}$.

Example

```julia
julia> import MathOptInterface as MOI

julia> model = MOI.Utilities.Model{Float64}()
MOIU.Model{Float64}

julia> x = MOI.add_variables(model, 3);

julia> MOI.add_constraint(model, MOI.VectorOfVariables(x), MOI.ExponentialCone())
MathOptInterface.ConstraintIndex{MathOptInterface.VectorOfVariables,
↪   MathOptInterface.ExponentialCone}(1)
```

MathOptInterface.DualExponentialCone – Type.

```
DualExponentialCone()
```

The 3-dimensional dual exponential cone $\{(u, v, w) \in \mathbb{R}^3 : -u \exp(v/u) \leq \exp(1)w, u < 0\}$.

Example

```julia
julia> import MathOptInterface as MOI

julia> model = MOI.Utilities.Model{Float64}()
MOIU.Model{Float64}
```

```julia
julia> x = MOI.add_variables(model, 3);

julia> MOI.add_constraint(model, MOI.VectorOfVariables(x), MOI.DualExponentialCone())
MathOptInterface.ConstraintIndex{MathOptInterface.VectorOfVariables,
↪  MathOptInterface.DualExponentialCone}(1)
```

MathOptInterface.PowerCone – Type.

```julia
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.

Example

```julia
julia> import MathOptInterface as MOI

julia> model = MOI.Utilities.Model{Float64}()
MOIU.Model{Float64}

julia> x = MOI.add_variables(model, 3);

julia> MOI.add_constraint(model, MOI.VectorOfVariables(x), MOI.PowerCone(0.5))
MathOptInterface.ConstraintIndex{MathOptInterface.VectorOfVariables,
↪  MathOptInterface.PowerCone{Float64}}(1)
```

MathOptInterface.DualPowerCone – Type.

```julia
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.

Example

```julia
julia> import MathOptInterface as MOI

julia> model = MOI.Utilities.Model{Float64}()
MOIU.Model{Float64}

julia> x = MOI.add_variables(model, 3);

julia> MOI.add_constraint(model, MOI.VectorOfVariables(x), MOI.DualPowerCone(0.5))
MathOptInterface.ConstraintIndex{MathOptInterface.VectorOfVariables,
↪  MathOptInterface.DualPowerCone{Float64}}(1)
```

MathOptInterface.RelativeEntropyCone – Type.

```
RelativeEntropyCone(dimension::Int)
```

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 $= 2n + 1$.

Example

```
julia> import MathOptInterface as MOI

julia> model = MOI.Utilities.Model{Float64}()
MOIU.Model{Float64}

julia> u = MOI.add_variable(model);

julia> v = MOI.add_variables(model, 3);

julia> w = MOI.add_variables(model, 3);

julia> MOI.add_constraint(
           model,
           MOI.VectorOfVariables([u; v; w]),
           MOI.RelativeEntropyCone(7),
       )
MathOptInterface.ConstraintIndex{MathOptInterface.VectorOfVariables,
↪   MathOptInterface.RelativeEntropyCone}(1)
```

MathOptInterface.NormSpectralCone – Type.

```
NormSpectralCone(row_dim::Int, column_dim::Int)
```

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 non-negative 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.

Example

```
julia> import MathOptInterface as MOI

julia> model = MOI.Utilities.Model{Float64}()
MOIU.Model{Float64}

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

julia> X = reshape(MOI.add_variables(model, 6), 2, 3)
```

```
2×3 Matrix{MathOptInterface.VariableIndex}:
 MOI.VariableIndex(2)  MOI.VariableIndex(4)  MOI.VariableIndex(6)
 MOI.VariableIndex(3)  MOI.VariableIndex(5)  MOI.VariableIndex(7)

julia> MOI.add_constraint(
           model,
           MOI.VectorOfVariables([t; vec(X)]),
           MOI.NormSpectralCone(2, 3),
       )
MathOptInterface.ConstraintIndex{MathOptInterface.VectorOfVariables,
↪  MathOptInterface.NormSpectralCone}(1)
```

MathOptInterface.NormNuclearCone – Type.

```
NormNuclearCone(row_dim::Int, column_dim::Int)
```

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 non-negative 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.

Example

```
julia> import MathOptInterface as MOI

julia> model = MOI.Utilities.Model{Float64}()
MOIU.Model{Float64}

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

julia> X = reshape(MOI.add_variables(model, 6), 2, 3)
2×3 Matrix{MathOptInterface.VariableIndex}:
 MOI.VariableIndex(2)  MOI.VariableIndex(4)  MOI.VariableIndex(6)
 MOI.VariableIndex(3)  MOI.VariableIndex(5)  MOI.VariableIndex(7)

julia> MOI.add_constraint(
           model,
           MOI.VectorOfVariables([t; vec(X)]),
           MOI.NormNuclearCone(2, 3),
       )
MathOptInterface.ConstraintIndex{MathOptInterface.VectorOfVariables,
↪  MathOptInterface.NormNuclearCone}(1)
```

MathOptInterface.SOS1 – Type.

```
SOS1{T<:Real}(weights::Vector{T})
```

The set corresponding to the Special Ordered Set (SOS) constraint of Type I.

Of the variables in the set, at most one can be nonzero.

The `weights` induce an ordering of the variables such that the kth element in the set corresponds to the kth weight in `weights`. Solvers may use these weights to improve the efficiency of the solution process, but the ordering does not change the set of feasible solutions.

Example

```julia
julia> import MathOptInterface as MOI

julia> model = MOI.Utilities.Model{Float64}()
MOIU.Model{Float64}

julia> x = MOI.add_variables(model, 3);

julia> MOI.add_constraint(
           model,
           MOI.VectorOfVariables(x),
           MOI.SOS1([1.0, 3.0, 2.5]),
       )
MathOptInterface.ConstraintIndex{MathOptInterface.VectorOfVariables,
↪   MathOptInterface.SOS1{Float64}}(1)
```

MathOptInterface.SOS2 – Type.

```julia
SOS2{T<:Real}(weights::Vector{T})
```

The set corresponding to the Special Ordered Set (SOS) constraint of Type II.

The `weights` induce an ordering of the variables such that the kth element in the set corresponds to the kth weight in `weights`. Therefore, the weights must be unique.

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.

Example

```julia
julia> import MathOptInterface as MOI

julia> model = MOI.Utilities.Model{Float64}()
MOIU.Model{Float64}

julia> x = MOI.add_variables(model, 3);

julia> MOI.add_constraint(
           model,
           MOI.VectorOfVariables(x),
           MOI.SOS2([1.0, 3.0, 2.5]),
       )
MathOptInterface.ConstraintIndex{MathOptInterface.VectorOfVariables,
↪   MathOptInterface.SOS2{Float64}}(1)
```

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

```julia
julia> import MathOptInterface as MOI

julia> model = MOI.Utilities.Model{Float64}()
MOIU.Model{Float64}

julia> x = MOI.add_variables(model, 2)
2-element Vector{MathOptInterface.VariableIndex}:
 MOI.VariableIndex(1)
 MOI.VariableIndex(2)

julia> y, _ = MOI.add_constrained_variable(model, MOI.ZeroOne())
(MOI.VariableIndex(3), MathOptInterface.ConstraintIndex{MathOptInterface.VariableIndex,
↪   MathOptInterface.ZeroOne}(3))

julia> f = MOI.VectorAffineFunction(
           [
               MOI.VectorAffineTerm(1, MOI.ScalarAffineTerm(1.0, y)),
               MOI.VectorAffineTerm(2, MOI.ScalarAffineTerm(1.0, x[1])),
               MOI.VectorAffineTerm(2, MOI.ScalarAffineTerm(1.0, x[2])),
           ],
           [0.0, 0.0],
       )
┌                                                      ┐
│0.0 + 1.0 MOI.VariableIndex(3)                        │
│0.0 + 1.0 MOI.VariableIndex(1) + 1.0 MOI.VariableIndex(2)│
└                                                      ┘

julia> s = MOI.Indicator{MOI.ACTIVATE_ON_ONE}(MOI.LessThan(9.0))
MathOptInterface.Indicator{MathOptInterface.ACTIVATE_ON_ONE,
↪   MathOptInterface.LessThan{Float64}}(MathOptInterface.LessThan{Float64}(9.0))

julia> MOI.add_constraint(model, f, s)
MathOptInterface.ConstraintIndex{MathOptInterface.VectorAffineFunction{Float64},
↪   MathOptInterface.Indicator{MathOptInterface.ACTIVATE_ON_ONE,
↪   MathOptInterface.LessThan{Float64}}}(1)
```

`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.

Example

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

```julia
julia> import MathOptInterface as MOI

julia> model = MOI.Utilities.Model{Float64}()
MOIU.Model{Float64}

julia> x, _ = MOI.add_constrained_variable(model, MOI.Interval(-1.0, 1.0));

julia> MOI.add_constraint(
           model,
           MOI.Utilities.vectorize([-4.0 * x - 3.0, x]),
```

```
            MOI.Complements(2),
        )
MathOptInterface.ConstraintIndex{MathOptInterface.VectorAffineFunction{Float64},
↪   MathOptInterface.Complements}(1)
```

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.

```
julia> import MathOptInterface as MOI

julia> model = MOI.Utilities.Model{Float64}()
MOIU.Model{Float64}

julia> x = MOI.add_variables(model, 4);

julia> MOI.add_constraint(model, MOI.VectorOfVariables(x[3:4]), MOI.Nonnegatives(2))
MathOptInterface.ConstraintIndex{MathOptInterface.VectorOfVariables,
↪   MathOptInterface.Nonnegatives}(1)

julia> MOI.add_constraint(
           model,
           MOI.VectorOfVariables(x),
           MOI.Complements(4),
       )
MathOptInterface.ConstraintIndex{MathOptInterface.VectorOfVariables,
↪   MathOptInterface.Complements}(1)
```

MathOptInterface.HyperRectangle – Type.

```
HyperRectangle(lower::Vector{T}, upper::Vector{T}) where {T}
```

The set $\{x \in \bar{\mathbb{R}}^d : x_i \in [lower_i, upper_i] \forall i = 1, \ldots, d\}$.

Example

```
julia> import MathOptInterface as MOI

julia> model = MOI.Utilities.Model{Float64}()
MOIU.Model{Float64}

julia> x = MOI.add_variables(model, 3)
3-element Vector{MathOptInterface.VariableIndex}:
 MOI.VariableIndex(1)
 MOI.VariableIndex(2)
 MOI.VariableIndex(3)
```

```
julia> MOI.add_constraint(
           model,
           MOI.VectorOfVariables(x),
           MOI.HyperRectangle(zeros(3), ones(3)),
       )
MathOptInterface.ConstraintIndex{MathOptInterface.VectorOfVariables,
↪   MathOptInterface.HyperRectangle{Float64}}(1)
```

Constraint programming sets

MathOptInterface.AllDifferent – Type.

```
AllDifferent(dimension::Int)
```

The set $\{x \in \mathbb{Z}^d\}$ such that no two elements in x take the same value and dimension = d.

Also known as

This constraint is called all_different in MiniZinc, and is sometimes also called distinct.

Example

To enforce x[1] != x[2] AND x[1] != x[3] AND x[2] != x[3]:

```
julia> import MathOptInterface as MOI

julia> model = MOI.Utilities.Model{Float64}()
MOIU.Model{Float64}

julia> x = [MOI.add_constrained_variable(model, MOI.Integer())[1] for _ in 1:3]
3-element Vector{MathOptInterface.VariableIndex}:
 MOI.VariableIndex(1)
 MOI.VariableIndex(2)
 MOI.VariableIndex(3)

julia> MOI.add_constraint(model, MOI.VectorOfVariables(x), MOI.AllDifferent(3))
MathOptInterface.ConstraintIndex{MathOptInterface.VectorOfVariables,
↪   MathOptInterface.AllDifferent}(1)
```

MathOptInterface.BinPacking – Type.

```
BinPacking(c::T, w::Vector{T}) where {T}
```

The set $\{x \in \mathbb{Z}^d\}$ where d = length(w), such that each item i in 1:d of weight w[i] is put into bin x[i], and the total weight of each bin does not exceed c.

There are additional assumptions that the capacity, c, and the weights, w, must all be non-negative.

The bin numbers depend on the bounds of x, so they may be something other than the integers 1:d.

Also known as

This constraint is called bin_packing in MiniZinc.

Example

```julia
julia> import MathOptInterface as MOI

julia> model = MOI.Utilities.Model{Float64}()
MOIU.Model{Float64}

julia> bins = MOI.add_variables(model, 5)
5-element Vector{MathOptInterface.VariableIndex}:
 MOI.VariableIndex(1)
 MOI.VariableIndex(2)
 MOI.VariableIndex(3)
 MOI.VariableIndex(4)
 MOI.VariableIndex(5)

julia> weights = Float64[1, 1, 2, 2, 3]
5-element Vector{Float64}:
 1.0
 1.0
 2.0
 2.0
 3.0

julia> MOI.add_constraint.(model, bins, MOI.Integer())
5-element Vector{MathOptInterface.ConstraintIndex{MathOptInterface.VariableIndex,
↪   MathOptInterface.Integer}}:
 MathOptInterface.ConstraintIndex{MathOptInterface.VariableIndex, MathOptInterface.Integer}(1)
 MathOptInterface.ConstraintIndex{MathOptInterface.VariableIndex, MathOptInterface.Integer}(2)
 MathOptInterface.ConstraintIndex{MathOptInterface.VariableIndex, MathOptInterface.Integer}(3)
 MathOptInterface.ConstraintIndex{MathOptInterface.VariableIndex, MathOptInterface.Integer}(4)
 MathOptInterface.ConstraintIndex{MathOptInterface.VariableIndex, MathOptInterface.Integer}(5)

julia> MOI.add_constraint.(model, bins, MOI.Interval(4.0, 6.0))
5-element Vector{MathOptInterface.ConstraintIndex{MathOptInterface.VariableIndex,
↪   MathOptInterface.Interval{Float64}}}:
 MathOptInterface.ConstraintIndex{MathOptInterface.VariableIndex,
↪   MathOptInterface.Interval{Float64}}(1)
 MathOptInterface.ConstraintIndex{MathOptInterface.VariableIndex,
↪   MathOptInterface.Interval{Float64}}(2)
 MathOptInterface.ConstraintIndex{MathOptInterface.VariableIndex,
↪   MathOptInterface.Interval{Float64}}(3)
 MathOptInterface.ConstraintIndex{MathOptInterface.VariableIndex,
↪   MathOptInterface.Interval{Float64}}(4)
 MathOptInterface.ConstraintIndex{MathOptInterface.VariableIndex,
↪   MathOptInterface.Interval{Float64}}(5)

julia> MOI.add_constraint(model, MOI.VectorOfVariables(bins), MOI.BinPacking(3.0, weights))
MathOptInterface.ConstraintIndex{MathOptInterface.VectorOfVariables,
↪   MathOptInterface.BinPacking{Float64}}(1)
```

MathOptInterface.Circuit – Type.

```
Circuit(dimension::Int)
```

The set $\left\{x \in \{1..d\}^d\right\}$ that constraints x to be a circuit, such that $x_i = j$ means that j is the successor of i, and dimension = d.

Graphs with multiple independent circuits, such as [2, 1, 3] and [2, 1, 4, 3], are not valid.

Also known as

This constraint is called circuit in MiniZinc, and it is equivalent to forming a (potentially sub-optimal) tour in the travelling salesperson problem.

Example

```julia
julia> import MathOptInterface as MOI

julia> model = MOI.Utilities.Model{Float64}()
MOIU.Model{Float64}

julia> x = [MOI.add_constrained_variable(model, MOI.Integer())[1] for _ in 1:3]
3-element Vector{MathOptInterface.VariableIndex}:
 MOI.VariableIndex(1)
 MOI.VariableIndex(2)
 MOI.VariableIndex(3)

julia> MOI.add_constraint(model, MOI.VectorOfVariables(x), MOI.Circuit(3))
MathOptInterface.ConstraintIndex{MathOptInterface.VectorOfVariables,
↪   MathOptInterface.Circuit}(1)
```

MathOptInterface.CountAtLeast – Type.

```
CountAtLeast(n::Int, d::Vector{Int}, set::Set{Int})
```

The set $\left\{x \in \mathbb{Z}^{d_1+d_2+\dots d_N}\right\}$, where x is partitioned into N subsets ($\{x_1, \dots, x_{d_1}\}$, $\{x_{d_1+1}, \dots, x_{d_1+d_2}\}$ and so on), and at least n elements of each subset take one of the values in set.

Also known as

This constraint is called at_least in MiniZinc.

Example

To ensure that 3 appears at least once in each of the subsets {a, b} and {b, c}:

```julia
julia> import MathOptInterface as MOI

julia> model = MOI.Utilities.Model{Float64}()
MOIU.Model{Float64}

julia> a, _ = MOI.add_constrained_variable(model, MOI.Integer())
(MOI.VariableIndex(1), MathOptInterface.ConstraintIndex{MathOptInterface.VariableIndex,
↪   MathOptInterface.Integer}(1))
```

```
julia> b, _ = MOI.add_constrained_variable(model, MOI.Integer())
(MOI.VariableIndex(2), MathOptInterface.ConstraintIndex{MathOptInterface.VariableIndex,
↪   MathOptInterface.Integer}(2))

julia> c, _ = MOI.add_constrained_variable(model, MOI.Integer())
(MOI.VariableIndex(3), MathOptInterface.ConstraintIndex{MathOptInterface.VariableIndex,
↪   MathOptInterface.Integer}(3))

julia> x, d, set = [a, b, b, c], [2, 2], [3]
(MathOptInterface.VariableIndex[MOI.VariableIndex(1), MOI.VariableIndex(2),
↪   MOI.VariableIndex(2), MOI.VariableIndex(3)], [2, 2], [3])

julia> MOI.add_constraint(model, MOI.VectorOfVariables(x), MOI.CountAtLeast(1, d, Set(set)))
MathOptInterface.ConstraintIndex{MathOptInterface.VectorOfVariables,
↪   MathOptInterface.CountAtLeast}(1)
```

MathOptInterface.CountBelongs – Type.

```
CountBelongs(dimenson::Int, set::Set{Int})
```

The set $\{(n, x) \in \mathbb{Z}^{1+d}\}$, such that n elements of the vector x take on of the values in set and dimension = 1 + d.

Also known as

This constraint is called among by MiniZinc.

Example

```
julia> import MathOptInterface as MOI

julia> model = MOI.Utilities.Model{Float64}()
MOIU.Model{Float64}

julia> n, _ = MOI.add_constrained_variable(model, MOI.Integer())
(MOI.VariableIndex(1), MathOptInterface.ConstraintIndex{MathOptInterface.VariableIndex,
↪   MathOptInterface.Integer}(1))

julia> x = [MOI.add_constrained_variable(model, MOI.Integer())[1] for _ in 1:3]
3-element Vector{MathOptInterface.VariableIndex}:
 MOI.VariableIndex(2)
 MOI.VariableIndex(3)
 MOI.VariableIndex(4)

julia> set = Set([3, 4, 5])
Set{Int64} with 3 elements:
  5
  4
  3

julia> MOI.add_constraint(model, MOI.VectorOfVariables([n; x]), MOI.CountBelongs(4, set))
MathOptInterface.ConstraintIndex{MathOptInterface.VectorOfVariables,
↪   MathOptInterface.CountBelongs}(1)
```

MathOptInterface.CountDistinct – Type.

```
CountDistinct(dimension::Int)
```

The set $\{(n, x) \in \mathbb{Z}^{1+d}\}$, such that the number of distinct values in x is n and dimension = 1 + d.

Also known as

This constraint is called nvalues in MiniZinc.

Example

To model:

- if n == 1', thenx[1] == x[2] == x[3]'
- if n == 2, then
    - x[1] == x[2] != x[3] or
    - x[1] != x[2] == x[3] or
    - x[1] == x[3] != x[2]
- if n == 3, then x[1] != x[2], x[2] != x[3] and x[3] != x[1]

```
julia> import MathOptInterface as MOI

julia> model = MOI.Utilities.Model{Float64}()
MOIU.Model{Float64}

julia> n, _ = MOI.add_constrained_variable(model, MOI.Integer())
(MOI.VariableIndex(1), MathOptInterface.ConstraintIndex{MathOptInterface.VariableIndex,
↪   MathOptInterface.Integer}(1))

julia> x = [MOI.add_constrained_variable(model, MOI.Integer())[1] for _ in 1:3]
3-element Vector{MathOptInterface.VariableIndex}:
 MOI.VariableIndex(2)
 MOI.VariableIndex(3)
 MOI.VariableIndex(4)

julia> MOI.add_constraint(model, MOI.VectorOfVariables(vcat(n, x)), MOI.CountDistinct(4))
MathOptInterface.ConstraintIndex{MathOptInterface.VectorOfVariables,
↪   MathOptInterface.CountDistinct}(1)
```

Relationship to AllDifferent

When the first element is d, CountDistinct is equivalent to an AllDifferent constraint.

MathOptInterface.CountGreaterThan – Type.

```
CountGreaterThan(dimension::Int)
```

The set $\{(c, y, x) \in \mathbb{Z}^{1+1+d}\}$, such that c is strictly greater than the number of occurances of y in x and dimension = 1 + 1 + d.

Also known as

This constraint is called count_gt in MiniZinc.

Example

```julia
julia> import MathOptInterface as MOI

julia> model = MOI.Utilities.Model{Float64}()
MOIU.Model{Float64}

julia> c, _ = MOI.add_constrained_variable(model, MOI.Integer())
(MOI.VariableIndex(1), MathOptInterface.ConstraintIndex{MathOptInterface.VariableIndex,
↪   MathOptInterface.Integer}(1))

julia> y, _ = MOI.add_constrained_variable(model, MOI.Integer())
(MOI.VariableIndex(2), MathOptInterface.ConstraintIndex{MathOptInterface.VariableIndex,
↪   MathOptInterface.Integer}(2))

julia> x = [MOI.add_constrained_variable(model, MOI.Integer())[1] for _ in 1:3]
3-element Vector{MathOptInterface.VariableIndex}:
 MOI.VariableIndex(3)
 MOI.VariableIndex(4)
 MOI.VariableIndex(5)

julia> MOI.add_constraint(model, MOI.VectorOfVariables([c; y; x]), MOI.CountGreaterThan(5))
MathOptInterface.ConstraintIndex{MathOptInterface.VectorOfVariables,
↪   MathOptInterface.CountGreaterThan}(1)
```

MathOptInterface.Cumulative – Type.

```
Cumulative(dimension::Int)
```

The set $\{(s, d, r, b) \in \mathbb{Z}^{3n+1}\}$, representing the cumulative global constraint, where n == length(s) == length(r) == length(b) and dimension = 3n + 1.

Cumulative requires that a set of tasks given by start times s, durations d, and resource requirements r, never requires more than the global resource bound b at any one time.

Also known as

This constraint is called cumulative in MiniZinc.

Example

```julia
julia> import MathOptInterface as MOI

julia> model = MOI.Utilities.Model{Float64}()
MOIU.Model{Float64}

julia> s = [MOI.add_constrained_variable(model, MOI.Integer())[1] for _ in 1:3]
3-element Vector{MathOptInterface.VariableIndex}:
 MOI.VariableIndex(1)
 MOI.VariableIndex(2)
 MOI.VariableIndex(3)
```

```julia
julia> d = [MOI.add_constrained_variable(model, MOI.Integer())[1] for _ in 1:3]
3-element Vector{MathOptInterface.VariableIndex}:
 MOI.VariableIndex(4)
 MOI.VariableIndex(5)
 MOI.VariableIndex(6)

julia> r = [MOI.add_constrained_variable(model, MOI.Integer())[1] for _ in 1:3]
3-element Vector{MathOptInterface.VariableIndex}:
 MOI.VariableIndex(7)
 MOI.VariableIndex(8)
 MOI.VariableIndex(9)

julia> b, _ = MOI.add_constrained_variable(model, MOI.Integer())
(MOI.VariableIndex(10), MathOptInterface.ConstraintIndex{MathOptInterface.VariableIndex,
↪   MathOptInterface.Integer}(10))

julia> MOI.add_constraint(model, MOI.VectorOfVariables([s; d; r; b]), MOI.Cumulative(10))
MathOptInterface.ConstraintIndex{MathOptInterface.VectorOfVariables,
↪   MathOptInterface.Cumulative}(1)
```

MathOptInterface.Path – Type.

```julia
Path(from::Vector{Int}, to::Vector{Int})
```

Given a graph comprised of a set of nodes 1..N and a set of arcs 1..E represented by an edge from node from[i] to node to[i], Path constrains the set $(s, t, ns, es) \in (1..N) \times (1..E) \times \{0,1\}^N \times \{0,1\}^E$, to form subgraph that is a path from node s to node t, where node n is in the path if ns[n] is 1, and edge e is in the path if es[e] is 1.

The path must be acyclic, and it must traverse all nodes n for which ns[n] is 1, and all edges e for which es[e] is 1.

Also known as

This constraint is called path in MiniZinc.

Example

```julia
julia> import MathOptInterface as MOI

julia> model = MOI.Utilities.Model{Float64}()
MOIU.Model{Float64}

julia> N, E = 4, 5
(4, 5)

julia> from = [1, 1, 2, 2, 3]
5-element Vector{Int64}:
 1
 1
 2
 2
```

```
 3

julia> to = [2, 3, 3, 4, 4]
5-element Vector{Int64}:
 2
 3
 3
 4
 4

julia> s, _ = MOI.add_constrained_variable(model, MOI.Integer())
(MOI.VariableIndex(1), MathOptInterface.ConstraintIndex{MathOptInterface.VariableIndex,
↪  MathOptInterface.Integer}(1))

julia> t, _ = MOI.add_constrained_variable(model, MOI.Integer())
(MOI.VariableIndex(2), MathOptInterface.ConstraintIndex{MathOptInterface.VariableIndex,
↪  MathOptInterface.Integer}(2))

julia> ns = MOI.add_variables(model, N)
4-element Vector{MathOptInterface.VariableIndex}:
 MOI.VariableIndex(3)
 MOI.VariableIndex(4)
 MOI.VariableIndex(5)
 MOI.VariableIndex(6)

julia> MOI.add_constraint.(model, ns, MOI.ZeroOne())
4-element Vector{MathOptInterface.ConstraintIndex{MathOptInterface.VariableIndex,
↪  MathOptInterface.ZeroOne}}:
 MathOptInterface.ConstraintIndex{MathOptInterface.VariableIndex, MathOptInterface.ZeroOne}(3)
 MathOptInterface.ConstraintIndex{MathOptInterface.VariableIndex, MathOptInterface.ZeroOne}(4)
 MathOptInterface.ConstraintIndex{MathOptInterface.VariableIndex, MathOptInterface.ZeroOne}(5)
 MathOptInterface.ConstraintIndex{MathOptInterface.VariableIndex, MathOptInterface.ZeroOne}(6)

julia> es = MOI.add_variables(model, E)
5-element Vector{MathOptInterface.VariableIndex}:
 MOI.VariableIndex(7)
 MOI.VariableIndex(8)
 MOI.VariableIndex(9)
 MOI.VariableIndex(10)
 MOI.VariableIndex(11)

julia> MOI.add_constraint.(model, es, MOI.ZeroOne())
5-element Vector{MathOptInterface.ConstraintIndex{MathOptInterface.VariableIndex,
↪  MathOptInterface.ZeroOne}}:
 MathOptInterface.ConstraintIndex{MathOptInterface.VariableIndex, MathOptInterface.ZeroOne}(7)
 MathOptInterface.ConstraintIndex{MathOptInterface.VariableIndex, MathOptInterface.ZeroOne}(8)
 MathOptInterface.ConstraintIndex{MathOptInterface.VariableIndex, MathOptInterface.ZeroOne}(9)
 MathOptInterface.ConstraintIndex{MathOptInterface.VariableIndex, MathOptInterface.ZeroOne}(10)
 MathOptInterface.ConstraintIndex{MathOptInterface.VariableIndex, MathOptInterface.ZeroOne}(11)

julia> MOI.add_constraint(model, MOI.VectorOfVariables([s; t; ns; es]), MOI.Path(from, to))
MathOptInterface.ConstraintIndex{MathOptInterface.VectorOfVariables, MathOptInterface.Path}(1)
```

MathOptInterface.Reified – Type.

```
Reified(set::AbstractSet)
```

The constraint $[z; f(x)] \in Reified(S)$ ensures that $f(x) \in S$ if and only if $z == 1$, where $z \in \{0, 1\}$.

```julia
julia> import MathOptInterface as MOI

julia> model = MOI.Utilities.UniversalFallback(MOI.Utilities.Model{Float64}())
MOIU.UniversalFallback{MOIU.Model{Float64}}
fallback for MOIU.Model{Float64}

julia> z, _ = MOI.add_constrained_variable(model, MOI.ZeroOne())
(MOI.VariableIndex(1), MathOptInterface.ConstraintIndex{MathOptInterface.VariableIndex,
↪   MathOptInterface.ZeroOne}(1))

julia> x = MOI.add_variable(model)
MOI.VariableIndex(2)

julia> MOI.add_constraint(
           model,
           MOI.Utilities.vectorize([z, 2.0 * x]),
           MOI.Reified(MOI.GreaterThan(1.0)),
       )
MathOptInterface.ConstraintIndex{MathOptInterface.VectorAffineFunction{Float64},
↪   MathOptInterface.Reified{MathOptInterface.GreaterThan{Float64}}}(1)
```

MathOptInterface.Table – Type.

```
Table(table::Matrix{T}) where {T}
```

The set $\{x \in \mathbb{R}^d\}$ where d = size(table, 2), such that x belongs to one row of table. That is, there exists some j in 1:size(table, 1), such that x[i] = table[j, i] for all i=1:size(table, 2).

Also known as

This constraint is called table in MiniZinc.

Example

```julia
julia> import MathOptInterface as MOI

julia> model = MOI.Utilities.Model{Float64}()
MOIU.Model{Float64}

julia> x = MOI.add_variables(model, 3)
3-element Vector{MathOptInterface.VariableIndex}:
 MOI.VariableIndex(1)
 MOI.VariableIndex(2)
 MOI.VariableIndex(3)

julia> table = Float64[1 1 0; 0 1 1; 1 0 1; 1 1 1]
4×3 Matrix{Float64}:
```

```
  1.0  1.0  0.0
  0.0  1.0  1.0
  1.0  0.0  1.0
  1.0  1.0  1.0

julia> MOI.add_constraint(model, MOI.VectorOfVariables(x), MOI.Table(table))
MathOptInterface.ConstraintIndex{MathOptInterface.VectorOfVariables,
↪  MathOptInterface.Table{Float64}}(1)
```

Matrix sets

Matrix sets are vectorized to be subtypes of AbstractVectorSet.

For sets of symmetric matrices, storing both the (i, j) and (j, i) elements is redundant. Use the AbstractSymmetricMatrixSe
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 additional constraints
are needed to force the equality of the (i, j) and (j, i) elements, use the AbstractSymmetricMatrixSetSquare
set.

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.)

Example

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 AbstractSymmetricMatrix

$$\max_{x \in \mathbb{R}} \quad x$$
$$\text{s.t.} \quad (1, -x, 1) \in \text{PositiveSemidefiniteConeTriangle}(2).$$

The dual is the following problem

$$\min_{x \in \mathbb{R}^3} \quad y_1 + y_3$$
$$\text{s.t.} \quad 2y_2 = 1$$
$$y \in \text{PositiveSemidefiniteConeTriangle}(2).$$

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.

Example

PositiveSemidefiniteConeSquare is a subtype of AbstractSymmetricMatrixSetSquare and constrain-
ing 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{
        ScaledPositiveSemidefiniteConeTriangle,
        AbstractSymmetricMatrixSetTriangle,
        AbstractSymmetricMatrixSetSquare,
    },
)
```

Side dimension of the matrices in set.

Convention

By convention, the side dimension should be stored in the side_dimension field. If this is not the case for a subtype of AbstractSymmetricMatrixSetTriangle, you must implement this method.

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::Int) <: AbstractSymmetricMatrixSetTriangle
```

The (vectorized) cone of symmetric positive semidefinite matrices, with non-negative side_dimension rows and columns.

See AbstractSymmetricMatrixSetTriangle for more details on the vectorized form.

MathOptInterface.PositiveSemidefiniteConeSquare – Type.

```
PositiveSemidefiniteConeSquare(side_dimension::Int) <: AbstractSymmetricMatrixSetSquare
```

The cone of symmetric positive semidefinite matrices, with non-negative 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.

Example

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.HermitianPositiveSemidefiniteConeTriangle` – Type.

```
HermitianPositiveSemidefiniteConeTriangle(side_dimension::Int) <: AbstractVectorSet
```

The (vectorized) cone of Hermitian positive semidefinite matrices, with non-negative `side_dimension` rows and columns.

Becaue the matrix is Hermitian, the diagonal elements are real, and the complex-valued lower triangular entries are obtained as the conjugate of corresponding upper triangular entries.

Vectorization format

The vectorized form starts with real part of the entries of the upper triangular part of the matrix, given column by column as explained in `AbstractSymmetricMatrixSetSquare`.

It is then followed by the imaginary part of the off-diagonal entries of the upper triangular part, also given column by column.

For example, the matrix

$$\begin{bmatrix} 1 & 2 + 7im & 4 + 8im \\ 2 - 7im & 3 & 5 + 9im \\ 4 - 8im & 5 - 9im & 6 \end{bmatrix}$$

has `side_dimension` 3 and is represented as the vector $[1, 2, 3, 4, 5, 6, 7, 8, 9]$.

`MathOptInterface.ScaledPositiveSemidefiniteConeTriangle` – Type.

```
ScaledPositiveSemidefiniteConeTriangle(side_dimension::Int) <: AbstractVectorSet
```

The scaled (vectorized) cone of symmetric positive semidefinite matrices, with non-negative `side_dimension` rows and columns.

Compared to the PositiveSemidefiniteConeTriangle, the off-diagonal entries are scaled by √2. Thanks to this scaling, Utilities.set_dot is the simply the sum of the pairwise product, while for PositiveSemidefiniteConeTriang the pairwise product additionally have to be multiplied by 2.

Example

```julia
julia> import MathOptInterface as MOI

julia> model = MOI.Utilities.Model{Float64}()
MOIU.Model{Float64}

julia> x = MOI.add_variables(model, 3);

julia> MOI.add_constraint(
           model,
           MOI.VectorOfVariables(x),
           MOI.ScaledPositiveSemidefiniteConeTriangle(2),
       )
MathOptInterface.ConstraintIndex{MathOptInterface.VectorOfVariables,
↪  MathOptInterface.ScaledPositiveSemidefiniteConeTriangle}(1)
```

MathOptInterface.LogDetConeTriangle – Type.

```julia
LogDetConeTriangle(side_dimension::Int)
```

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 non-negative argument side_dimension is the side dimension of the matrix X, i.e., its number of rows or columns.

Example

```julia
julia> import MathOptInterface as MOI

julia> model = MOI.Utilities.Model{Float64}()
MOIU.Model{Float64}

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

julia> X = MOI.add_variables(model, 3);

julia> MOI.add_constraint(
           model,
           MOI.VectorOfVariables([t; X]),
           MOI.LogDetConeTriangle(2),
       )
MathOptInterface.ConstraintIndex{MathOptInterface.VectorOfVariables,
↪  MathOptInterface.LogDetConeTriangle}(1)
```

MathOptInterface.LogDetConeSquare – Type.

```
LogDetConeSquare(side_dimension::Int)
```

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 non-negative argument side_dimension is the side dimension of the matrix X, i.e., its number of rows or columns.

Example

```julia
julia> import MathOptInterface as MOI

julia> model = MOI.Utilities.Model{Float64}()
MOIU.Model{Float64}

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

julia> X = reshape(MOI.add_variables(model, 4), 2, 2)
2×2 Matrix{MathOptInterface.VariableIndex}:
 MOI.VariableIndex(2)  MOI.VariableIndex(4)
 MOI.VariableIndex(3)  MOI.VariableIndex(5)

julia> MOI.add_constraint(
           model,
           MOI.VectorOfVariables([t; vec(X)]),
           MOI.LogDetConeSquare(2),
       )
MathOptInterface.ConstraintIndex{MathOptInterface.VectorOfVariables,
↪   MathOptInterface.LogDetConeSquare}(1)
```

MathOptInterface.RootDetConeTriangle – Type.

```
RootDetConeTriangle(side_dimension::Int)
```

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 non-negative argument side_dimension is the side dimension of the matrix X, i.e., its number of rows or columns.

Example

```julia
julia> import MathOptInterface as MOI

julia> model = MOI.Utilities.Model{Float64}()
MOIU.Model{Float64}

julia> t = MOI.add_variable(model)
MOI.VariableIndex(1)
```

```
julia> X = MOI.add_variables(model, 3);

julia> MOI.add_constraint(
           model,
           MOI.VectorOfVariables([t; X]),
           MOI.RootDetConeTriangle(2),
       )
MathOptInterface.ConstraintIndex{MathOptInterface.VectorOfVariables,
↪   MathOptInterface.RootDetConeTriangle}(1)
```

MathOptInterface.RootDetConeSquare – Type.

```
RootDetConeSquare(side_dimension::Int)
```

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 non-negative argument side_dimension is the side dimension of the matrix X, i.e., its number of rows or columns.

Example

```
julia> import MathOptInterface as MOI

julia> model = MOI.Utilities.Model{Float64}()
MOIU.Model{Float64}

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

julia> X = reshape(MOI.add_variables(model, 4), 2, 2)
2×2 Matrix{MathOptInterface.VariableIndex}:
 MOI.VariableIndex(2)  MOI.VariableIndex(4)
 MOI.VariableIndex(3)  MOI.VariableIndex(5)

julia> MOI.add_constraint(
           model,
           MOI.VectorOfVariables([t; vec(X)]),
           MOI.RootDetConeSquare(2),
       )
MathOptInterface.ConstraintIndex{MathOptInterface.VectorOfVariables,
↪   MathOptInterface.RootDetConeSquare}(1)
```

41.2 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, CanonicalConstraintFunction, ConstraintFunction and ConstraintSet: these attributes are set indirectly by add_constraint and add_constraints.

MathOptInterface.get – Function.

```
MOI.get(b::AbstractBridge, ::MOI.NumberOfVariables)::Int64
```

Return the number of variables created by the bridge b in the model.

See also MOI.NumberOfConstraints.

Implementation notes

- There is a default fallback, so you need only implement this if the bridge adds new variables.

```
MOI.get(b::AbstractBridge, ::MOI.ListOfVariableIndices)
```

Return the list of variables created by the bridge b.

See also `MOI.ListOfVariableIndices`.

Implementation notes

- There is a default fallback, so you need only implement this if the bridge adds new variables.

```
MOI.get(b::AbstractBridge, ::MOI.NumberOfConstraints{F,S})::Int64 where {F,S}
```

Return the number of constraints of the type F-in-S created by the bridge b.

See also `MOI.NumberOfConstraints`.

Implementation notes

- There is a default fallback, so you need only implement this for the constraint types returned by
  `added_constraint_types`.

```
MOI.get(b::AbstractBridge, ::MOI.ListOfConstraintIndices{F,S}) where {F,S}
```

Return a `Vector{ConstraintIndex{F,S}}` with indices of all constraints of type F-in-S created by the bride b.

See also `MOI.ListOfConstraintIndices`.

Implementation notes

- There is a default fallback, so you need only implement this for the constraint types returned by
  `added_constraint_types`.

```
function MOI.get(
    model::MOI.ModelLike,
    attr::MOI.AbstractConstraintAttribute,
    bridge::AbstractBridge,
)
```

Return the value of the attribute `attr` of the model `model` for the constraint bridged by `bridge`.

```
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}},
) where {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,S}
```

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.

```
get(model::GenericModel, attr::MathOptInterface.AbstractOptimizerAttribute)
```

Return the value of the attribute `attr` from the model's MOI backend.

source

```
get(model::GenericModel, attr::MathOptInterface.AbstractModelAttribute)
```

Return the value of the attribute `attr` from the model's MOI backend.

source

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.

```
function MOI.set(
    model::MOI.ModelLike,
    attr::MOI.AbstractConstraintAttribute,
    bridge::AbstractBridge,
    value,
)
```

Set the value of the attribute `attr` of the model `model` for the constraint bridged by `bridge`.

```
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,
) where {F,S}
```

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.

```
set(
    model::ModelLike,
    ::ConstraintSet,
    c::ConstraintIndex{F,S},
    set::S,
) where {F,S}
```

Change the set of constraint `c` to the new set `set` which should be of the same type as the original set.

```
set(
    model::ModelLike,
    ::ConstraintFunction,
    c::ConstraintIndex{F,S},
    func::F,
) where {F,S}
```

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`; a `SettingVariableIndexNotAllowed` error is thrown instead. This is because, it would require changing the index `c` since the index of `VariableIndex` constraints must be the same as the index of the variable.

`MathOptInterface.supports` – Function.

```
MOI.supports(
    model::MOI.ModelLike,
    attr::MOI.AbstractConstraintAttribute,
    BT::Type{<:AbstractBridge},
)
```

Return a Bool indicating whether BT supports setting attr to model.

```
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)
```

Write the current model to the file at `filename`.

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, look up each variable index by name using `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.

AbstractOptimizerAttributes 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 Utilities 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.

Interaction with `ObjectiveFunction`

Setting the sense to FEASIBILITY_SENSE unsets the ObjectiveFunction attribute. That is, if you first set ObjectiveFunction and then set ObjectiveSense to be FEASIBILITY_SENSE, no objective function will be passed to the solver.

In addition, some reformulations of ObjectiveFunction via bridges rely on the value of ObjectiveSense. Therefore, you should set ObjectiveSense before setting ObjectiveFunction.

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 a 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.optimize!` – Method.

```
optimize!(dest::AbstractOptimizer, src::ModelLike)::Tuple{IndexMap,Bool}
```

A "one-shot" call that copies the problem from `src` into `dest` and then uses `dest` to optimize the problem.

Returns a tuple of an `IndexMap` and a Bool copied.

- The `IndexMap` object translates variable and constraint indices from the `src` model to the corresponding indices in the `dest` optimizer. See `copy_to` for details.

- If `copied == true`, `src` was copied to `dest` and then cached, allowing incremental modification if supported by the solver.

- If `copied == false`, a cache of the model was not kept in `dest`. Therefore, only the solution information (attributes for which `is_set_by_optimize` is true) is available to query.

> **Note**
>
> The main purpose of `optimize!` method with two arguments is for use in `Utilities.CachingOptimizer`.

Relationship to the single-argument `optimize!`

The default fallback of `optimize!(dest::AbstractOptimizer, src::ModelLike)` is

```
function optimize!(dest::AbstractOptimizer, src::ModelLike)
    index_map = copy_to(dest, src)
    optimize!(dest)
    return index_map, true
end
```

Therefore, subtypes of `AbstractOptimizer` should either implement this two-argument method, or implement both `copy_to(::Optimizer, ::ModelLike)` and `optimize!(::Optimizer)`.

`MathOptInterface.instantiate` – Function.

```
instantiate(
    optimizer_constructor,
    with_cache_type::Union{Nothing,Type} = nothing,
    with_bridge_type::Union{Nothing,Type} = nothing,
)
```

Create an instance of an 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.

withcachetype

If `with_cache_type` is not `nothing`, then the optimizer is wrapped in a `Utilities.CachingOptimizer` to store a cache of the model. This is most useful if the optimizer you are constructing does not support the incremental interface (see `supports_incremental_interface`).

withbridgetype

If `with_bridge_type` is not `nothing`, the optimizer is wrapped in a `Bridges.full_bridge_optimizer`, enabling all the bridges defined in the MOI.Bridges submodule with coefficient type `with_bridge_type`.

In addition, if the optimizer created by `optimizer_constructor` does not support the incremental interface (see `supports_incremental_interface`), then, irrespective of `with_cache_type`, the optimizer is wrapped in a `Utilities.CachingOptimizer` to store a cache of the bridged model.

If `with_cache_type` and `with_bridge_type` are both not `nothing`, then they must be the same type.

MathOptInterface.default_cache – Function.

```
default_cache(optimizer::ModelLike, ::Type{T}) where {T}
```

Return a new instance of the default model type to be used as cache for optimizer in a `Utilities.CachingOptimizer`
for holding constraints of coefficient type T. By default, this returns `Utilities.UniversalFallback(Utilities.Model{T}())`
If copying from a instance of a given model type is faster for `optimizer` then a new method returning an
instance of this model type should be defined.

Optimizer attributes

MathOptInterface.AbstractOptimizerAttribute – Type.

```
AbstractOptimizerAttribute
```

Abstract supertype for attribute objects that can be used to set or get attributes (properties) of the opti-
mizer.

Notes

The difference between `AbstractOptimizerAttribute` and `AbstractModelAttribute` lies in the behavior
of `is_empty`, `empty!` and `copy_to`. Typically optimizer attributes affect only 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., 'VersionNum-
> ber("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 (in seconnds) for an optimization. When set to `nothing`, it deactivates the solver time limit. The default value is `nothing`.

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.

MathOptInterface.AbsoluteGapTolerance – Type.

```
AbsoluteGapTolerance()
```

An optimizer attribute for setting the absolute gap tolerance for an optimization. This is an optimizer attribute, and should be set before calling `optimize!`. When set to `nothing` (if supported), uses solver default.

To set a relative gap tolerance, see `RelativeGapTolerance`.

> **Warning**
>
> The mathematical definition of "absolute gap", and its treatment during the optimization, are solver-dependent. However, assuming no other limit nor issue is encountered during the optimization, most solvers that implement this attribute will stop once $|f - b|g_{abs}$, where b is the best bound, f is the best feasible objective value, and g_{abs} is the absolute gap.

`MathOptInterface.RelativeGapTolerance` – Type.

```
RelativeGapTolerance()
```

An optimizer attribute for setting the relative gap tolerance for an optimization. This is an optimizer attribute, and should be set before calling `optimize!`. When set to `nothing` (if supported), uses solver default.

If you are looking for the relative gap of the current best solution, see `RelativeGap`. If no limit nor issue is encountered during the optimization, the value of `RelativeGap` should be at most as large as `RelativeGapTolerance`.

```
# Before optimizing: set relative gap tolerance
# set 0.1% relative gap tolerance
MOI.set(model, MOI.RelativeGapTolerance(), 1e-3)
MOI.optimize!(model)

# After optimizing (assuming all went well)
# The relative gap tolerance has not changed...
MOI.get(model, MOI.RelativeGapTolerance())  # returns 1e-3
# ... and the relative gap of the obtained solution is smaller or equal to the
# tolerance
MOI.get(model, MOI.RelativeGap())  # should return something ≤ 1e-3
```

> **Warning**
>
> The mathematical definition of "relative gap", and its allowed range, are solver-dependent. Typically, solvers expect a value between `0.0` and `1.0`.

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.

For a mixed-integer program (MIP), the return value is 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 ConflictParticipationStatus(

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

41.3   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.

41.4 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.

Implementation notes

- This method depends only on the type of the inputs, not the runtime values.
- There is a default fallback, so you need only implement this method for constraint types that the bridge implements.

```
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.

41.5 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))
```

Multiple modifications in Constraint Functions

```
modify(
    model::ModelLike,
    cis::AbstractVector{<:ConstraintIndex},
    changes::AbstractVector{<:AbstractFunctionModification},
)
```

Apply multiple modifications specified by changes to the functions of constraints cis.

A ModifyConstraintNotAllowed error is thrown if modifying constraints is not supported by model.

Examples

```
modify(
    model,
    [ci, ci],
    [
        ScalarCoefficientChange{Float64}(VariableIndex(1), 1.0),
        ScalarCoefficientChange{Float64}(VariableIndex(2), 0.5),
    ],
)
```

Multiple modifications in the Objective Function

```
modify(
    model::ModelLike,
    attr::ObjectiveFunction,
    changes::AbstractVector{<:AbstractFunctionModification},
)
```

Apply multiple modifications specified by changes to the functions of constraints cis.

A ModifyObjectiveNotAllowed error is thrown if modifying objective coefficients is not supported by model.

Examples

```
modify(
    model,
    ObjectiveFunction{ScalarAffineFunction{Float64}}(),
    [
        ScalarCoefficientChange{Float64}(VariableIndex(1), 1.0),
        ScalarCoefficientChange{Float64}(VariableIndex(2), 0.5),
    ],
)
```

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}},
) where {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.

41.6 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 NLPBlockData.

MathOptInterface.NLPBoundsPair – Type.

```
NLPBoundsPair(lower::Float64, upper::Float64)
```

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 $f(x)$.

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 must not re-order variables.

Attributes

MathOptInterface.NLPBlock – Type.

```
NLPBlock()
```

An AbstractModelAttribute that stores an NLPBlockData, representing a set of nonlinear constraints, and optionally a nonlinear objective.

MathOptInterface.NLPBlockDual – Type.

```
NLPBlockDual(result_index::Int = 1)
```

An AbstractModelAttribute for 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 AbstractModelAttribute for the 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},
)::Nothing
```

Initialize d with the set of features in requested_features. Check features_available before calling initialize to see what features are supported by d.

> **Warning**
>
> This method must be called before any other methods.

Features

The following features are defined:

- :Grad: enables eval_objective_gradient
- :Jac: enables eval_constraint_jacobian

- :JacVec: enables eval_constraint_jacobian_product and eval_constraint_jacobian_transpose_product
- :Hess: enables eval_hessian_lagrangian
- :HessVec: enables eval_hessian_lagrangian_product
- :ExprGraph: enables objective_expr and constraint_expr.

In all cases, including when requested_features is empty, eval_objective and eval_constraint are supported.

Examples

```
MOI.initialize(d, Symbol[])
MOI.initialize(d, [:ExprGraph])
MOI.initialize(d, MOI.features_available(d))
```

MathOptInterface.features_available – Function.

```
features_available(d::AbstractNLPEvaluator)::Vector{Symbol}
```

Returns the subset of features available for this problem instance.

See initialize for the list of defined features.

MathOptInterface.eval_objective – Function.

```
eval_objective(d::AbstractNLPEvaluator, x::AbstractVector{T})::T where {T}
```

Evaluate the objective $f(x)$, returning a scalar value.

MathOptInterface.eval_constraint – Function.

```
eval_constraint(d::AbstractNLPEvaluator,
    g::AbstractVector{T},
    x::AbstractVector{T},
)::Nothing where {T}
```

Given a set of vector-valued constraints $l \leq g(x) \leq u$, evaluate the constraint function $g(x)$, storing the result in the vector g.

Implementation notes

When implementing this method, you must not assume that g is Vector{Float64}, but you may assume that it supports setindex! and length. For example, it may be the view of a vector.

MathOptInterface.eval_objective_gradient – Function.

```
eval_objective_gradient(
    d::AbstractNLPEvaluator,
    grad::AbstractVector{T},
    x::AbstractVector{T},
)::Nothing where {T}
```

Evaluate the gradient of the objective function $grad = \nabla f(x)$ as a dense vector, storing the result in the vector grad.

Implementation notes

When implementing this method, you must not assume that grad is Vector{Float64}, but you may assume that it supports setindex! and length. For example, it may be the view of a vector.

MathOptInterface.jacobian_structure – Function.

```
jacobian_structure(d::AbstractNLPEvaluator)::Vector{Tuple{Int64,Int64}}
```

Returns a vector of tuples, (row, column), where each indicates the position of a structurally nonzero element in 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 the nonlinear constraints $g(x)$.

The 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.

The sparsity structure is assumed to be independent of the point x.

MathOptInterface.eval_constraint_jacobian – Function.

```
eval_constraint_jacobian(d::AbstractNLPEvaluator,
    J::AbstractVector{T},
    x::AbstractVector{T},
)::Nothing where {T}
```

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.

Implementation notes

When implementing this method, you must not assume that J is Vector{Float64}, but you may assume that it supports setindex! and length. For example, it may be the view of a vector.

MathOptInterface.eval_constraint_jacobian_product – Function.

```
eval_constraint_jacobian_product(
    d::AbstractNLPEvaluator,
    y::AbstractVector{T},
    x::AbstractVector{T},
    w::AbstractVector{T},
)::Nothing where {T}
```

Computes the Jacobian-vector product $y = J_g(x)w$, storing the result in the vector y.

The vectors have dimensions such that `length(w) == length(x)`, and `length(y)` is the number of non-linear constraints.

Implementation notes

When implementing this method, you must not assume that y is `Vector{Float64}`, but you may assume that it supports `setindex!` and `length`. For example, it may be the `view` of a vector.

MathOptInterface.eval_constraint_jacobian_transpose_product – Function.

```
eval_constraint_jacobian_transpose_product(
    d::AbstractNLPEvaluator,
    y::AbstractVector{T},
    x::AbstractVector{T},
    w::AbstractVector{T},
)::Nothing where {T}
```

Computes the Jacobian-transpose-vector product $y = J_g(x)^T w$, storing the result in the vector y.

The vectors have dimensions such that `length(y) == length(x)`, and `length(w)` is the number of non-linear constraints.

Implementation notes

When implementing this method, you must not assume that y is `Vector{Float64}`, but you may assume that it supports `setindex!` and `length`. For example, it may be the `view` of a vector.

MathOptInterface.hessian_lagrangian_structure – Function.

```
hessian_lagrangian_structure(
    d::AbstractNLPEvaluator,
)::Vector{Tuple{Int64,Int64}}
```

Returns a vector of tuples, (`row`, `column`), where each indicates the position of a structurally nonzero element in the Hessian-of-the-Lagrangian matrix: $\nabla^2 f(x) + \sum_{i=1}^{m} \nabla^2 g_i(x)$.

The 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.

The sparsity structure is assumed to be independent of the point x.

MathOptInterface.hessian_objective_structure – Function.

```
hessian_objective_structure(
    d::AbstractNLPEvaluator,
)::Vector{Tuple{Int64,Int64}}
```

Returns a vector of tuples, (`row`, `column`), where each indicates the position of a structurally nonzero element in the Hessian matrix: $\nabla^2 f(x)$.

The 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.

The sparsity structure is assumed to be independent of the point x.

MathOptInterface.hessian_constraint_structure – Function.

```
hessian_constraint_structure(
    d::AbstractNLPEvaluator,
    i::Int64,
)::Vector{Tuple{Int64,Int64}}
```

Returns a vector of tuples, (row, column), where each indicates the position of a structurally nonzero element in the Hessian matrix: $\nabla^2 g_i(x)$.

The 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.

The sparsity structure is assumed to be independent of the point x.

MathOptInterface.eval_hessian_objective – Function.

```
eval_hessian_objective(
    d::AbstractNLPEvaluator,
    H::AbstractVector{T},
    x::AbstractVector{T},
)::Nothing where {T}
```

This function computes the sparse Hessian matrix: $\nabla^2 f(x)$, storing the result in the vector H in the same order as the indices returned by hessian_objective_structure.

Implementation notes

When implementing this method, you must not assume that H is Vector{Float64}, but you may assume that it supports setindex! and length. For example, it may be the view of a vector.

MathOptInterface.eval_hessian_constraint – Function.

```
eval_hessian_constraint(
    d::AbstractNLPEvaluator,
    H::AbstractVector{T},
    x::AbstractVector{T},
    i::Int64,
)::Nothing where {T}
```

This function computes the sparse Hessian matrix: $\nabla^2 g_i(x)$, storing the result in the vector H in the same order as the indices returned by hessian_constraint_structure.

Implementation notes

When implementing this method, you must not assume that H is Vector{Float64}, but you may assume that it supports setindex! and length. For example, it may be the view of a vector.

MathOptInterface.eval_hessian_lagrangian – Function.

```
eval_hessian_lagrangian(
    d::AbstractNLPEvaluator,
    H::AbstractVector{T},
    x::AbstractVector{T},
    σ::T,
    μ::AbstractVector{T},
)::Nothing where {T}
```

Given scalar weight σ and vector of constraint weights μ, this function 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.

Implementation notes

When implementing this method, you must not assume that H is Vector{Float64}, but you may assume that it supports setindex! and length. For example, it may be the view of a vector.

MathOptInterface.eval_hessian_lagrangian_product – Function.

```
eval_hessian_lagrangian_product(
    d::AbstractNLPEvaluator,
    h::AbstractVector{T},
    x::AbstractVector{T},
    v::AbstractVector{T},
    σ::T,
    μ::AbstractVector{T},
)::Nothing where {T}
```

Given scalar weight σ and vector of constraint weights μ, computes the Hessian-of-the-Lagrangian-vector product $h = \left(\sigma \nabla^2 f(x) + \sum_{i=1}^{m} \mu_i \nabla^2 g_i(x) \right) v$, storing the result in the vector h.

The vectors have dimensions such that length(h) == length(x) == length(v).

Implementation notes

When implementing this method, you must not assume that h is Vector{Float64}, but you may assume that it supports setindex! and length. For example, it may be the view of a vector.

MathOptInterface.objective_expr – Function.

```
objective_expr(d::AbstractNLPEvaluator)::Expr
```

Returns a Julia Expr object representing the expression graph of the objective function.

Format

The expression has a number of limitations, compared with arbitrary Julia expressions:

- All sums and products are flattened out as simple Expr(:+, ...) and Expr(:*, ...) objects.
- All decision variables must be of the form Expr(:ref, :x, MOI.VariableIndex(i)), where i is the ith variable in ListOfVariableIndices.
- 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, or error if they encounter unsupported functions.

Examples

The expression $x_1 + \sin(x_2 / \exp(x_3))$ is represented as

```
:(x[MOI.VariableIndex(1)] + sin(x[MOI.VariableIndex(2)] / exp(x[MOI.VariableIndex[3]]))))
```

or equivalently

```
Expr(
    :call,
    :+,
    Expr(:ref, :x, MOI.VariableIndex(1)),
    Expr(
        :call,
        :/,
        Expr(:call, :sin, Expr(:ref, :x, MOI.VariableIndex(2))),
        Expr(:call, :exp, Expr(:ref, :x, MOI.VariableIndex(3))),
    ),
)
```

MathOptInterface.constraint_expr – Function.

```
constraint_expr(d::AbstractNLPEvaluator, i::Integer)::Expr
```

Returns a Julia Expr object representing the expression graph for the ith nonlinear constraint.

Format

The format is the same as objective_expr, with an additional comparison operator indicating the sense of and bounds on the constraint.

For single-sided comparisons, the body of the constraint must be on the left-hand side, and the right-hand side must be a constant.

For double-sided comparisons (that is, $l \leq f(x) \leq u$), the body of the constraint must be in the middle, and the left- and right-hand sides must be constants.

The bounds on the constraints must match the NLPBoundsPairs passed to NLPBlockData.

Examples

```
:(x[MOI.VariableIndex(1)]^2 <= 1.0)
:(x[MOI.VariableIndex(1)]^2 >= 2.0)
:(x[MOI.VariableIndex(1)]^2 == 3.0)
:(4.0 <= x[MOI.VariableIndex(1)]^2 <= 5.0)
```

41.7 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 example, 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 no guarantee that the callback is called every time 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.

41.8 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 throw 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` is 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` is 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 (for example, 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
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
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 42

Submodules

42.1 Benchmarks

Overview

The `Benchmarks` submodule

To aid the development of efficient solver wrappers, MathOptInterface provides benchmarking capability. Benchmarking a wrapper follows a two-step process.

First, prior to making changes, create a baseline for the benchmark results on a given benchmark suite as follows:

```
using SolverPackage  # Replace with your choice of solver.
import MathOptInterface as MOI

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
import MathOptInterface as MOI

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
)
```

42.2 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 hyper-graph 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
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}
```

Now, use `optimizer` as normal, and bridging will happen lazily behind the scenes. By lazily, we mean that bridging will happen if and only 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
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.LazyBridgeOptimizer`.

First, wrap an inner optimizer:

```julia
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})
```

List of bridges

List of bridges

This section describes the Bridges.AbstractBridges that are implemented in MathOptInterface.

Constraint bridges

These bridges are subtypes of Bridges.Constraint.AbstractBridge.

MathOptInterface.Bridges.Constraint.GreaterToIntervalBridge – Type.

```
GreaterToIntervalBridge{T,F} <: Bridges.Constraint.AbstractBridge
```

GreaterToIntervalBridge implements the following reformulations:

- $f(x) \geq l$ into $f(x) \in [l, \infty)$

Source node

GreaterToIntervalBridge supports:

- F in MOI.GreaterThan{T}

Target nodes

GreaterToIntervalBridge creates:

- F in `MOI.Interval{T}`

MathOptInterface.Bridges.Constraint.LessToIntervalBridge – Type.

```
LessToIntervalBridge{T,F} <: Bridges.Constraint.AbstractBridge
```

LessToIntervalBridge implements the following reformulations:

- $f(x) \leq u$ into $f(x) \in (-\infty, u]$

Source node

LessToIntervalBridge supports:

- F in `MOI.LessThan{T}`

Target nodes

LessToIntervalBridge creates:

- F in `MOI.Interval{T}`

MathOptInterface.Bridges.Constraint.GreaterToLessBridge – Type.

```
GreaterToLessBridge{T,F,G} <: Bridges.Constraint.AbstractBridge
```

GreaterToLessBridge implements the following reformulation:

- $f(x) \geq l$ into $-f(x) \leq -l$

Source node

GreaterToLessBridge supports:

- G in `MOI.GreaterThan{T}`

Target nodes

GreaterToLessBridge creates:

- F in `MOI.LessThan{T}`

MathOptInterface.Bridges.Constraint.LessToGreaterBridge – Type.

```
LessToGreaterBridge{T,F,G} <: Bridges.Constraint.AbstractBridge
```

LessToGreaterBridge implements the following reformulation:

- $f(x) \leq u$ into $-f(x) \geq -u$

Source node

LessToGreaterBridge supports:

- G in MOI.LessThan{T}

Target nodes

LessToGreaterBridge creates:

- F in MOI.GreaterThan{T}

MathOptInterface.Bridges.Constraint.NonnegToNonposBridge – Type.

```
NonnegToNonposBridge{T,F,G} <: Bridges.Constraint.AbstractBridge
```

NonnegToNonposBridge implements the following reformulation:

- $f(x) \in \mathbb{R}_+$ into $-f(x) \in \mathbb{R}_-$

Source node

NonnegToNonposBridge supports:

- G in MOI.Nonnegatives

Target nodes

NonnegToNonposBridge creates:

- F in MOI.Nonpositives

MathOptInterface.Bridges.Constraint.NonposToNonnegBridge – Type.

```
NonposToNonnegBridge{T,F,G} <: Bridges.Constraint.AbstractBridge
```

NonposToNonnegBridge implements the following reformulation:

- $f(x) \in \mathbb{R}_-$ into $-f(x) \in \mathbb{R}_+$

Source node

NonposToNonnegBridge supports:

- G in MOI.Nonpositives

Target nodes

NonposToNonnegBridge creates:

- F in MOI.Nonnegatives

MathOptInterface.Bridges.Constraint.VectorizeBridge – Type.

```
VectorizeBridge{T,F,S,G} <: Bridges.Constraint.AbstractBridge
```

VectorizeBridge implements the following reformulations:

- $g(x) \geq a$ into $[g(x) - a] \in \mathbb{R}_+$
- $g(x) \leq a$ into $[g(x) - a] \in \mathbb{R}_-$
- $g(x) == a$ into $[g(x) - a] \in \{0\}$

where T is the coefficient type of g(x) - a.

Source node

VectorizeBridge supports:

- G in MOI.GreaterThan{T}
- G in MOI.LessThan{T}
- G in MOI.EqualTo{T}

Target nodes

VectorizeBridge creates:

- F in S, where S is one of MOI.Nonnegatives, MOI.Nonpositives, MOI.Zeros depending on the type of the input set.

MathOptInterface.Bridges.Constraint.ScalarizeBridge – Type.

```
ScalarizeBridge{T,F,S}
```

ScalarizeBridge implements the following reformulations:

- $f(x) - a \in \mathbb{R}_+$ into $f_i(x) \geq a_i$ for all i
- $f(x) - a \in \mathbb{R}_-$ into $f_i(x) \leq a_i$ for all i
- $f(x) - a \in \{0\}$ into $f_i(x) == a_i$ for all i

Source node

ScalarizeBridge supports:

- G in MOI.Nonnegatives{T}
- G in MOI.Nonpositives{T}
- G in MOI.Zeros{T}

Target nodes

ScalarizeBridge creates:

- F in S, where S is one of `MOI.GreaterThan{T}`, `MOI.LessThan{T}`, and `MOI.EqualTo{T}`, depending on the type of the input set.

MathOptInterface.Bridges.Constraint.ScalarSlackBridge – Type.

```
ScalarSlackBridge{T,F,S} <: Bridges.Constraint.AbstractBridge
```

ScalarSlackBridge implements the following reformulation:

- $f(x) \in S$ into $f(x) - y == 0$ and $y \in S$

Source node

ScalarSlackBridge supports:

- G in S, where G is not `MOI.VariableIndex` and S is not `MOI.EqualTo`

Target nodes

ScalarSlackBridge creates:

- F in `MOI.EqualTo{T}`
- `MOI.VariableIndex` in S

MathOptInterface.Bridges.Constraint.VectorSlackBridge – Type.

```
VectorSlackBridge{T,F,S} <: Bridges.Constraint.AbstractBridge
```

VectorSlackBridge implements the following reformulation:

- $f(x) \in S$ into $f(x) - y \in \{0\}$ and $y \in S$

Source node

VectorSlackBridge supports:

- G in S, where G is not `MOI.VectorOfVariables` and S is not `MOI.Zeros`

Target nodes

VectorSlackBridge creates:

- F in `MOI.Zeros`
- `MOI.VectorOfVariables` in S

MathOptInterface.Bridges.Constraint.ScalarFunctionizeBridge – Type.

```
ScalarFunctionizeBridge{T,S} <: Bridges.Constraint.AbstractBridge
```

ScalarFunctionizeBridge implements the following reformulations:

- $x \in S$ into $1x + 0 \in S$

Source node

ScalarFunctionizeBridge supports:

- [MOI.VariableIndex](#) in S

Target nodes

ScalarFunctionizeBridge creates:

- [MOI.ScalarAffineFunction{T}](#) in S

MathOptInterface.Bridges.Constraint.VectorFunctionizeBridge – Type.

```
VectorFunctionizeBridge{T,S} <: Bridges.Constraint.AbstractBridge
```

VectorFunctionizeBridge implements the following reformulations:

- $x \in S$ into $Ix + 0 \in S$

Source node

VectorFunctionizeBridge supports:

- [MOI.VectorOfVariables](#) in S

Target nodes

VectorFunctionizeBridge creates:

- [MOI.VectorAffineFunction{T}](#) in S

MathOptInterface.Bridges.Constraint.SplitComplexEqualToBridge – Type.

```
SplitComplexEqualToBridge{T,F,G} <: Bridges.Constraint.AbstractBridge
```

SplitComplexEqualToBridge implements the following reformulation:

- $f(x) + g(x) * im = a + b * im$ into $f(x) = a$ and $g(x) = b$

Source node

SplitComplexEqualToBridge supports:

- G in [MOI.EqualTo{Complex{T}}](#)

where G is a function with Complex coefficients.

Target nodes

SplitComplexEqualToBridge creates:

- F in `MOI.EqualTo{T}`

where F is the type of the real/imaginary part of G.

`MathOptInterface.Bridges.Constraint.SplitComplexZerosBridge` – Type.

```
SplitComplexZerosBridge{T,F,G} <: Bridges.Constraint.AbstractBridge
```

SplitComplexZerosBridge implements the following reformulation:

- $f(x) \in \{0\}^n$ into $\mathrm{Re}(f(x)) \in \{0\}^n$ and $\mathrm{Im}(f(x)) \in \{0\}^n$

Source node

SplitComplexZerosBridge supports:

- G in `MOI.Zeros`

where G is a function with `Complex` coefficients.

Target nodes

SplitComplexZerosBridge creates:

- F in `MOI.Zeros`

where F is the type of the real/imaginary part of G.

`MathOptInterface.Bridges.Constraint.SplitHyperRectangleBridge` – Type.

```
SplitHyperRectangleBridge{T,G,F} <: Bridges.Constraint.AbstractBridge
```

SplitHyperRectangleBridge implements the following reformulation:

- $f(x) \in \mathsf{HyperRectangle}(l, u)$ to $[f(x) - l; u - f(x)] \in \mathbb{R}_+$.

Source node

SplitHyperRectangleBridge supports:

- F in `MOI.HyperRectangle`

Target nodes

SplitHyperRectangleBridge creates:

- G in `MOI.Nonnegatives`

`MathOptInterface.Bridges.Constraint.SplitIntervalBridge` – Type.

```
SplitIntervalBridge{T,F,S,LS,US} <: Bridges.Constraint.AbstractBridge
```

SplitIntervalBridge implements the following reformulations:

- $l \leq f(x) \leq u$ into $f(x) \geq l$ and $f(x) \leq u$
- $f(x) = b$ into $f(x) \geq b$ and $f(x) \leq b$
- $f(x) \in \{0\}$ into $f(x) \in \mathbb{R}_+$ and $f(x) \in \mathbb{R}_-$

Source node

SplitIntervalBridge supports:

- F in `MOI.Interval{T}`
- F in `MOI.EqualTo{T}`
- F in `MOI.Zeros`

Target nodes

SplitIntervalBridge creates:

- F in `MOI.LessThan{T}`
- F in `MOI.GreaterThan{T}`

or

- F in `MOI.Nonnegatives`
- F in `MOI.Nonpositives`

> **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 `MOI.ConstraintSet` is set, a lower or upper bound constraint may be deleted or created accordingly.

MathOptInterface.Bridges.Constraint.SOCtoRSOCBridge – Type.

```
SOCtoRSOCBridge{T,F,G} <: Bridges.Constraint.AbstractBridge
```

SOCtoRSOCBridge implements the following reformulation:

- $||x||_2 \leq t$ into $(t + x_1)(t - x_1) \geq ||(x_2 \ldots, x_N)||_2^2$

Assumptions

- SOCtoRSOCBridge assumes that the length of x is at least one.

Source node

SOCtoRSOCBridge supports:

- G in MOI.SecondOrderCone

Target node

SOCtoRSOCBridge creates:

- F in MOI.RotatedSecondOrderCone

MathOptInterface.Bridges.Constraint.RSOCtoSOCBridge – Type.

```
RSOCtoSOCBridge{T,F,G} <: Bridges.Constraint.AbstractBridge
```

RSOCtoSOCBridge implements the following reformulation:

- $||x||_2^2 \leq 2tu$ into $||\frac{t-u}{\sqrt{2}}, x||_2 \leq \frac{t+u}{\sqrt{2}}$

Source node

RSOCtoSOCBridge supports:

- G in MOI.RotatedSecondOrderCone

Target node

RSOCtoSOCBridge creates:

- F in MOI.SecondOrderCone

MathOptInterface.Bridges.Constraint.SOCtoNonConvexQuadBridge – Type.

```
SOCtoNonConvexQuadBridge{T} <: Bridges.Constraint.AbstractBridge
```

SOCtoNonConvexQuadBridge implements the following reformulations:

- $||x||_2 \leq t$ into $\sum x^2 - t^2 \leq 0$ and $1t + 0 \geq 0$

The MOI.ScalarAffineFunction $1t + 0$ is used in case the variable has other bound constraints.

> **Warning**
>
> This transformation starts from a convex constraint and creates a non-convex constraint. Unless the solver has explicit support for detecting second-order cones in quadratic form, this may (wrongly) be interpreted by the solver as being non-convex. Therefore, this bridge is not added automatically by MOI.Bridges.full_bridge_optimizer. Care is recommended when adding this bridge to a optimizer.

Source node

SOCtoNonConvexQuadBridge supports:

- `MOI.VectorOfVariables` in `MOI.SecondOrderCone`

Target nodes

SOCtoNonConvexQuadBridge creates:

- `MOI.ScalarQuadraticFunction{T}` in `MOI.LessThan{T}`
- `MOI.ScalarAffineFunction{T}` in `MOI.GreaterThan{T}`

MathOptInterface.Bridges.Constraint.RSOCtoNonConvexQuadBridge – Type.

```
RSOCtoNonConvexQuadBridge{T} <: Bridges.Constraint.AbstractBridge
```

RSOCtoNonConvexQuadBridge implements the following reformulations:

- $||x||_2^2 \leq 2tu$ into $\sum x^2 - 2tu \leq 0$, $1t + 0 \geq 0$, and $1u + 0 \geq 0$.

The `MOI.ScalarAffineFunction`s $1t + 0$ and $1u + 0$ are used in case the variables have other bound constraints.

> **Warning**
>
> This transformation starts from a convex constraint and creates a non-convex constraint. Unless the solver has explicit support for detecting rotated second-order cones in quadratic form, this may (wrongly) be interpreted by the solver as being non-convex. Therefore, this bridge is not added automatically by `MOI.Bridges.full_bridge_optimizer`. Care is recommended when adding this bridge to a optimizer.

Source node

RSOCtoNonConvexQuadBridge supports:

- `MOI.VectorOfVariables` in `MOI.RotatedSecondOrderCone`

Target nodes

RSOCtoNonConvexQuadBridge creates:

- `MOI.ScalarQuadraticFunction{T}` in `MOI.LessThan{T}`
- `MOI.ScalarAffineFunction{T}` in `MOI.GreaterThan{T}`

MathOptInterface.Bridges.Constraint.QuadtoSOCBridge – Type.

```
QuadtoSOCBridge{T} <: Bridges.Constraint.AbstractBridge
```

QuadtoSOCBridge converts quadratic inequalities

$$\frac{1}{2}x^T Q x + a^T x \leq ub$$

into `MOI.RotatedSecondOrderCone` constraints, but it only applies when Q is positive definite.

This is because, if `Q` is positive definite, there exists `U` such that $Q = U^T U$, and so the inequality can then be rewritten as;

$$\|Ux\|_2^2 \leq 2(-a^T x + ub)$$

Therefore, `QuadtoSOCBridge` implements the following reformulations:

- $\frac{1}{2}x^T Q x + a^T x \leq ub$ into $(1, -a^T x + ub, Ux) \in RotatedSecondOrderCone$ where $Q = U^T U$
- $\frac{1}{2}x^T Q x + a^T x \geq lb$ into $(1, a^T x - lb, Ux) \in RotatedSecondOrderCone$ where $-Q = U^T U$

Source node

`QuadtoSOCBridge` supports:

- `MOI.ScalarAffineFunction{T}` in `MOI.LessThan{T}`
- `MOI.ScalarAffineFunction{T}` in `MOI.GreaterThan{T}`

Target nodes

`RelativeEntropyBridge` creates:

- `MOI.VectorAffineFunction{T}` in `MOI.RotatedSecondOrderCone`

Errors

This bridge errors if `Q` is not positive definite.

`MathOptInterface.Bridges.Constraint.SOCtoPSDBridge` – Type.

```
SOCtoPSDBridge{T,F,G} <: Bridges.Constraint.AbstractBridge
```

`SOCtoPSDBridge` implements the following reformulation:

- $||x||_2 \leq t$ into $\begin{bmatrix} t & x^\top \\ x & t\mathbf{I} \end{bmatrix} \succeq 0$

> **Warning**
>
> This bridge is not added by default by `MOI.Bridges.full_bridge_optimizer` because bridging second order cone constraints to semidefinite constraints can be achieved by the `SOCtoRSOCBridge` followed by the `RSOCtoPSDBridge`, while creating a smaller semidefinite constraint.

Source node

`SOCtoPSDBridge` supports:

- G in `MOI.SecondOrderCone`

Target nodes

SOCtoPSDBridge creates:

- F in [MOI.PositiveSemidefiniteConeTriangle]()

MathOptInterface.Bridges.Constraint.RSOCtoPSDBridge – Type.

```
RSOCtoPSDBridge{T,F,G} <: Bridges.Constraint.AbstractBridge
```

RSOCtoPSDBridge implements the following reformulation:

- $||x||_2^2 \leq 2t \cdot u$ into $\begin{bmatrix} t & x^\top \\ x & 2tu\mathbf{I} \end{bmatrix} \succeq 0$

Source node

RSOCtoPSDBridge supports:

- G in [MOI.RotatedSecondOrderCone]()

Target nodes

RSOCtoPSDBridge creates:

- F in [MOI.PositiveSemidefiniteConeTriangle]()

MathOptInterface.Bridges.Constraint.NormInfinityBridge – Type.

```
NormInfinityBridge{T,F,G} <: Bridges.Constraint.AbstractBridge
```

NormInfinityBridge implements the following reformulation:

- $|x|_\infty \leq t$ into $[t - x_i, t + x_i] \in \mathbb{R}_+$.

Source node

NormInfinityBridge supports:

- G in [MOI.NormInfinityCone{T}]()

Target nodes

NormInfinityBridge creates:

- F in [MOI.Nonnegatives]()

MathOptInterface.Bridges.Constraint.NormOneBridge – Type.

```
NormOneBridge{T,F,G} <: Bridges.Constraint.AbstractBridge
```

NormOneBridge implements the following reformulation:

- $\sum |x_i| \leq t$ into $[t - \sum y_i, y_i - x_i, y_i + x_i] \in \mathbb{R}_+$.

Source node

NormOneBridge supports:

- G in `MOI.NormOneCone{T}`

Target nodes

NormOneBridge creates:

- F in `MOI.Nonnegatives`

MathOptInterface.Bridges.Constraint.NormToPowerBridge – Type.

```
NormToPowerBridge{T,F} <: Bridges.Constraint.AbstractBridge
```

NormToPowerBridge implements the following reformulation:

- $(t,x) \in NormCone(p, 1+d)$ into $(r_i, t, x_i) \in PowerCone(1/p)$ for all i, and $\sum\limits_i r_i == t$.

For details, see Alizadeh, F., and Goldfarb, D. (2001). "Second-order cone programming." Mathematical Programming, Series B, 95:3-51.

Source node

NormToPowerBridge supports:

- F in `MOI.NormCone`

Target nodes

NormToPowerBridge creates:

- F in `MOI.PowerCone{T}`
- `MOI.ScalarAffineFunction` in `MOI.EqualTo`

MathOptInterface.Bridges.Constraint.NormOneConeToNormConeBridge – Type.

```
NormOneConeToNormConeBridge{T,F} <: Bridges.Constraint.AbstractBridge
```

NormOneConeToNormConeBridge implements the following reformulations:

- $(t,x) in NormOneCone(d)$ into $(t,x) in NormCone(1,d)$

Source node

NormOneConeToNormConeBridge supports:

- F in MOI.NormOneCone

Target nodes

NormOneConeToNormConeBridge creates:

- F in MOI.NormCone

MathOptInterface.Bridges.Constraint.SecondOrderConeToNormConeBridge – Type.

```
SecondOrderConeToNormConeBridge{T,F} <: Bridges.Constraint.AbstractBridge
```

SecondOrderConeToNormConeBridge implements the following reformulations:

- $(t, x) in SecondOrderCone(d)$ into $(t, x) in NormCone(2, d)$

Source node

SecondOrderConeToNormConeBridge supports:

- F in MOI.SecondOrderCone

Target nodes

SecondOrderConeToNormConeBridge creates:

- F in MOI.NormCone

MathOptInterface.Bridges.Constraint.NormInfinityConeToNormConeBridge – Type.

```
NormInfinityConeToNormConeBridge{T,F} <: Bridges.Constraint.AbstractBridge
```

NormInfinityConeToNormConeBridge implements the following reformulations:

- $(t, x) in NormInfinityCone(d)$ into $(t, x) in NormCone(Inf, d)$

Source node

NormInfinityConeToNormConeBridge supports:

- F in MOI.NormInfinityCone

Target nodes

NormInfinityConeToNormConeBridge creates:

- F in MOI.NormCone

MathOptInterface.Bridges.Constraint.GeoMeantoRelEntrBridge – Type.

```
GeoMeantoRelEntrBridge{T,F,G,H} <: Bridges.Constraint.AbstractBridge
```

GeoMeantoRelEntrBridge implements the following reformulation:

- $(u, w) \in GeometricMeanCone$ into $(0, w, (u + y)\mathbf{1}) \in RelativeEntropyCone$ and $y \geq 0$

Source node

GeoMeantoRelEntrBridge supports:

- H in `MOI.GeometricMeanCone`

Target nodes

GeoMeantoRelEntrBridge creates:

- G in `MOI.RelativeEntropyCone`
- F in `MOI.Nonnegatives`

Derivation

The derivation of the bridge is as follows:

$$
\begin{aligned}
(u, w) \in GeometricMeanCone &\iff u \leq \left(\prod_{i=1}^{n} w_i \right)^{1/n} \\
&\iff 0 \leq u + y \leq \left(\prod_{i=1}^{n} w_i \right)^{1/n}, y \geq 0 \\
&\iff 1 \leq \frac{\left(\prod_{i=1}^{n} w_i \right)^{1/n}}{u + y}, y \geq 0 \\
&\iff 1 \leq \left(\prod_{i=1}^{n} \frac{w_i}{u + y} \right)^{1/n}, y \geq 0 \\
&\iff 0 \leq \sum_{i=1}^{n} \log \left(\frac{w_i}{u + y} \right), y \geq 0 \\
&\iff 0 \geq \sum_{i=1}^{n} \log \left(\frac{u + y}{w_i} \right), y \geq 0 \\
&\iff 0 \geq \sum_{i=1}^{n} (u + y) \log \left(\frac{u + y}{w_i} \right), y \geq 0 \\
&\iff (0, w, (u + y)\mathbf{1}) \in RelativeEntropyCone, y \geq 0
\end{aligned}
$$

This derivation assumes that $u + y > 0$, which is enforced by the relative entropy cone.

`MathOptInterface.Bridges.Constraint.GeoMeanToPowerBridge` – Type.

```
GeoMeanToPowerBridge{T,F} <: Bridges.Constraint.AbstractBridge
```

GeoMeanToPowerBridge implements the following reformulation:

- $(y, x...) \in GeometricMeanCone(1+d)$ into $(x_1, t, y) \in PowerCone(1/d)$ and $(t, x_2, ..., x_d) in GeometricMeanC$
  which is then recursively expanded into more PowerCone constraints.

Source node

GeoMeanToPowerBridge supports:

- F in MOI.GeometricMeanCone

Target nodes

GeoMeanToPowerBridge creates:

- F in MOI.PowerCone{T}
- MOI.VectorOfVariables in MOI.Nonnegatives

MathOptInterface.Bridges.Constraint.GeoMeanBridge – Type.

```
GeoMeanBridge{T,F,G,H} <: Bridges.Constraint.AbstractBridge
```

GeoMeanBridge implements a reformulation from MOI.GeometricMeanCone into MOI.RotatedSecondOrderCone.

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 $\exists y \geq 0$ such that:

$$t \leq y,$$
$$y^4 \leq x_1 x_2 x_3 y.$$

Note that we need to create y and not use t^4 directly because t is allowed to be negative.

This is equivalent to:

$$t \leq \frac{y_1}{\sqrt{4}},$$
$$y_1^2 \leq 2y_2 y_3,$$
$$y_2^2 \leq 2x_1 x_2,$$
$$y_3^2 \leq 2x_3(y_1/\sqrt{4})$$
$$y \geq 0.$$

More generally, you can show how the geometric mean code is recursively expanded into a set of new variables y in `MOI.Nonnegatives`, a set of `MOI.RotatedSecondOrderCone` constraints, and a `MOI.LessThan` constraint between t and y_1.

Source node

GeoMeanBridge supports:

- H in `MOI.GeometricMeanCone`

Target nodes

GeoMeanBridge creates:

- F in `MOI.LessThan{T}`
- G in `MOI.RotatedSecondOrderCone`
- G in `MOI.Nonnegatives`

MathOptInterface.Bridges.Constraint.RelativeEntropyBridge – Type.

```
RelativeEntropyBridge{T,F,G,H} <: Bridges.Constraint.AbstractBridge
```

RelativeEntropyBridge implements the following reformulation that converts a `MOI.RelativeEntropyCone` into an `MOI.ExponentialCone`:

- $u \geq \sum_{i=1}^{n} w_i \log\left(\frac{w_i}{v_i}\right)$ into $y_i \geq 0$, $u \geq \sum_{i=1}^{n} y_i$, and $(-y_i, w_i, v_i) \in ExponentialCone$.

Source node

RelativeEntropyBridge supports:

- H in `MOI.RelativeEntropyCone`

Target nodes

RelativeEntropyBridge creates:

- F in `MOI.GreaterThan{T}`
- G in `MOI.ExponentialCone`

MathOptInterface.Bridges.Constraint.NormSpectralBridge – Type.

```
NormSpectralBridge{T,F,G} <: Bridges.Constraint.AbstractBridge
```

NormSpectralBridge implements the following reformulation:

- $t \geq \sigma_1(X)$ into $\begin{bmatrix} t\mathbf{I} & X^{\top} \\ X & t\mathbf{I} \end{bmatrix} \succeq 0$

Source node

NormSpectralBridge supports:

- G in `MOI.NormSpectralCone`

Target nodes

NormSpectralBridge creates:

- F in `MOI.PositiveSemidefiniteConeTriangle`

MathOptInterface.Bridges.Constraint.NormNuclearBridge – Type.

```
NormNuclearBridge{T,F,G,H} <: Bridges.Constraint.AbstractBridge
```

NormNuclearBridge implements the following reformulation:

- $t \geq \sum_i \sigma_i(X)$ into $\begin{bmatrix} U & X^\top \\ X & V \end{bmatrix} \succeq 0$ and $2t \geq tr(U) + tr(V)$.

Source node

NormNuclearBridge supports:

- H in `MOI.NormNuclearCone`

Target nodes

NormNuclearBridge creates:

- F in `MOI.GreaterThan{T}`
- G in `MOI.PositiveSemidefiniteConeTriangle`

MathOptInterface.Bridges.Constraint.SquareBridge – Type.

```
SquareBridge{T,F,G,TT,ST} <: Bridges.Constraint.AbstractBridge
```

SquareBridge implements the following reformulations:

- $(t, u, X) \in LogDetConeSquare$ into $(t, u, Y) in LogDetConeTriangle$
- $(t, X) \in RootDetConeSquare$ into $(t, Y) in RootDetConeTriangle$
- $X \in AbstractSymmetricMatrixSetSquare$ into $Y in AbstractSymmetricMatrixSetTriangle$

where Y is the upper triangluar component of X.

In addition, constraints are added as necessary to constrain the matrix X to be symmetric. For example, the constraint for the matrix:

$$\begin{pmatrix} 1 & 1+x & 2-3x \\ 1+x & 2+x & 3-x \\ 2-3x & 2+x & 2x \end{pmatrix}$$

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) $3-x == 2+x$. Note that no symmetrization constraint needs to be added between the off-diagonal entries (1, 2) and (2, 1) or between (1, 3) and (3, 1) because the expressions are the same.

Source node

SquareBridge supports:

- F in ST

Target nodes

SquareBridge creates:

- G in TT

MathOptInterface.Bridges.Constraint.HermitianToSymmetricPSDBridge – Type.

```
HermitianToSymmetricPSDBridge{T,F,G} <: Bridges.Constraint.AbstractBridge
```

HermitianToSymmetricPSDBridge implements the following reformulation:

- Hermitian positive semidefinite n x n complex matrix to a symmetric positive semidefinite 2n x 2n real matrix.

See also `MOI.Bridges.Variable.HermitianToSymmetricPSDBridge`.

Source node

HermitianToSymmetricPSDBridge supports:

- G in `MOI.HermitianPositiveSemidefiniteConeTriangle`

Target node

HermitianToSymmetricPSDBridge creates:

- F in `MOI.PositiveSemidefiniteConeTriangle`

Reformulation

The reformulation is best described by example.

The Hermitian matrix:

$$\begin{bmatrix} x_{11} & x_{12} + y_{12}im & x_{13} + y_{13}im \\ x_{12} - y_{12}im & x_{22} & x_{23} + y_{23}im \\ x_{13} - y_{13}im & x_{23} - y_{23}im & x_{33} \end{bmatrix}$$

is positive semidefinite if and only if the symmetric matrix:

$$\begin{bmatrix} x_{11} & x_{12} & x_{13} & 0 & y_{12} & y_{13} \\ & x_{22} & x_{23} & -y_{12} & 0 & y_{23} \\ & & x_{33} & -y_{13} & -y_{23} & 0 \\ & & & x_{11} & x_{12} & x_{13} \\ & & & & x_{22} & x_{23} \\ & & & & & x_{33} \end{bmatrix}$$

is positive semidefinite.

The bridge achieves this reformulation by constraining the above matrix to belong to the `MOI.PositiveSemidefiniteConeTri`

`MathOptInterface.Bridges.Constraint.SymmetricMatrixScalingBridge` – Type.

```
SymmetricMatrixScalingBridge{T,F,G} <: Bridges.Constraint.AbstractBridge
```

`SymmetricMatrixScalingBridge` implements the reformulation from constraints in the `MOI.PositiveSemidefiniteConeTria` to constraints in the `MOI.ScaledPositiveSemidefiniteConeTriangle`.

Source node

`SymmetricMatrixScalingBridge` supports:

- G in MOI.PositiveSemidefiniteConeTriangle

Target node

`SymmetricMatrixScalingBridge` creates:

- F in MOI.ScaledPositiveSemidefiniteConeTriangle

`MathOptInterface.Bridges.Constraint.SymmetricMatrixInverseScalingBridge` – Type.

```
SymmetricMatrixInverseScalingBridge{T,F,G} <: Bridges.Constraint.AbstractBridge
```

`SymmetricMatrixInverseScalingBridge` implements the reformulation from constraints in the `MOI.ScaledPositiveSemidef` to constraints in the `MOI.PositiveSemidefiniteConeTriangle`.

Source node

`SymmetricMatrixInverseScalingBridge` supports:

- G in `MOI.ScaledPositiveSemidefiniteConeTriangle`

Target node

`SymmetricMatrixInverseScalingBridge` creates:

- F in `MOI.PositiveSemidefiniteConeTriangle`

`MathOptInterface.Bridges.Constraint.RootDetBridge` – Type.

```
RootDetBridge{T,F,G,H} <: Bridges.Constraint.AbstractBridge
```

The `MOI.RootDetConeTriangle` is representable by `MOI.PositiveSemidefiniteConeTriangle` and `MOI.GeometricMeanCone` constraints, see [1, p. 149].

Indeed, $t \le \det(X)^{1/n}$ if and only if there exists a lower triangular matrix such that:

$$\begin{pmatrix} X \\ \top & \mathrm{Diag}() \end{pmatrix} \succeq 0$$
$$(t, \mathrm{Diag}()) \in GeometricMeanCone$$

Source node

`RootDetBridge` supports:

- I in `MOI.RootDetConeTriangle`

Target nodes

`RootDetBridge` creates:

- F in `MOI.PositiveSemidefiniteConeTriangle`
- G in `MOI.GeometricMeanCone`

[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} <: Bridges.Constraint.AbstractBridge
```

The `MOI.LogDetConeTriangle` is representable by `MOI.PositiveSemidefiniteConeTriangle` and `MOI.ExponentialCone` constraints.

Indeed, $\log\det(X) = \sum\limits_{i=1}^{n} \log(\delta_i)$ where δ_i are the eigenvalues of X.

Adapting the method from [1, p. 149], we see that $t \le u\log(\det(X/u))$ for $u > 0$ if and only if there exists a lower triangular matrix such that

$$\begin{pmatrix} X \\ \top & \mathrm{Diag}() \end{pmatrix} \succeq 0$$

$$t - \sum_{i=1}^{n} u \log \left(\frac{ii}{u} \right) \leq 0$$

Which we reformulate further into

$$\begin{pmatrix} X \\ \top & \mathrm{Diag}() \end{pmatrix} \succeq 0$$

$$(l_i, u, {}_{ii}) \in ExponentialCone \quad \forall i$$

$$t - \sum_{i=1}^{n} l_i \leq 0$$

Source node

LogDetBridge supports:

- I in `MOI.LogDetConeTriangle`

Target nodes

LogDetBridge creates:

- F in `MOI.PositiveSemidefiniteConeTriangle`
- G in `MOI.ExponentialCone`
- H in `MOI.LessThan{T}`

[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,F,S} <: Bridges.Constraint.AbstractBridge
```

IndicatorActiveOnFalseBridge implements the following reformulation:

- $\neg z \implies f(x) \in S$ into $y \implies f(x) \in S$, $z + y = 1$, and $y \in \{0, 1\}$

Source node

IndicatorActiveOnFalseBridge supports:

- `MOI.VectorAffineFunction{T}` in `MOI.Indicator{MOI.ACTIVATE_ON_ZERO,S}`

Target nodes

IndicatorActiveOnFalseBridge creates:

- `MOI.VectorAffineFunction{T}` in `MOI.Indicator{MOI.ACTIVATE_ON_ONE,S}`
- `MOI.ScalarAffineFunction{T}` in `MOI.EqualTo`
- `MOI.VariableIndex` in `MOI.ZeroOne`

MathOptInterface.Bridges.Constraint.IndicatorGreaterToLessThanBridge – Type.

```
IndicatorGreaterToLessThanBridge{T,A} <: Bridges.Constraint.AbstractBridge
```

IndicatorGreaterToLessThanBridge implements the following reformulation:

- $z \implies f(x) \geq l$ into $z \implies -f(x) \leq -l$

Source node

IndicatorGreaterToLessThanBridge supports:

- `MOI.VectorAffineFunction{T}` in `MOI.Indicator{A,MOI.GreaterThan{T}}`

Target nodes

IndicatorGreaterToLessThanBridge creates:

- `MOI.VectorAffineFunction{T}` in `MOI.Indicator{A,MOI.LessThan{T}}`

MathOptInterface.Bridges.Constraint.IndicatorLessToGreaterThanBridge – Type.

```
IndicatorLessToGreaterThanBridge{T,A} <: Bridges.Constraint.AbstractBridge
```

IndicatorLessToGreaterThanBridge implements the following reformulations:

- $z \implies f(x) \leq u$ into $z \implies -f(x) \geq -u$

Source node

IndicatorLessToGreaterThanBridge supports:

- `MOI.VectorAffineFunction{T}` in `MOI.Indicator{A,MOI.LessThan{T}}`

Target nodes

IndicatorLessToGreaterThanBridge creates:

- `MOI.VectorAffineFunction{T}` in `MOI.Indicator{A,MOI.GreaterThan{T}}`

MathOptInterface.Bridges.Constraint.IndicatorSOS1Bridge – Type.

```
IndicatorSOS1Bridge{T,S} <: Bridges.Constraint.AbstractBridge
```

IndicatorSOS1Bridge implements the following reformulation:

- $z \implies f(x) \in S$ into $f(x) + y \in S$, $SOS1(y, z)$

> **Warning**
>
> This bridge assumes that the solver supports `MOI.SOS1{T}` constraints in which one of the variables (y) is continuous.

Source node

IndicatorSOS1Bridge supports:

- `MOI.VectorAffineFunction{T}` in `MOI.Indicator{MOI.ACTIVATE_ON_ONE,S}`

Target nodes

IndicatorSOS1Bridge creates:

- `MOI.ScalarAffineFunction{T}` in S
- `MOI.VectorOfVariables` in `MOI.SOS1{T}`

MathOptInterface.Bridges.Constraint.SemiToBinaryBridge – Type.

```
SemiToBinaryBridge{T,S} <: Bridges.Constraint.AbstractBridge
```

SemiToBinaryBridge implements the following reformulations:

- $x \in \{0\} \cup [l, u]$ into

$$
\begin{aligned}
x &\leq zu \\
x &\geq zl \\
z &\in \{0, 1\}
\end{aligned}
$$

- $x \in \{0\} \cup \{l, \ldots, u\}$ into

$$
\begin{aligned}
x &\leq zu \\
x &\geq zl \\
z &\in \{0, 1\} \\
x &\in \mathbb{Z}
\end{aligned}
$$

Source node

SemiToBinaryBridge supports:

- `MOI.VariableIndex` in `MOI.Semicontinuous{T}`
- `MOI.VariableIndex` in `MOI.Semiinteger{T}`

Target nodes

SemiToBinaryBridge creates:

- `MOI.VariableIndex` in `MOI.ZeroOne`
- `MOI.ScalarAffineFunction{T}` in `MOI.LessThan{T}`
- `MOI.ScalarAffineFunction{T}` in `MOI.GreaterThan{T}`
- `MOI.VariableIndex{T}` in `MOI.Integer` (if S is `MOI.Semiinteger{T}`

MathOptInterface.Bridges.Constraint.ZeroOneBridge – Type.

```
ZeroOneBridge{T} <: Bridges.Constraint.AbstractBridge
```

ZeroOneBridge implements the following reformulation:

- $x \in \{0, 1\}$ into $x \in \mathbb{Z}, 1x \in [0, 1]$.

> **Note**
>
> ZeroOneBridge adds a linear constraint instead of adding variable bounds to avoid conflicting with bounds set by the user.

Source node

ZeroOneBridge supports:

- `MOI.VariableIndex` in `MOI.ZeroOne`

Target nodes

ZeroOneBridge creates:

- `MOI.VariableIndex` in `MOI.Integer`
- `MOI.ScalarAffineFunction{T}` in `MOI.Interval{T}`

MathOptInterface.Bridges.Constraint.NumberConversionBridge – Type.

```
NumberConversionBridge{T,F1,S1,F2,S2} <: Bridges.Constraint.AbstractBridge
```

NumberConversionBridge implements the following reformulation:

- $f1(x) \in S1$ to $f2(x) \in S2$

where f and S are the same functional form, but differ in their coefficient type.

Source node

NumberConversionBridge supports:

- F1 in S1

Target node

NumberConversionBridge creates:

- F2 in S2

MathOptInterface.Bridges.Constraint.AllDifferentToCountDistinctBridge – Type.

```
AllDifferentToCountDistinctBridge{T,F} <: Bridges.Constraint.AbstractBridge
```

AllDifferentToCountDistinctBridge implements the following reformulations:

- $x \in \mathsf{AllDifferent}(d)$ to $(n, x) \in \mathsf{CountDistinct}(1 + d)$ and $n = d$
- $f(x) \in \mathsf{AllDifferent}(d)$ to $(d, f(x)) \in \mathsf{CountDistinct}(1 + d)$

Source node

AllDifferentToCountDistinctBridge supports:

- F in `MOI.AllDifferent`

where F is `MOI.VectorOfVariables` or `MOI.VectorAffineFunction{T}`.

Target nodes

AllDifferentToCountDistinctBridge creates:

- F in `MOI.CountDistinct`
- `MOI.VariableIndex` in `MOI.EqualTo{T}`

MathOptInterface.Bridges.Constraint.ReifiedAllDifferentToCountDistinctBridge – Type.

```
ReifiedAllDifferentToCountDistinctBridge{T,F} <:
Bridges.Constraint.AbstractBridge
```

ReifiedAllDifferentToCountDistinctBridge implements the following reformulations:

- $r \iff x \in \mathsf{AllDifferent}(d)$ to $r \iff (n, x) \in \mathsf{CountDistinct}(1 + d)$ and $n = d$
- $r \iff f(x) \in \mathsf{AllDifferent}(d)$ to $r \iff (d, f(x)) \in \mathsf{CountDistinct}(1 + d)$

Source node

ReifiedAllDifferentToCountDistinctBridge supports:

- F in `MOI.Reified{MOI.AllDifferent}`

where F is `MOI.VectorOfVariables` or `MOI.VectorAffineFunction{T}`.

Target nodes

ReifiedAllDifferentToCountDistinctBridge creates:

- F in `MOI.Reified{MOI.CountDistinct}`
- `MOI.VariableIndex` in `MOI.EqualTo{T}`

MathOptInterface.Bridges.Constraint.BinPackingToMILPBridge – Type.

```
BinPackingToMILPBridge{T,F} <: Bridges.Constraint.AbstractBridge
```

BinPackingToMILPBridge implements the following reformulation:

- $x \in BinPacking(c, w)$ into a mixed-integer linear program.

Reformulation

The reformulation is non-trivial, and it depends on the finite domain of each variable x_i, which we as define $S_i = \{l_i, \ldots, u_i\}$.

First, we introduce new binary variables z_{ij}, which are 1 if variable x_i takes the value j in the optimal solution and 0 otherwise:

$$z_{ij} \in \{0, 1\} \;\; \forall i \in 1 \ldots d, j \in S_i$$
$$x_i - \sum_{j \in S_i} j \cdot z_{ij} = 0 \;\; \forall i \in 1 \ldots d$$
$$\sum_{j \in S_i} z_{ij} = 1 \;\; \forall i \in 1 \ldots d$$

Then, we add the capacity constraint for all possible bins j:

$$\sum_i w_i z_{ij} \leq c \forall j \in \bigcup_{i=1,\ldots,d} S_i$$

Source node

BinPackingToMILPBridge supports:

- F in `MOI.BinPacking{T}`

Target nodes

BinPackingToMILPBridge creates:

- `MOI.VariableIndex` in `MOI.ZeroOne`
- `MOI.ScalarAffineFunction{T}` in `MOI.EqualTo{T}`
- `MOI.ScalarAffineFunction{T}` in `MOI.LessThan{T}`

MathOptInterface.Bridges.Constraint.CircuitToMILPBridge – Type.

```
CircuitToMILPBridge{T,F} <: Bridges.Constraint.AbstractBridge
```

CircuitToMILPBridge implements the following reformulation:

- $x \in \text{Circuit}(d)$ to the Miller-Tucker-Zemlin formulation of the Traveling Salesperson Problem.

Source node

CircuitToMILPBridge supports:

- F in MOI.Circuit

where F is MOI.VectorOfVariables or MOI.VectorAffineFunction{T}.

Target nodes

CircuitToMILPBridge creates:

- MOI.VariableIndex in MOI.ZeroOne
- MOI.VariableIndex in MOI.Integer
- MOI.VariableIndex in MOI.Interval{T}
- MOI.ScalarAffineFunction{T} in MOI.EqualTo{T}
- MOI.ScalarAffineFunction{T} in MOI.LessThan{T}

MathOptInterface.Bridges.Constraint.CountAtLeastToCountBelongsBridge – Type.

```
CountAtLeastToCountBelongsBridge{T,F} <: Bridges.Constraint.AbstractBridge
```

CountAtLeastToCountBelongsBridge implements the following reformulation:

- $x \in \mathsf{CountAtLeast}(n, d, \mathcal{S})$ to $(n_i, x_{d_i}) \in \mathsf{CountBelongs}(1 + d, \mathcal{S})$ and $n_i \geq n$ for all i.

Source node

CountAtLeastToCountBelongsBridge supports:

- F in MOI.CountAtLeast

where F is MOI.VectorOfVariables or MOI.VectorAffineFunction{T}.

Target nodes

CountAtLeastToCountBelongsBridge creates:

- F in MOI.CountBelongs
- MOI.VariableIndex in MOI.GreaterThan{T}

MathOptInterface.Bridges.Constraint.CountBelongsToMILPBridge – Type.

```
CountBelongsToMILPBridge{T,F} <: Bridges.Constraint.AbstractBridge
```

CountBelongsToMILPBridge implements the following reformulation:

- $(n, x) \in \mathsf{CountBelongs}(1 + d, \mathcal{S})$ into a mixed-integer linear program.

Reformulation

The reformulation is non-trivial, and it depends on the finite domain of each variable x_i, which we as define $S_i = \{l_i, \ldots, u_i\}$.

First, we introduce new binary variables z_{ij}, which are 1 if variable x_i takes the value j in the optimal solution and 0 otherwise:

$$
z_{ij} \in \{0,1\} \;\; \forall i \in 1\ldots d, j \in S_i
$$
$$
x_i - \sum_{j \in S_i} j \cdot z_{ij} = 0 \;\; \forall i \in 1\ldots d
$$
$$
\sum_{j \in S_i} z_{ij} = 1 \;\; \forall i \in 1\ldots d
$$

Finally, n is constrained to be the number of z_{ij} elements that are in $\mathcal{S}$:

$$
n - \sum_{i \in 1\ldots d, j \in \mathcal{S}} z_{ij} = 0
$$

Source node

CountBelongsToMILPBridge supports:

- F in `MOI.CountBelongs`

where F is `MOI.VectorOfVariables` or `MOI.VectorAffineFunction{T}`.

Target nodes

CountBelongsToMILPBridge creates:

- `MOI.VariableIndex` in `MOI.ZeroOne`
- `MOI.ScalarAffineFunction{T}` in `MOI.EqualTo{T}`

MathOptInterface.Bridges.Constraint.CountDistinctToMILPBridge – Type.

```
CountDistinctToMILPBridge{T,F} <: Bridges.Constraint.AbstractBridge
```

CountDistinctToMILPBridge implements the following reformulation:

- $(n, x) \in$ CountDistinct$(1 + d)$ into a mixed-integer linear program.

Reformulation

The reformulation is non-trivial, and it depends on the finite domain of each variable x_i, which we as define $S_i = \{l_i, \ldots, u_i\}$.

First, we introduce new binary variables z_{ij}, which are 1 if variable x_i takes the value j in the optimal solution and 0 otherwise:

$$z_{ij} \in \{0,1\} \ \ \forall i \in 1 \ldots d, j \in S_i$$

$$x_i - \sum_{j \in S_i} j \cdot z_{ij} = 0 \ \ \forall i \in 1 \ldots d$$

$$\sum_{j \in S_i} z_{ij} = 1 \ \ \forall i \in 1 \ldots d$$

Then, we introduce new binary variables y_j, which are 1 if a variable takes the value j in the optimal solution and 0 otherwise.

$$y_j \in \{0,1\} \ \ \forall j \in \bigcup_{i=1,\ldots,d} S_i$$

$$y_j \leq \sum_{i \in 1 \ldots d : j \in S_i} z_{ij} \leq M y_j \ \ \forall j \in \bigcup_{i=1,\ldots,d} S_i$$

Finally, n is constrained to be the number of y_j elements that are non-zero:

$$n - \sum_{j \in \bigcup_{i=1,\ldots,d} S_i} y_j = 0$$

Source node

CountDistinctToMILPBridge supports:

- F in `MOI.CountDistinct`

where F is `MOI.VectorOfVariables` or `MOI.VectorAffineFunction{T}`.

Target nodes

CountDistinctToMILPBridge creates:

- `MOI.VariableIndex` in `MOI.ZeroOne`
- `MOI.ScalarAffineFunction{T}` in `MOI.EqualTo{T}`
- `MOI.ScalarAffineFunction{T}` in `MOI.LessThan{T}`

MathOptInterface.Bridges.Constraint.ReifiedCountDistinctToMILPBridge – Type.

```
ReifiedCountDistinctToMILPBridge{T,F} <: Bridges.Constraint.AbstractBridge
```

ReifiedCountDistinctToMILPBridge implements the following reformulation:

- $r \iff (n, x) \in$ CountDistinct$(1 + d)$ into a mixed-integer linear program.

Reformulation

The reformulation is non-trivial, and it depends on the finite domain of each variable x_i, which we as define $S_i = \{l_i, \ldots, u_i\}$.

First, we introduce new binary variables z_{ij}, which are 1 if variable x_i takes the value j in the optimal solution and 0 otherwise:

$$z_{ij} \in \{0, 1\} \ \ \forall i \in 1 \ldots d, j \in S_i$$
$$x_i - \sum_{j \in S_i} j \cdot z_{ij} = 0 \ \ \forall i \in 1 \ldots d$$
$$\sum_{j \in S_i} z_{ij} = 1 \ \ \forall i \in 1 \ldots d$$

Then, we introduce new binary variables y_j, which are 1 if a variable takes the value j in the optimal solution and 0 otherwise.

$$y_j \in \{0, 1\} \ \ \forall j \in \bigcup_{i=1,\ldots,d} S_i$$
$$y_j \le \sum_{i \in 1 \ldots d: j \in S_i} z_{ij} \le M y_j \ \ \forall j \in \bigcup_{i=1,\ldots,d} S_i$$

Finally, n is constrained to be the number of y_j elements that are non-zero, with some slack:

$$n - \sum_{j \in \bigcup_{i=1,\ldots,d} S_i} y_j = \delta^+ - \delta^-$$

And then the slack is constrained to respect the reif variable r:

$$d_1 \le \delta^+ \le M d_1$$
$$d_2 \le \delta^- \le M d_s$$
$$d_1 + d_2 + r = 1$$
$$d_1, d_2 \in \{0, 1\}$$

Source node

ReifiedCountDistinctToMILPBridge supports:

- F in `MOI.Reified{MOI.CountDistinct}`

where F is `MOI.VectorOfVariables` or `MOI.VectorAffineFunction{T}`.

Target nodes

ReifiedCountDistinctToMILPBridge creates:

- `MOI.VariableIndex` in `MOI.ZeroOne`
- `MOI.ScalarAffineFunction{T}` in `MOI.EqualTo{T}`
- `MOI.ScalarAffineFunction{T}` in `MOI.LessThan{T}`

`MathOptInterface.Bridges.Constraint.CountGreaterThanToMILPBridge` – Type.

```
CountGreaterThanToMILPBridge{T,F} <: Bridges.Constraint.AbstractBridge
```

CountGreaterThanToMILPBridge implements the following reformulation:

- $(c, y, x) \in CountGreaterThan()$ into a mixed-integer linear program.

Source node

CountGreaterThanToMILPBridge supports:

- F in `MOI.CountGreaterThan`

Target nodes

CountGreaterThanToMILPBridge creates:

- `MOI.VariableIndex` in `MOI.ZeroOne`
- `MOI.ScalarAffineFunction{T}` in `MOI.EqualTo{T}`
- `MOI.ScalarAffineFunction{T}` in `MOI.GreaterThan{T}`

MathOptInterface.Bridges.Constraint.TableToMILPBridge – Type.

```
TableToMILPBridge{T,F} <: Bridges.Constraint.AbstractBridge
```

TableToMILPBridge implements the following reformulation:

- $x \in Table(t)$ into

$$
\begin{aligned}
z_j \in \{0, 1\} \quad & \forall i, j \\
\sum_{j=1}^{n} z_j = 1 & \\
\sum_{j=1}^{n} t_{ij} z_j = x_i \quad & \forall i
\end{aligned}
$$

Source node

TableToMILPBridge supports:

- F in `MOI.Table{T}`

Target nodes

TableToMILPBridge creates:

- `MOI.VariableIndex` in `MOI.ZeroOne`
- `MOI.ScalarAffineFunction{T}` in `MOI.EqualTo{T}`

Objective bridges

These bridges are subtypes of [Bridges.Objective.AbstractBridge](#).

MathOptInterface.Bridges.Objective.FunctionizeBridge – Type.

```
FunctionizeBridge{T}
```

FunctionizeBridge implements the following reformulations:

- $\min\{x\}$ into $\min\{1x + 0\}$
- $\max\{x\}$ into $\max\{1x + 0\}$

where T is the coefficient type of 1 and 0.

Source node

FunctionizeBridge supports:

- [MOI.ObjectiveFunction{MOI.VariableIndex}](#)

Target nodes

FunctionizeBridge creates:

- One objective node: [MOI.ObjectiveFunction{MOI.ScalarAffineFunction{T}}](#)

MathOptInterface.Bridges.Objective.QuadratizeBridge – Type.

```
QuadratizeBridge{T}
```

QuadratizeBridge implements the following reformulations:

- $\min\{a^\top x + b\}$ into $\min\{x^\top \mathbf{0} x + a^\top x + b\}$
- $\max\{a^\top x + b\}$ into $\max\{x^\top \mathbf{0} x + a^\top x + b\}$

where T is the coefficient type of 0.

Source node

QuadratizeBridge supports:

- [MOI.ObjectiveFunction{MOI.ScalarAffineFunction{T}}](#)

Target nodes

QuadratizeBridge creates:

- One objective node: [MOI.ObjectiveFunction{MOI.ScalarQuadraticFunction{T}}](#)

MathOptInterface.Bridges.Objective.SlackBridge – Type.

```
SlackBridge{T,F,G}
```

SlackBridge implements the following reformulations:

- $\min\{f(x)\}$ into $\min\{y \mid f(x) - y \leq 0\}$
- $\max\{f(x)\}$ into $\max\{y \mid f(x) - y \geq 0\}$

where F is the type of `f(x) - y`, G is the type of `f(x)`, and T is the coefficient type of `f(x)`.

Source node

SlackBridge supports:

- `MOI.ObjectiveFunction{G}`

Target nodes

SlackBridge creates:

- One variable node: `MOI.VariableIndex` in `MOI.Reals`
- One objective node: `MOI.ObjectiveFunction{MOI.VariableIndex}`
- One constraint node, that depends on the `MOI.ObjectiveSense`:
    - F-in-`MOI.LessThan` if MIN_SENSE
    - F-in-`MOI.GreaterThan` if MAX_SENSE

> **Warning**
>
> When using this bridge, changing the optimization sense is not supported. Set the sense to `MOI.FEASIBILITY_SENSE` first to delete the bridge, then set `MOI.ObjectiveSense` and re-add the objective.

`MathOptInterface.Bridges.Objective.VectorFunctionizeBridge` – Type.

```
VectorFunctionizeBridge{T}
```

VectorFunctionizeBridge implements the following reformulations:

- $\min\{x\}$ into $\min\{1x + 0\}$
- $\max\{x\}$ into $\max\{1x + 0\}$

where T is the coefficient type of 1 and 0.

Source node

VectorFunctionizeBridge supports:

- `MOI.ObjectiveFunction{MOI.VectorOfVariables}`

Target nodes

VectorFunctionizeBridge creates:

- One objective node: MOI.ObjectiveFunction{MOI.VectorAffineFunction{T}}

MathOptInterface.Bridges.Objective.VectorSlackBridge – Type.

```
VectorSlackBridge{T,F,G}
```

VectorSlackBridge implements the following reformulations:

- $\min\{f(x)\}$ into $\min\{y \mid y - f(x) \in \mathbb{R}_+\}$
- $\max\{f(x)\}$ into $\max\{y \mid f(x) - y \in \mathbb{R}_+\}$

where F is the type of f(x) - y, G is the type of f(x), and T is the coefficient type of f(x).

Source node

VectorSlackBridge supports:

- MOI.ObjectiveFunction{G}

Target nodes

VectorSlackBridge creates:

- One variable node: MOI.VectorOfVariables in MOI.Reals
- One objective node: MOI.ObjectiveFunction{MOI.VectorOfVariables}
- One constraint node: F-in-MOI.Nonnegatives

> **Warning**
>
> When using this bridge, changing the optimization sense is not supported. Set the sense to
> MOI.FEASIBILITY_SENSE first to delete the bridge, then set MOI.ObjectiveSense and re-add the
> objective.

Variable bridges

These bridges are subtypes of Bridges.Variable.AbstractBridge.

MathOptInterface.Bridges.Variable.FreeBridge – Type.

```
FreeBridge{T} <: Bridges.Variable.AbstractBridge
```

FreeBridge implements the following reformulation:

- $x \in \mathbb{R}$ into $y, z \geq 0$ with the substitution rule $x = y - z$,

where T is the coefficient type of y - z.

Source node

FreeBridge supports:

- `MOI.VectorOfVariables` in `MOI.Reals`

Target nodes

FreeBridge creates:

- One variable node: `MOI.VectorOfVariables` in `MOI.Nonnegatives`

`MathOptInterface.Bridges.Variable.NonposToNonnegBridge` – Type.

```
NonposToNonnegBridge{T} <: Bridges.Variable.AbstractBridge
```

NonposToNonnegBridge implements the following reformulation:

- $x \in \mathbb{R}_-$ into $y \in \mathbb{R}_+$ with the substitution rule $x = -y$,

where T is the coefficient type of -y.

Source node

NonposToNonnegBridge supports:

- `MOI.VectorOfVariables` in `MOI.Nonpositives`

Target nodes

NonposToNonnegBridge creates:

- One variable node: `MOI.VectorOfVariables` in `MOI.Nonnegatives`,

`MathOptInterface.Bridges.Variable.RSOCtoPSDBridge` – Type.

```
RSOCtoPSDBridge{T} <: Bridges.Variable.AbstractBridge
```

RSOCtoPSDBridge implements the following reformulation:

- $||x||_2^2 \leq 2tu$ where $t, u \geq 0$ into $Y \succeq 0$, with the substitution rule: $Y = \begin{bmatrix} t & x^\top \\ x & 2u\mathbf{I} \end{bmatrix}$.

Additional bounds are added to ensure the off-diagonals of the $2uI$ submatrix are 0, and linear constraints are added to ennsure the diagonal of $2uI$ takes the same values.

As a special case, if $|x|| = 0$, then RSOCtoPSDBridge reformulates into $(t, u) \in \mathbb{R}_+$.

Source node

RSOCtoPSDBridge supports:

- `MOI.VectorOfVariables` in `MOI.RotatedSecondOrderCone`

Target nodes

`RSOCtoPSDBridge` creates:

- One variable node that depends on the input dimension:
    - `MOI.VectorOfVariables` in `MOI.Nonnegatives` if dimension is 1 or 2
    - `MOI.VectorOfVariables` in
    `MOI.PositiveSemidefiniteConeTriangle` otherwise
- The constraint node `MOI.VariableIndex` in `MOI.EqualTo`
- The constrant node `MOI.ScalarAffineFunction` in `MOI.EqualTo`

MathOptInterface.Bridges.Variable.RSOCtoSOCBridge – Type.

```
RSOCtoSOCBridge{T} <: Bridges.Variable.AbstractBridge
```

`RSOCtoSOCBridge` implements the following reformulation:

- $||x||_2^2 \leq 2tu$ into $||v||_2 \leq w$, with the substitution rules $t = \frac{w}{\sqrt{2}} + \frac{v_1}{\sqrt{2}}$, $u = \frac{w}{\sqrt{2}} - \frac{v_1}{\sqrt{2}}$, and $x = (v_2, \ldots, v_N)$.

Source node

`RSOCtoSOCBridge` supports:

- `MOI.VectorOfVariables` in `MOI.RotatedSecondOrderCone`

Target node

`RSOCtoSOCBridge` creates:

- `MOI.VectorOfVariables` in `MOI.SecondOrderCone`

MathOptInterface.Bridges.Variable.SOCtoRSOCBridge – Type.

```
SOCtoRSOCBridge{T} <: Bridges.Variable.AbstractBridge
```

`SOCtoRSOCBridge` implements the following reformulation:

- $||x||_2 \leq t$ into $2uv \geq ||w||_2^2$, with the substitution rules $t = \frac{u}{\sqrt{2}} + \frac{v}{\sqrt{2}}$, $x = (\frac{u}{\sqrt{2}} - \frac{v}{\sqrt{2}}, w)$.

Assumptions

- `SOCtoRSOCBridge` assumes that $|x| \geq 1$.

Source node

`SOCtoRSOCBridge` supports:

- `MOI.VectorOfVariables` in `MOI.SecondOrderCone`

Target node

`SOCtoRSOCBridge` creates:

- `MOI.VectorOfVariables` in `MOI.RotatedSecondOrderCone`

`MathOptInterface.Bridges.Variable.VectorizeBridge` – Type.

```
VectorizeBridge{T,S} <: Bridges.Variable.AbstractBridge
```

`VectorizeBridge` implements the following reformulations:

- $x \geq a$ into $[y] \in \mathbb{R}_+$ with the substitution rule $x = a + y$
- $x \leq a$ into $[y] \in \mathbb{R}_-$ with the substitution rule $x = a + y$
- $x == a$ into $[y] \in \{0\}$ with the substitution rule $x = a + y$

where T is the coefficient type of a + y.

Source node

`VectorizeBridge` supports:

- `MOI.VariableIndex` in `MOI.GreaterThan{T}`
- `MOI.VariableIndex` in `MOI.LessThan{T}`
- `MOI.VariableIndex` in `MOI.EqualTo{T}`

Target nodes

`VectorizeBridge` creates:

- One variable node: `MOI.VectorOfVariables` in S, where S is one of `MOI.Nonnegatives`, `MOI.Nonpositives`, `MOI.Zeros` depending on the type of S.

`MathOptInterface.Bridges.Variable.ZerosBridge` – Type.

```
ZerosBridge{T} <: Bridges.Variable.AbstractBridge
```

`ZerosBridge` implements the following reformulation:

- $x \in \{0\}$ into the substitution rule $x = 0$,

where T is the coefficient type of 0.

Source node

`ZerosBridge` supports:

- `MOI.VectorOfVariables` in `MOI.Zeros`

Target nodes

ZerosBridge does not create target nodes. It replaces all instances of x with 0 via substitution. This means that no variables are created in the underlying model.

Caveats

The bridged variables are similar to parameters with zero values. Parameters with non-zero values can be created with constrained variables in MOI.EqualTo by combining a VectorizeBridge and this bridge.

However, functions modified by ZerosBridge cannot be unbridged. That is, for a given function, we cannot determine if the bridged variables were used.

A related implication is that this bridge does not support MOI.ConstraintDual. However, if a MOI.Utilities.CachingOptimi: is used, the dual can be determined by the bridged optimizer using MOI.Utilities.get_fallback because the caching optimizer records the unbridged function.

MathOptInterface.Bridges.Variable.HermitianToSymmetricPSDBridge – Type.

```
HermitianToSymmetricPSDBridge{T} <: Bridges.Variable.AbstractBridge
```

HermitianToSymmetricPSDBridge implements the following reformulation:

- Hermitian positive semidefinite n x n complex matrix to a symmetric positive semidefinite 2n x 2n real matrix satisfying equality constraints described below.

Source node

HermitianToSymmetricPSDBridge supports:

- MOI.VectorOfVariables in MOI.HermitianPositiveSemidefiniteConeTriangle

Target node

HermitianToSymmetricPSDBridge creates:

- MOI.VectorOfVariables in MOI.PositiveSemidefiniteConeTriangle
- MOI.ScalarAffineFunction{T} in MOI.EqualTo{T}

Reformulation

The reformulation is best described by example.

The Hermitian matrix:

$$\begin{bmatrix} x_{11} & x_{12} + y_{12}im & x_{13} + y_{13}im \\ x_{12} - y_{12}im & x_{22} & x_{23} + y_{23}im \\ x_{13} - y_{13}im & x_{23} - y_{23}im & x_{33} \end{bmatrix}$$

is positive semidefinite if and only if the symmetric matrix:

$$\begin{bmatrix} x_{11} & x_{12} & x_{13} & 0 & y_{12} & y_{13} \\ & x_{22} & x_{23} & -y_{12} & 0 & y_{23} \\ & & x_{33} & -y_{13} & -y_{23} & 0 \\ & & & x_{11} & x_{12} & x_{13} \\ & & & & x_{22} & x_{23} \\ & & & & & x_{33} \end{bmatrix}$$

is positive semidefinite.

The bridge achieves this reformulation by adding a new set of variables in `MOI.PositiveSemidefiniteConeTriangle(6)`, and then adding three groups of equality constraints to:

- constrain the two x blocks to be equal
- force the diagonal of the y blocks to be 0
- force the lower triangular of the y block to be the negative of the upper triangle.

API Reference

Bridges

AbstractBridge API

MathOptInterface.Bridges.AbstractBridge – Type.

```
abstract type AbstractBridge end
```

An abstract type representing a bridged constraint or variable in a `MOI.Bridges.AbstractBridgeOptimizer`.

All bridges must implement:

- added_constrained_variable_types
- added_constraint_types
- MOI.get(::AbstractBridge, ::MOI.NumberOfVariables)
- MOI.get(::AbstractBridge, ::MOI.ListOfVariableIndices)
- MOI.get(::AbstractBridge, ::MOI.NumberOfConstraints)
- MOI.get(::AbstractBridge, ::MOI.ListOfConstraintIndices)

Subtypes of `AbstractBridge` may have additional requirements. Consult their docstrings for details.

In addition, all subtypes may optionally implement the following constraint attributes with the bridge in place of the constraint index:

- MOI.ConstraintDual
- MOI.ConstraintPrimal

MathOptInterface.Bridges.added_constrained_variable_types – Function.

```
added_constrained_variable_types(
    BT::Type{<:AbstractBridge},
)::Vector{Tuple{Type}}
```

Return a list of the types of constrained variables that bridges of concrete type BT add.

Implementation notes

- This method depends only on the type of the bridge, not the runtime value. If the bridge may add a constrained variable, the type must be included in the return vector.

- If the bridge adds a free variable via MOI.add_variable or MOI.add_variables, the return vector must include (MOI.Reals,).

Example

```
julia> MOI.Bridges.added_constrained_variable_types(
           MOI.Bridges.Variable.NonposToNonnegBridge{Float64},
       )
1-element Vector{Tuple{Type}}:
 (MathOptInterface.Nonnegatives,)
```

MathOptInterface.Bridges.added_constraint_types – Function.

```
added_constraint_types(
    BT::Type{<:AbstractBridge},
)::Vector{Tuple{Type,Type}}
```

Return a list of the types of constraints that bridges of concrete type BT add.

Implementation notes

- This method depends only on the type of the bridge, not the runtime value. If the bridge may add a constraint, the type must be included in the return vector.

Example

```
julia> MOI.Bridges.added_constraint_types(
           MOI.Bridges.Constraint.ZeroOneBridge{Float64},
       )
2-element Vector{Tuple{Type, Type}}:
 (MathOptInterface.ScalarAffineFunction{Float64}, MathOptInterface.Interval{Float64})
 (MathOptInterface.VariableIndex, MathOptInterface.Integer)
```

MathOptInterface.get – Method.

```
MOI.get(b::AbstractBridge, ::MOI.NumberOfVariables)::Int64
```

Return the number of variables created by the bridge b in the model.

See also MOI.NumberOfConstraints.

Implementation notes

- There is a default fallback, so you need only implement this if the bridge adds new variables.

MathOptInterface.get – Method.

```
MOI.get(b::AbstractBridge, ::MOI.ListOfVariableIndices)
```

Return the list of variables created by the bridge b.

See also MOI.ListOfVariableIndices.

Implementation notes

- There is a default fallback, so you need only implement this if the bridge adds new variables.

MathOptInterface.get – Method.

```
MOI.get(b::AbstractBridge, ::MOI.NumberOfConstraints{F,S})::Int64 where {F,S}
```

Return the number of constraints of the type F-in-S created by the bridge b.

See also MOI.NumberOfConstraints.

Implementation notes

- There is a default fallback, so you need only implement this for the constraint types returned by added_constraint_types.

MathOptInterface.get – Method.

```
MOI.get(b::AbstractBridge, ::MOI.ListOfConstraintIndices{F,S}) where {F,S}
```

Return a Vector{ConstraintIndex{F,S}} with indices of all constraints of type F-in-S created by the bride b.

See also MOI.ListOfConstraintIndices.

Implementation notes

- There is a default fallback, so you need only implement this for the constraint types returned by added_constraint_types.

MathOptInterface.Bridges.needs_final_touch – Function.

```
needs_final_touch(bridge::AbstractBridge)::Bool
```

Return whether final_touch is implemented by bridge.

MathOptInterface.Bridges.final_touch – Function.

```
final_touch(bridge::AbstractBridge, model::MOI.ModelLike)::Nothing
```

A function that is called immediately prior to `MOI.optimize!` to allow bridges to modify their reformulations with repsect to other variables and constraints in `model`.

For example, if the correctness of `bridge` depends on the bounds of a variable or the fact that variables are integer, then the bridge can implement `final_touch` to check assumptions immediately before a call to `MOI.optimize!`.

If you implement this method, you must also implement `needs_final_touch`.

Constraint bridge API

MathOptInterface.Bridges.Constraint.AbstractBridge – Type.

```
abstract type AbstractBridge <: MOI.Bridges.AbstractType
```

Subtype of `MOI.Bridges.AbstractBridge` for constraint bridges.

In addition to the required implementation described in `MOI.Bridges.AbstractBridge`, subtypes of `AbstractBridge` must additionally implement:

- `MOI.supports_constraint(::Type{<:AbstractBridge}, ::Type{<:MOI.AbstractFunction}, ::Type{<:MOI.Abst`
- `concrete_bridge_type`
- `bridge_constraint`

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.

Implementation notes

- This method depends only on the type of the inputs, not the runtime values.
- There is a default fallback, so you need only implement this method for constraint types that the bridge implements.

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`.

Example

The `SplitIntervalBridge` bridges a `MOI.VariableIndex`-in-`MOI.Interval` constraint into a `MOI.VariableIndex`-in-`MOI.GreaterThan` and a `MOI.VariableIndex`-in-`MOI.LessThan` constraint.

```julia
julia> MOI.Bridges.Constraint.concrete_bridge_type(
           MOI.Bridges.Constraint.SplitIntervalBridge{Float64},
           MOI.VariableIndex,
           MOI.Interval{Float64},
       )
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,
)::BT
```

Bridge the constraint `func`-in-`set` using bridge `BT` to `model` and returns a bridge object of type `BT`.

Implementation notes

- The bridge type BT should be a concrete type, that is, all the type parameters of the bridge must be set.

`MathOptInterface.Bridges.Constraint.AbstractFunctionConversionBridge` – *Type*.

```
abstract type AbstractFunctionConversionBridge{F,S} <: AbstractBridge end
```

Abstract type to support writing bridges in which the function changes but the set does not.

By convention, the transformed function is stored in the `.constraint` field.

`MathOptInterface.Bridges.Constraint.SingleBridgeOptimizer` – *Type*.

```
SingleBridgeOptimizer{BT<:AbstractBridge}(model::MOI.ModelLike)
```

Return `AbstractBridgeOptimizer` that always bridges any objective function supported by the bridge BT.

This is in contrast with the `MOI.Bridges.LazyBridgeOptimizer`, which only bridges the objective function if it is supported by the bridge BT and unsupported by `model`.

Example

```
julia> struct MyNewBridge{T} <: MOI.Bridges.Constraint.AbstractBridge end

julia> bridge = MOI.Bridges.Constraint.SingleBridgeOptimizer{MyNewBridge{Float64}}(
           MOI.Utilities.Model{Float64}(),
       )
MOIB.Constraint.SingleBridgeOptimizer{MyNewBridge{Float64}, MOIU.Model{Float64}}
with 0 constraint bridges
with inner model MOIU.Model{Float64}
```

Implementation notes

All bridges should simplify the creation of `SingleBridgeOptimizers` by defining a constant that wraps the bridge in a `SingleBridgeOptimizer`.

```
julia> const MyNewBridgeModel{T,OT<:MOI.ModelLike} =
           MOI.Bridges.Constraint.SingleBridgeOptimizer{MyNewBridge{T},OT};
```

This enables users to create bridged models as follows:

```
julia> MyNewBridgeModel{Float64}(MOI.Utilities.Model{Float64}())
MOIB.Constraint.SingleBridgeOptimizer{MyNewBridge{Float64}, MOIU.Model{Float64}}
with 0 constraint bridges
with inner model MOIU.Model{Float64}
```

`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.

`MathOptInterface.Bridges.Constraint.FlipSignBridge` – Type.

```
FlipSignBridge{T,S1,S2,F,G}
```

An abstract type that simplifies the creation of other bridges.

`MathOptInterface.Bridges.Constraint.AbstractToIntervalBridge` – Type.

```
AbstractToIntervalBridge{T<:AbstractFloat,S,F}
```

An abstract type that simplifies the creation of other bridges.

> **Warning**
>
> T must 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 `MOI.Bridges.full_bridge_optimizer` when T is a subtype of `AbstractFloat`.

MathOptInterface.Bridges.Constraint.SetMapBridge – Type.

```
abstract type SetMapBridge{T,S2,S1,F,G} <: AbstractBridge end
```

Consider two type of sets, S1 and S2, and a linear mapping A such 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;

- MOI.Bridges.map_set
- MOI.Bridges.map_function.

Implementing a method for these two functions is sufficient to bridge constraints. However, in order for the getters and setters of attributes such as dual solutions and starting values to work as well, a method for the following functions must be implemented:

- MOI.Bridges.inverse_map_set
- MOI.Bridges.inverse_map_function
- MOI.Bridges.adjoint_map_function
- MOI.Bridges.inverse_adjoint_map_function

See the docstrings of each function to see which feature would be missing if it was not implemented for a given bridge.

Objective bridge API

MathOptInterface.Bridges.Objective.AbstractBridge – Type.

```
abstract type AbstractBridge <: MOI.Bridges.AbstractBridge end
```

Subtype of MOI.Bridges.AbstractBridge for objective bridges.

In addition to the required implementation described in MOI.Bridges.AbstractBridge, subtypes of AbstractBridge must additionally implement:

- supports_objective_function
- concrete_bridge_type
- bridge_objective
- MOI.Bridges.set_objective_function_type

MathOptInterface.Bridges.Objective.supports_objective_function – Function.

```
supports_objective_function(
    BT::Type{<:MOI.Bridges.Objective.AbstractBridge},
    F::Type{<:MOI.AbstractFunction},
)::Bool
```

Return a Bool indicating whether the bridges of type BT support bridging objective functions of type F.

Implementation notes

- This method depends only on the type of the inputs, not the runtime values.

- There is a default fallback, so you need only implement this method For objective functions that the bridge implements.

MathOptInterface.Bridges.set_objective_function_type – Function.

```
set_objective_function_type(
    BT::Type{<:Objective.AbstractBridge},
)::Type{<:MOI.AbstractFunction}
```

Return the type of objective function that bridges of concrete type BT set.

Implementation notes

- This method depends only on the type of the bridge, not the runtime value.

Example

```
julia> MOI.Bridges.set_objective_function_type(
           MOI.Bridges.Objective.FunctionizeBridge{Float64},
       )
MathOptInterface.ScalarAffineFunction{Float64}
```

MathOptInterface.Bridges.Objective.concrete_bridge_type – Function.

```
concrete_bridge_type(
    BT::Type{<:MOI.Bridges.Objective.AbstractBridge},
    F::Type{<:MOI.AbstractFunction},
)::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.AbstractFunction,
)::BT
```

Bridge the objective function func using bridge BT to model and returns a bridge object of type BT.

Implementation notes

- The bridge type BT must be a concrete type, that is, all the type parameters of the bridge must be set.

MathOptInterface.Bridges.Objective.SingleBridgeOptimizer – Type.

```
SingleBridgeOptimizer{BT<:AbstractBridge}(model::MOI.ModelLike)
```

Return `AbstractBridgeOptimizer` that always bridges any objective function supported by the bridge BT.

This is in contrast with the MOI.Bridges.LazyBridgeOptimizer, which only bridges the objective function if it is supported by the bridge BT and unsupported by model.

Example

```julia
julia> struct MyNewBridge{T} <: MOI.Bridges.Objective.AbstractBridge end

julia> bridge = MOI.Bridges.Objective.SingleBridgeOptimizer{MyNewBridge{Float64}}(
           MOI.Utilities.Model{Float64}(),
       )
MOIB.Objective.SingleBridgeOptimizer{MyNewBridge{Float64}, MOIU.Model{Float64}}
with 0 objective bridges
with inner model MOIU.Model{Float64}
```

Implementation notes

All bridges should simplify the creation of `SingleBridgeOptimizers` by defining a constant that wraps the bridge in a `SingleBridgeOptimizer`.

```julia
julia> const MyNewBridgeModel{T,OT<:MOI.ModelLike} =
           MOI.Bridges.Objective.SingleBridgeOptimizer{MyNewBridge{T},OT};
```

This enables users to create bridged models as follows:

```julia
julia> MyNewBridgeModel{Float64}(MOI.Utilities.Model{Float64}())
MOIB.Objective.SingleBridgeOptimizer{MyNewBridge{Float64}, MOIU.Model{Float64}}
with 0 objective bridges
with inner model MOIU.Model{Float64}
```

MathOptInterface.Bridges.Objective.add_all_bridges – Function.

```
add_all_bridges(model, ::Type{T}) where {T}
```

Add all bridges defined in the `Bridges.Objective` submodule to `model`.

The coefficient type used is T.

Variable bridge API

MathOptInterface.Bridges.Variable.AbstractBridge – Type.

```
abstract type AbstractBridge <: MOI.Bridges.AbstractBridge end
```

Subtype of `MOI.Bridges.AbstractBridge` for variable bridges.

In addition to the required implementation described in `MOI.Bridges.AbstractBridge`, subtypes of `AbstractBridge` must additionally implement:

- `supports_constrained_variable`
- `concrete_bridge_type`
- `bridge_constrained_variable`

MathOptInterface.Bridges.Variable.supports_constrained_variable – Function.

```
supports_constrained_variable(
    BT::Type{<:AbstractBridge},
    S::Type{<:MOI.AbstractSet},
)::Bool
```

Return a Bool indicating whether the bridges of type BT support bridging constrained variables in S. That is, it returns `true` if the bridge of type BT converts constrained variables of type S into a form supported by the solver.

Implementation notes

- This method depends only on the type of the bridge and set, not the runtime values.
- There is a default fallback, so you need only implement this method for sets that the bridge implements.

Example

```
julia> MOI.Bridges.Variable.supports_constrained_variable(
           MOI.Bridges.Variable.NonposToNonnegBridge{Float64},
           MOI.Nonpositives,
       )
true

julia> MOI.Bridges.Variable.supports_constrained_variable(
           MOI.Bridges.Variable.NonposToNonnegBridge{Float64},
           MOI.Nonnegatives,
       )
false
```

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 MOI.GreaterThan is bridged into variables in MOI.Nonnegatives by the VectorizeBridge:

```julia
julia> MOI.Bridges.Variable.concrete_bridge_type(
           MOI.Bridges.Variable.VectorizeBridge{Float64},
           MOI.GreaterThan{Float64},
       )
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,
)::BT
```

Bridge the constrained variable in set using bridge BT to model and returns a bridge object of type BT.

Implementation notes

- The bridge type BT must be a concrete type, that is, all the type parameters of the bridge must be set.

MathOptInterface.Bridges.Variable.SingleBridgeOptimizer – Type.

```
SingleBridgeOptimizer{BT<:AbstractBridge}(model::MOI.ModelLike)
```

Return MOI.Bridges.AbstractBridgeOptimizer that always bridges any variables constrained on creation supported by the bridge BT.

This is in contrast with the MOI.Bridges.LazyBridgeOptimizer, which only bridges the variables constrained on creation if they are supported by the bridge BT and unsupported by model.

> **Warning**
>
> Two SingleBridgeOptimizers cannot be used together as both of them assume that the underlying model only returns variable indices with nonnegative values. Use MOI.Bridges.LazyBridgeOptimizer instead.

Example

```julia
julia> struct MyNewBridge{T} <: MOI.Bridges.Variable.AbstractBridge end

julia> bridge = MOI.Bridges.Variable.SingleBridgeOptimizer{MyNewBridge{Float64}}(
           MOI.Utilities.Model{Float64}(),
```

```
        )
MOIB.Variable.SingleBridgeOptimizer{MyNewBridge{Float64}, MOIU.Model{Float64}}
with 0 variable bridges
with inner model MOIU.Model{Float64}
```

Implementation notes

All bridges should simplify the creation of `SingleBridgeOptimizers` by defining a constant that wraps the bridge in a `SingleBridgeOptimizer`.

```
julia> const MyNewBridgeModel{T,OT<:MOI.ModelLike} =
           MOI.Bridges.Variable.SingleBridgeOptimizer{MyNewBridge{T},OT};
```

This enables users to create bridged models as follows:

```
julia> MyNewBridgeModel{Float64}(MOI.Utilities.Model{Float64}())
MOIB.Variable.SingleBridgeOptimizer{MyNewBridge{Float64}, MOIU.Model{Float64}}
with 0 variable bridges
with inner model MOIU.Model{Float64}
```

MathOptInterface.Bridges.Variable.add_all_bridges – Function.

```
add_all_bridges(model, ::Type{T}) where {T}
```

Add all bridges defined in the `Bridges.Variable` submodule to `model`.

The coefficient type used is T.

MathOptInterface.Bridges.Variable.FlipSignBridge – Type.

```
abstract type FlipSignBridge{T,S1,S2} <: SetMapBridge{T,S2,S1} end
```

An abstract type that simplifies the creation of other bridges.

MathOptInterface.Bridges.Variable.SetMapBridge – Type.

```
abstract type SetMapBridge{T,S1,S2} <: AbstractBridge end
```

Consider two type of sets, S1 and S2, and a linear mapping A such 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:

- MOI.Bridges.map_set

- MOI.Bridges.map_function

Implementing a method for these two functions is sufficient to bridge constrained variables. However, in order for the getters and setters of attributes such as dual solutions and starting values to work as well, a method for the following functions must be implemented:

- MOI.Bridges.inverse_map_set
- MOI.Bridges.inverse_map_function
- MOI.Bridges.adjoint_map_function
- MOI.Bridges.inverse_adjoint_map_function.

See the docstrings of each function to see which feature would be missing if it was not implemented for a given bridge.

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::MOI.Bridges.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 ith 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.

AbstractBridgeOptimizer API

MathOptInterface.Bridges.AbstractBridgeOptimizer – Type.

```
abstract type AbstractBridgeOptimizer <: MOI.AbstractOptimizer end
```

An abstract type that implements generic functions for bridges.

Implementation notes

By convention, the inner optimizer should be stored in a model field. If not, the optimizer must implement
MOI.optimize!.

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.supports_constraint_bridges – Function.

```
supports_constraint_bridges(b::AbstractBridgeOptimizer)::Bool
```

Return a Bool indicating if b supports MOI.Bridges.Constraint.AbstractBridge.

MathOptInterface.Bridges.recursive_model – Function.

```
recursive_model(b::AbstractBridgeOptimizer)
```

If a variable, constraint, or objective is bridged, return the context of the inner variables. For most optimizers, this should be b.model.

LazyBridgeOptimizer API

MathOptInterface.Bridges.LazyBridgeOptimizer – Type.

```
LazyBridgeOptimizer(model::MOI.ModelLike)
```

The LazyBridgeOptimizer is a bridge optimizer that supports multiple bridges, and only bridges things which are not supported by the internal model.

Internally, the LazyBridgeOptimizer solves a shortest hyper-path problem to determine which bridges to use.

In general, you should use full_bridge_optimizer instead of this constructor because full_bridge_optimizer automatically adds a large number of supported bridges.

See also: add_bridge, remove_bridge, has_bridge and full_bridge_optimizer.

Example

```
julia> model = MOI.Bridges.LazyBridgeOptimizer(MOI.Utilities.Model{Float64}())
MOIB.LazyBridgeOptimizer{MOIU.Model{Float64}}
with 0 variable bridges
with 0 constraint bridges
with 0 objective bridges
with inner model MOIU.Model{Float64}

julia> MOI.Bridges.add_bridge(model, MOI.Bridges.Variable.FreeBridge{Float64})

julia> MOI.Bridges.has_bridge(model, MOI.Bridges.Variable.FreeBridge{Float64})
true
```

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, as well as the bridges in the list returned by the ListOfNonstandardBridges attribute.

Example

```
julia> model = MOI.Utilities.Model{Float64}();

julia> bridged_model = MOI.Bridges.full_bridge_optimizer(model, Float64);
```

Exceptions

The following bridges are not added by `full_bridge_optimizer`, except if they are in the list returned by the `ListOfNonstandardBridges` attribute:

- `Constraint.SOCtoNonConvexQuadBridge`
- `Constraint.RSOCtoNonConvexQuadBridge](@ref)`
- `Constraint.SOCtoPSDBridge`
- If T is not a subtype of `AbstractFloat`, subtypes of `Constraint.AbstractToIntervalBridge`
    - `Constraint.GreaterToIntervalBridge`
    - `Constraint.LessToIntervalBridge`)

See the docstring of the each bridge for the reason they are not added.

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 `MOI.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:

```julia
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.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.print_active_bridges – Function.

```
print_active_bridges([io::IO=stdout,] b::MOI.Bridges.LazyBridgeOptimizer)
```

Print the set of bridges that are active in the model b.

```
print_active_bridges(
    [io::IO=stdout,]
    b::MOI.Bridges.LazyBridgeOptimizer,
    F::Type{<:MOI.AbstractFunction}
)
```

Print the set of bridges required for an objective function of type F.

```
print_active_bridges(
    [io::IO=stdout,]
    b::MOI.Bridges.LazyBridgeOptimizer,
    F::Type{<:MOI.AbstractFunction},
    S::Type{<:MOI.AbstractSet},
)
```

Print the set of bridges required for a constraint of type F-in-S.

```
print_active_bridges(
    [io::IO=stdout,]
    b::MOI.Bridges.LazyBridgeOptimizer,
    S::Type{<:MOI.AbstractSet}
)
```

Print the set of bridges required for a variable constrained to set S.

MathOptInterface.Bridges.print_graph – Function.

```
print_graph([io::IO = stdout,] b::LazyBridgeOptimizer)
```

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 by all the bridges added to b.

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.

To see which nodes are used, call print_active_bridges.

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

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.

SetMap API

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 MOI.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 MOI.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 MOI.ConstraintPrimal of variable bridges. For constraint bridges, this is used for bridging the constraint, setting the MOI.ConstraintFunction and MOI.ConstraintPrimalStart and modifying the function with MOI.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 MOI.VariablePrimal and MOI.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 MOI.VariablePrimalStart of variable bridges and for getting the MOI.ConstraintFunction, the MOI.ConstraintPrimal and the MOI.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 MOI.ConstraintDual and MOI.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 MOI.ConstraintDual of variable bridges and setting the MOI.ConstraintDualStart of constraint bridges.

Bridging graph API

MathOptInterface.Bridges.Graph – Type.

```
Graph()
```

A type-stable datastructure for computing the shortest hyperpath problem.

Nodes

There are three types of nodes in the graph:

- VariableNode
- ConstraintNode
- ObjectiveNode

Add nodes to the graph using add_node.

Edges

There are two types of edges in the graph:

- Edge
- ObjectiveEdge

Add edges to the graph using add_edge.

For the ability to add a variable constrained on creation as a free variable followed by a constraint, use set_variable_constraint_node.

Optimal hyper-edges

Use bridge_index to compute the minimum-cost bridge leaving a node.

Note that bridge_index lazy runs a Bellman-Ford algorithm to compute the set of minimum cost edges. Thus, the first call to bridge_index after adding new nodes or edges will take longer than subsequent calls.

MathOptInterface.Bridges.VariableNode – Type.

```
VariableNode(index::Int)
```

A node in Graph representing a variable constrained on creation.

MathOptInterface.Bridges.ConstraintNode – Type.

```
ConstraintNode(index::Int)
```

A node in Graph representing a constraint.

MathOptInterface.Bridges.ObjectiveNode – Type.

```
ObjectiveNode(index::Int)
```

A node in Graph representing an objective function.

MathOptInterface.Bridges.Edge – Type.

```
Edge(
    bridge_index::Int,
    added_variables::Vector{VariableNode},
    added_constraints::Vector{ConstraintNode},
)
```

Return a new datastructure representing an edge in Graph that starts at a VariableNode or a ConstraintNode.

MathOptInterface.Bridges.ObjectiveEdge – Type.

```
ObjectiveEdge(
    bridge_index::Int,
    added_variables::Vector{VariableNode},
    added_constraints::Vector{ConstraintNode},
)
```

Return a new datastructure representing an edge in Graph that starts at an ObjectiveNode.

MathOptInterface.Bridges.add_node – Function.

```
add_node(graph::Graph, ::Type{VariableNode})::VariableNode
add_node(graph::Graph, ::Type{ConstraintNode})::ConstraintNode
add_node(graph::Graph, ::Type{ObjectiveNode})::ObjectiveNode
```

Add a new node to graph.

MathOptInterface.Bridges.add_edge – Function.

```
add_edge(graph::Graph, node::VariableNode, edge::Edge)::Nothing
add_edge(graph::Graph, node::ConstraintNode, edge::Edge)::Nothing
add_edge(graph::Graph, node::ObjectiveNode, edge::ObjectiveEdge)::Nothing
```

Add edge to graph, where edge starts at node and connects to the nodes defined in edge.

MathOptInterface.Bridges.set_variable_constraint_node – Function.

```
set_variable_constraint_node(
    graph::Graph,
    variable_node::VariableNode,
    constraint_node::ConstraintNode,
    cost::Int,
)
```

As an alternative to variable_node, add a virtual edge to graph that represents adding a free variable, followed by a constraint of type constraint_node, with bridging cost cost.

Why is this needed?

Variables can either be added as a variable constrained on creation, or as a free variable which then has a constraint added to it.

MathOptInterface.Bridges.bridge_index – Function.

```
bridge_index(graph::Graph, node::VariableNode)::Int
bridge_index(graph::Graph, node::ConstraintNode)::Int
bridge_index(graph::Graph, node::ObjectiveNode)::Int
```

Return the optimal index of the bridge to chose from node.

MathOptInterface.Bridges.is_variable_edge_best – Function.

```
is_variable_edge_best(graph::Graph, node::VariableNode)::Bool
```

Return a Bool indicating whether node should be added as a variable constrained on creation, or as a free variable followed by a constraint.

42.3 FileFormats

Overview

The FileFormats submodule

The FileFormats module provides functions 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. Specific 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
julia> model = MOI.FileFormats.Model(format = MOI.FileFormats.FORMAT_NL)
An AMPL (.nl) model
```

The REW file format

```julia
julia> model = MOI.FileFormats.Model(format = MOI.FileFormats.FORMAT_REW)
A Mathematical Programming System (MPS) model
```

Note that the REW format is identical to the MPS file format, except that all names are replaced with generic identifiers.

The SDPA file format

```julia
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
julia> src = MOI.Utilities.Model{Float64}()
MOIU.Model{Float64}

julia> MOI.add_variable(src)
MOI.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:
  MOI.VariableIndex(1) => MOI.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": 4
  },
  "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}:
 MOI.VariableIndex(1)

julia> rm("file.mof.json")  # Clean up after ourselves.
```

Detecting the file-type 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 (for example, the CBF format supports integer variables but not binary). If so, 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")
```

> **Note**
>
> Even after bridging, it may still not be possible to write the model to file because of unsupported constraints (for example, 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
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
julia> good_model = JSON.parse("""
       {
         "version": {
           "major": 1,
           "minor": 4
         },
         "variables": [{"name": "x"}],
         "objective": {"sense": "feasibility"},
         "constraints": []
       }
       """);

julia> isvalid(schema, good_model)
true
```

If we construct an invalid file, for example by mis-typing name as NaMe, the validation fails:

```julia
julia> bad_model = JSON.parse("""
       {
         "version": {
           "major": 1,
           "minor": 4
         },
         "variables": [{"NaMe": "x"}],
         "objective": {"sense": "feasibility"},
         "constraints": []
       }
       """);

julia> isvalid(schema, bad_model)
false
```

Use `JSONSchema.validate` to obtain more insight into why the validation failed:

```julia
julia> JSONSchema.validate(schema, bad_model)
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_REW: the .rew file format, which is MPS with generic names
- FORMAT_SDPA: the SemiDefinite Programming Algorithm format

MathOptInterface.FileFormats.CBF.Model – Type.

```
Model()
```

Create an empty instance of FileFormats.CBF.Model.

MathOptInterface.FileFormats.LP.Model – Type.

```
Model(; kwargs...)
```

Create an empty instance of FileFormats.LP.Model.

Keyword arguments are:

- `maximum_length::Int=255`: the maximum length for the name of a variable. lp_solve 5.0 allows only 16 characters, while CPLEX 12.5+ allow 255.
- `warn::Bool=false`: print a warning when variables or constraints are renamed.

`MathOptInterface.FileFormats.MOF.Model` – Type.

```
Model(; kwargs...)
```

Create an empty instance of FileFormats.MOF.Model.

Keyword arguments are:

- `print_compact::Bool=false`: print the JSON file in a compact format without spaces or newlines.
- `warn::Bool=false`: print a warning when variables or constraints are renamed
- `differentiation_backend::MOI.Nonlinear.AbstractAutomaticDifferentiation = MOI.Nonlinear.SparseRever` automatic differentiation backend to use when reading models with nonlinear constraints and objectives.

`MathOptInterface.FileFormats.MPS.Model` – Type.

```
Model(; kwargs...)
```

Create an empty instance of FileFormats.MPS.Model.

Keyword arguments are:

- `warn::Bool=false`: print a warning when variables or constraints are renamed.
- `print_objsense::Bool=false`: print the OBJSENSE section when writing
- `generic_names::Bool=false`: strip all names in the model and replace them with the generic names C$i and R$i for the i'th column and row respectively.
- `quadratic_format::QuadraticFormat = kQuadraticFormatGurobi`: specify the solver-specific extension used when writing the quadratic components of the model. Options are kQuadraticFormatGurobi, kQuadraticFormatCPLEX, and kQuadraticFormatMosek.

`MathOptInterface.FileFormats.NL.Model` – Type.

```
Model()
```

Create a new Optimizer object.

`MathOptInterface.FileFormats.SDPA.Model` – Type.

```
Model(; number_type::Type = Float64)
```

Create an empty instance of `FileFormats.SDPA.Model{number_type}`.

It is important to be aware that the SDPA file format is interpreted in geometric form and not standard conic form. The standard conic form and geometric conic form are two dual standard forms for semidefinite programs (SDPs). The geometric conic form of an SDP is as follows:

$$\min_{y \in \mathbb{R}^m} \qquad\qquad\qquad b^T y \tag{42.1}$$

$$\text{s.t.} \qquad\qquad\qquad \sum_{i=1}^{m} A_i y_i - C \in \mathbb{K} \tag{42.2}$$

where $\mathcal{K}$ is a cartesian product of nonnegative orthant and positive semidefinite matrices that align with a block diagonal structure shared with the matrices `A_i` and `C`.

In other words, the geometric conic form contains free variables and affine constraints in either the nonnegative orthant or the positive semidefinite cone. That is, in the MathOptInterface's terminology, `MOI.VectorAffineFunction`-in-`MOI.Nonnegatives` and `MOI.VectorAffineFunction`-in-`MOI.PositiveSemidefiniteConeTriangle` constraints.

The corresponding standard conic form of the dual SDP is as follows:

$$\max_{X \in \mathbb{K}} \qquad\qquad \text{tr}(CX) \tag{42.3}$$

$$\text{s.t.} \qquad\qquad \text{tr}(A_i X) = b_i \qquad\qquad i = 1, \ldots, m. \tag{42.4}$$

In other words, the standard conic form contains nonnegative and positive semidefinite variables with equality constraints. That is, in the MathOptInterface's terminology, `MOI.VectorOfVariables`-in-`MOI.Nonnegatives`, `MOI.VectorOfVariables`-in-`MOI.PositiveSemidefiniteConeTriangle` and `MOI.ScalarAffineFunction`-in-`MOI.EqualTo` constraints.

If a model is in standard conic form, use `Dualization.jl` to transform it into the geometric conic form before writting it. Otherwise, the nonnegative (resp. positive semidefinite) variables will be bridged into free variables with affine constraints constraining them to belong to the nonnegative orthant (resp. positive semidefinite cone) by the `MOI.Bridges.Constraint.VectorFunctionizeBridge`. Moreover, equality constraints will be bridged into pairs of affine constraints in the nonnegative orthant by the `MOI.Bridges.Constraint.SplitInter` and then the `MOI.Bridges.Constraint.VectorizeBridge`.

If a solver is in standard conic form, use `Dualization.jl` to transform the model read into standard conic form before copying it to the solver. Otherwise, the free variables will be bridged into pairs of variables in the nonnegative orthant by the `MOI.Bridges.Variable.FreeBridge` and affine constraints will be bridged into equality constraints by creating a slack variable by the `MOI.Bridges.Constraint.VectorSlackBridge`.

Other helpers

`MathOptInterface.FileFormats.NL.SolFileResults` – Type.

```
SolFileResults(filename::String, model::Model)
```

Parse the `.sol` file `filename` created by solving `model` and return a `SolFileResults` struct.

The returned struct supports the `MOI.get` API for querying result attributes such as `MOI.TerminationStatus`, `MOI.VariablePrimal`, and `MOI.ConstraintDual`.

```
SolFileResults(
    raw_status::String,
    termination_status::MOI.TerminationStatusCode,
)
```

Return a SolFileResults struct with MOI.RawStatusString set to raw_status, MOI.TerminationStatus set to termination_status, and MOI.PrimalStatus and MOI.DualStatus set to NO_SOLUTION.

All other attributes are un-set.

42.4 Nonlinear

Overview

Nonlinear

> **Warning**
>
> The Nonlinear submodule is experimental. Until this message is removed, breaking changes may be introduced in any minor or patch release of MathOptInterface.

The Nonlinear submodule contains data structures and functions for working with a nonlinear optimization problem in the form of an expression graph. This page explains the API and describes the rationale behind its design.

Standard form

Nonlinear programs (NLPs) are a class of optimization problems in which some of the constraints or the objective function are nonlinear:

$$\min_{x \in \mathbb{R}^n} f_0(x) \tag{42.5}$$

$$\text{s.t.} l_j \leq f_j(x) \leq u_j \qquad\qquad j = 1 \ldots m \tag{42.6}$$

There may be additional constraints, as well as things like variable bounds and integrality restrictions, but we do not consider them here because they are best dealt with by other components of MathOptInterface.

API overview

The core element of the Nonlinear submodule is Nonlinear.Model:

```julia
julia> const Nonlinear = MOI.Nonlinear;

julia> model = Nonlinear.Model()
A Nonlinear.Model with:
 0 objectives
 0 parameters
 0 expressions
 0 constraints
```

Nonlinear.Model is a mutable struct that stores all of the nonlinear information added to the model.

Decision variables

Decision variables are represented by VariableIndexes. The user is responsible for creating these using MOI.VariableIndex(i), where i is the column associated with the variable.

Expressions

The input data structure is a Julia Expr. The input expressions can incorporate VariableIndexes, but these must be interpolated into the expression with $:

```julia
julia> x = MOI.VariableIndex(1)
MOI.VariableIndex(1)

julia> input = :(1 + sin($x)^2)
:(1 + sin(MathOptInterface.VariableIndex(1)) ^ 2)
```

There are a number of restrictions on the input Expr:

- It cannot contain macros

- It cannot contain broadcasting

- It cannot contain splatting (except in limited situations)

- It cannot contain linear algebra, such as matrix-vector products

- It cannot contain generator expressions, including sum(i for i in S)

Given an input expression, add an expression using Nonlinear.add_expression:

```julia
julia> expr = Nonlinear.add_expression(model, input)
MathOptInterface.Nonlinear.ExpressionIndex(1)
```

The return value, expr, is a Nonlinear.ExpressionIndex that can then be interpolated into other input expressions.

Looking again at model, we see:

```julia
julia> model
A Nonlinear.Model with:
 0 objectives
 0 parameters
 1 expression
 0 constraints
```

Parameters

In addition to constant literals like 1 or 1.23, you can create parameters. Parameters are placeholders whose values can change before passing the expression to the solver. Create a parameter using Nonlinear.add_parameter, which accepts a default value:

```
julia> p = Nonlinear.add_parameter(model, 1.23)
MathOptInterface.Nonlinear.ParameterIndex(1)
```

The return value, p, is a `Nonlinear.ParameterIndex` that can then be interpolated into other input expressions.

Looking again at model, we see:

```
julia> model
A Nonlinear.Model with:
 0 objectives
 1 parameter
 1 expression
 0 constraints
```

Update a parameter as follows:

```
julia> model[p]
1.23

julia> model[p] = 4.56
4.56

julia> model[p]
4.56
```

Objectives

Set a nonlinear objective using `Nonlinear.set_objective`:

```
julia> Nonlinear.set_objective(model, :($p + $expr + $x))

julia> model
A Nonlinear.Model with:
 1 objective
 1 parameter
 1 expression
 0 constraints
```

Clear a nonlinear objective by passing nothing:

```
julia> Nonlinear.set_objective(model, nothing)

julia> model
A Nonlinear.Model with:
 0 objectives
 1 parameter
 1 expression
 0 constraints
```

But we'll re-add the objective for later:

```
julia> Nonlinear.set_objective(model, :($p + $expr + $x));
```

Constraints

Add a constraint using Nonlinear.add_constraint:

```
julia> c = Nonlinear.add_constraint(model, :(1 + sqrt($x)), MOI.LessThan(2.0))
MathOptInterface.Nonlinear.ConstraintIndex(1)

julia> model
A Nonlinear.Model with:
 1 objective
 1 parameter
 1 expression
 1 constraint
```

The return value, c, is a Nonlinear.ConstraintIndex that is a unique identifier for the constraint. Interval constraints are also supported:

```
julia> c2 = Nonlinear.add_constraint(model, :(1 + sqrt($x)), MOI.Interval(-1.0, 2.0))
MathOptInterface.Nonlinear.ConstraintIndex(2)

julia> model
A Nonlinear.Model with:
 1 objective
 1 parameter
 1 expression
 2 constraints
```

Delete a constraint using Nonlinear.delete:

```
julia> Nonlinear.delete(model, c2)

julia> model
A Nonlinear.Model with:
 1 objective
 1 parameter
 1 expression
 1 constraint
```

User-defined operators

By default, Nonlinear supports a wide range of univariate and multivariate operators. However, you can also define your own operators by registering them.

Univariate operators

Register a univariate user-defined operator using Nonlinear.register_operator:

```julia
julia> f(x) = 1 + sin(x)^2
f (generic function with 1 method)

julia> Nonlinear.register_operator(model, :my_f, 1, f)
```

Now, you can use :my_f in expressions:

```julia
julia> new_expr = Nonlinear.add_expression(model, :(my_f($x + 1)))
MathOptInterface.Nonlinear.ExpressionIndex(2)
```

By default, Nonlinear will compute first- and second-derivatives of the registered operator using ForwardDiff.jl.
Override this by passing functions which compute the respective derivative:

```julia
julia> f'(x) = 2 * sin(x) * cos(x)
f′ (generic function with 1 method)

julia> Nonlinear.register_operator(model, :my_f2, 1, f, f′)
```

or

```julia
julia> f''(x) = 2 * (cos(x)^2 - sin(x)^2)
f′′ (generic function with 1 method)

julia> Nonlinear.register_operator(model, :my_f3, 1, f, f′, f′′)
```

Multivariate operators

Register a multivariate user-defined operator using Nonlinear.register_operator:

```julia
julia> g(x...) = x[1]^2 + x[1] * x[2] + x[2]^2
g (generic function with 1 method)

julia> Nonlinear.register_operator(model, :my_g, 2, g)
```

Now, you can use :my_g in expressions:

```julia
julia> new_expr = Nonlinear.add_expression(model, :(my_g($x + 1, $x)))
MathOptInterface.Nonlinear.ExpressionIndex(3)
```

By default, Nonlinear will compute the gradient of the registered operator using ForwardDiff.jl. (Hessian
information is not supported.) Override this by passing a function to compute the gradient:

```julia
julia> function ∇g(ret, x...)
           ret[1] = 2 * x[1] + x[2]
           ret[2] = x[1] + 2 * x[2]
           return
       end
∇g (generic function with 1 method)

julia> Nonlinear.register_operator(model, :my_g2, 2, g, ∇g)
```

MathOptInterface

MathOptInterface communicates the nonlinear portion of an optimization problem to solvers using concrete subtypes of AbstractNLPEvaluator, which implement the Nonlinear programming API.

Create an AbstractNLPEvaluator from Nonlinear.Model using Nonlinear.Evaluator.

Nonlinear.Evaluator requires an Nonlinear.AbstractAutomaticDifferentiation backend and an ordered list of the variables that are included in the model.

There following backends are available to choose from within MOI, although other packages may add more options by sub-typing Nonlinear.AbstractAutomaticDifferentiation:

- Nonlinear.ExprGraphOnly

- Nonlinear.SparseReverseMode.

```julia
julia> evaluator = Nonlinear.Evaluator(model, Nonlinear.ExprGraphOnly(), [x])
Nonlinear.Evaluator with available features:
  * :ExprGraph
```

The functions of the Nonlinear programming API implemented by Nonlinear.Evaluator depends upon the chosen Nonlinear.AbstractAutomaticDifferentiation backend.

The :ExprGraph feature means we can call objective_expr and constraint_expr to retrieve the expression graph of the problem. However, we cannot call gradient terms such as eval_objective_gradient because Nonlinear.ExprGraphOnly does not have the capability to differentiate a nonlinear expression.

If, instead, we pass Nonlinear.SparseReverseMode, then we get access to :Grad, the gradient of the objective function, :Jac, the Jacobian matrix of the constraints, :JacVec, the ability to compute Jacobian-vector products, and :ExprGraph.

```julia
julia> evaluator = Nonlinear.Evaluator(
           model,
           Nonlinear.SparseReverseMode(),
           [x],
       )
Nonlinear.Evaluator with available features:
  * :Grad
  * :Jac
  * :JacVec
  * :ExprGraph
```

However, before using the evaluator, we need to call initialize:

```julia
julia> MOI.initialize(evaluator, [:Grad, :Jac, :JacVec, :ExprGraph])
```

Now we can call methods like eval_objective:

```julia
julia> x = [1.0]
1-element Vector{Float64}:
 1.0
```

```
julia> MOI.eval_objective(evaluator, x)
7.268073418273571
```

and eval_objective_gradient:

```
julia> grad = [0.0]
1-element Vector{Float64}:
 0.0

julia> MOI.eval_objective_gradient(evaluator, grad, x)

julia> grad
1-element Vector{Float64}:
 1.909297426825682
```

Instead of passing Nonlinear.Evaluator directly to solvers, solvers query the NLPBlock attribute, which returns an NLPBlockData. This object wraps an Nonlinear.Evaluator and includes other information such as constraint bounds and whether the evaluator has a nonlinear objective. Create and set NLPBlockData as follows:

```
julia> block = MOI.NLPBlockData(evaluator);

julia> model = MOI.Utilities.UniversalFallback(MOI.Utilities.Model{Float64}());

julia> MOI.set(model, MOI.NLPBlock(), block);
```

> **Warning**
>
> Only call NLPBlockData once you have finished modifying the problem in model.

Putting everything together, you can create a nonlinear optimization problem in MathOptInterface as follows:

```
import MathOptInterface as MOI

function build_model(
    model::MOI.ModelLike;
    backend::MOI.Nonlinear.AbstractAutomaticDifferentiation,
)
    x = MOI.add_variable(model)
    y = MOI.add_variable(model)
    MOI.set(model, MOI.ObjectiveSense(), MOI.MIN_SENSE)
    nl_model = MOI.Nonlinear.Model()
    MOI.Nonlinear.set_objective(nl_model, :($x^2 + $y^2))
    evaluator = MOI.Nonlinear.Evaluator(nl_model, backend, [x, y])
    MOI.set(model, MOI.NLPBlock(), MOI.NLPBlockData(evaluator))
    return
end

# Replace `model` and `backend` with your optimizer and backend of choice.
```

```
model = MOI.Utilities.UniversalFallback(MOI.Utilities.Model{Float64}())
build_model(model; backend = MOI.Nonlinear.SparseReverseMode())
```

Expression-graph representation

Nonlinear.Model stores nonlinear expressions in Nonlinear.Expressions. This section explains the design of the expression graph data structure in Nonlinear.Expression.

Given a nonlinear function like f(x) = sin(x)^2 + x, a conceptual aid for thinking about the graph representation of the expression is to convert it into Polish prefix notation:

```
f(x, y) = (+ (^ (sin x) 2) x)
```

This format identifies each operator (function), as well as a list of arguments. Operators can be univariate, like sin, or multivariate, like +.

A common way of representing Polish prefix notation in code is as follows:

```julia
julia> x = MOI.VariableIndex(1);

julia> struct ExprNode
           op::Symbol
           children::Vector{Union{ExprNode,Float64,MOI.VariableIndex}}
       end

julia> expr = ExprNode(:+, [ExprNode(:^, [ExprNode(:sin, [x]), 2.0]), x]);
```

This data structure follows our Polish prefix notation very closely, and we can easily identify the arguments to an operator. However, it has a significant draw-back: each node in the graph requires a Vector, which is heap-allocated and tracked by Julia's garbage collector (GC). For large models, we can expect to have millions of nodes in the expression graph, so this overhead quickly becomes prohibitive for computation.

An alternative is to record the expression as a linear tape:

```julia
julia> expr = Any[:+, 2, :^, 2, :sin, 1, x, 2.0, x]
9-element Vector{Any}:
  :+
 2
  :^
 2
  :sin
 1
  MOI.VariableIndex(1)
 2.0
  MOI.VariableIndex(1)
```

The Int after each operator Symbol specifies the number of arguments.

This data-structure is a single vector, which resolves our problem with the GC, but each element is the abstract type, Any, and so any operations on it will lead to slower dynamic dispatch. It's also hard to identify the children of each operation without reading the entire tape.

To summarize, representing expression graphs in Julia has the following challenges:

- Nodes in the expression graph should not contain a heap-allocated object

- All data-structures should be concretely typed

- It should be easy to identify the children of a node

Sketch of the design in Nonlinear

`Nonlinear` overcomes these problems by decomposing the data structure into a number of different concrete-typed vectors.

First, we create vectors of the supported uni- and multivariate operators.

```julia
julia> const UNIVARIATE_OPERATORS = [:sin];

julia> const MULTIVARIATE_OPERATORS = [:+, :^];
```

In practice, there are many more supported operations than the ones listed here.

Second, we create an enum to represent the different types of nodes present in the expression graph:

```julia
julia> @enum(
           NodeType,
           NODE_CALL_MULTIVARIATE,
           NODE_CALL_UNIVARIATE,
           NODE_VARIABLE,
           NODE_VALUE,
       )
```

In practice, there are node types other than the ones listed here.

Third, we create two concretely typed structs as follows:

```julia
julia> struct Node
           type::NodeType
           parent::Int
           index::Int
       end

julia> struct Expression
           nodes::Vector{Node}
           values::Vector{Float64}
       end
```

For each node node in the `.nodes` field, if `node.type` is:

- `NODE_CALL_MULTIVARIATE`, we look up `MULTIVARIATE_OPERATORS[node.index]` to retrieve the operator

- `NODE_CALL_UNIVARIATE`, we look up `UNIVARIATE_OPERATORS[node.index]` to retrieve the operator

- `NODE_VARIABLE`, we create `MOI.VariableIndex(node.index)`

- `NODE_VALUE`, we look up `values[node.index]`

The `.parent` field of each node is the integer index of the parent node in `.nodes`. For the first node, the parent is `-1` by convention.

Therefore, we can represent our function as:

```julia
julia> expr = Expression(
           [
               Node(NODE_CALL_MULTIVARIATE, -1, 1),
               Node(NODE_CALL_MULTIVARIATE, 1, 2),
               Node(NODE_CALL_UNIVARIATE, 2, 1),
               Node(NODE_VARIABLE, 3, 1),
               Node(NODE_VALUE, 2, 1),
               Node(NODE_VARIABLE, 1, 1),
           ],
           [2.0],
       );
```

This is less readable than the other options, but does this data structure meet our design goals?

Instead of a heap-allocated object for each node, we only have two `Vectors` for each expression, nodes and `values`, as well as two constant vectors for the `OPERATORS`. In addition, all fields are concretely typed, and there are no Union or Any types.

For our third goal, it is not easy to identify the children of a node, but it is easy to identify the parent of any node. Therefore, we can use `Nonlinear.adjacency_matrix` to compute a sparse matrix that maps parents to their children.

The tape is also ordered topologically, so that a reverse pass of the nodes evaluates all children nodes before their parent.

The design in practice

In practice, Node and `Expression` are exactly `Nonlinear.Node` and `Nonlinear.Expression`. However, `Nonlinear.NodeType` has more fields to account for comparison operators such as `:>=` and `:<=`, logic operators such as `:&&` and `:||`, nonlinear parameters, and nested subexpressions.

Moreover, instead of storing the operators as global constants, they are stored in `Nonlinear.OperatorRegistry`, and it also stores a vector of logic operators and a vector of comparison operators. In addition to `Nonlinear.DEFAULT_UNIVARIATE` and `Nonlinear.DEFAULT_MULTIVARIATE_OPERATORS`, you can register user-defined functions using `Nonlinear.register_operat`

`Nonlinear.Model` is a struct that stores the `Nonlinear.OperatorRegistry`, as well as a list of parameters and subexpressions in the model.

ReverseAD

`Nonlinear.ReverseAD` is a submodule for computing derivatives of a nonlinear optimization problem using sparse reverse-mode automatic differentiation (AD).

This section does not attempt to explain how sparse reverse-mode AD works, but instead explains why MOI contains its own implementation, and highlights notable differences from similar packages.

> **Warning**
>
> Don't use the API in ReverseAD to compute derivatives. Instead, create a `Nonlinear.Evaluator` object with `Nonlinear.SparseReverseMode` as the backend, and then query the MOI API methods.

Design goals

The JuliaDiff organization maintains a list of packages for doing AD in Julia. At last count, there were at least ten packages——not including `ReverseAD`——for reverse-mode AD in Julia. `ReverseAD` exists because it has a different set of design goals.

- **Goal: handle scale and sparsity.** The types of nonlinear optimization problems that MOI represents can be large scale (10^5 or more functions across 10^5 or more variables) with very sparse derivatives. The ability to compute a sparse Hessian matrix is essential. To the best of our knowledge, `ReverseAD` is the only reverse-mode AD system in Julia that handles sparsity by default.

- **Goal: limit the scope to improve robustness.** Most other AD packages accept arbitrary Julia functions as input and then trace an expression graph using operator overloading. This means they must deal (or detect and ignore) with control flow, I/O, and other vagaries of Julia. In contrast, `ReverseAD` only accepts functions in the form of `Nonlinear.Expression`, which greatly limits the range of syntax that it must deal with. By reducing the scope of what we accept as input to functions relevant for mathematical optimization, we can provide a simpler implementation with various performance optimizations.

- **Goal: provide outputs which match what solvers expect.** Other AD packages focus on differentiating individual Julia functions. In contrast, `ReverseAD` has a very specific use-case: to generate outputs needed by the MOI nonlinear API. This means it needs to efficiently compute sparse Hessians, and it needs subexpression handling to avoid recomputing subexpressions that are shared between functions.

History

ReverseAD started life as ReverseDiffSparse.jl, development of which began in early 2014(!). This was well before the other AD packages started development. Because we had a well-tested, working AD in JuMP, there was less motivation to contribute to and explore other AD packages. The lack of historical interaction also meant that other packages were not optimized for the types of problems that JuMP is built for (that is, large-scale sparse problems). When we first created MathOptInterface, we kept the AD in JuMP to simplify the transition, and post-poned the development of a first-class nonlinear interface in MathOptInterface.

Prior to the introduction of `Nonlinear`, JuMP's nonlinear implementation was a confusing mix of functions and types spread across the code base and in the private `_Derivatives` submodule. This made it hard to swap the AD system for another. The main motivation for refactoring JuMP to create the `Nonlinear` submodule in MathOptInterface was to abstract the interface between JuMP and the AD system, allowing us to swap-in and test new AD systems in the future.

API Reference

Nonlinear Modeling

More information can be found in the Nonlinear section of the manual.

`MathOptInterface.Nonlinear` – Module.

```
Nonlinear
```

> **Warning**
>
> The Nonlinear submodule is experimental. Until this message is removed, breaking changes may be introduced in any minor or patch release of MathOptInterface.

MathOptInterface.Nonlinear.Model – Type.

```
Model()
```

The core datastructure for representing a nonlinear optimization problem.

It has the following fields:

- `objective::Union{Nothing,Expression}` : holds the nonlinear objective function, if one exists, otherwise nothing.
- `expressions::Vector{Expression}` : a vector of expressions in the model.
- `constraints::OrderedDict{ConstraintIndex,Constraint}` : a map from ConstraintIndex to the corresponding Constraint. An `OrderedDict` is used instead of a `Vector` to support constraint deletion.
- `parameters::Vector{Float64}` : holds the current values of the parameters.
- `operators::OperatorRegistry` : stores the operators used in the model.

Expressions

MathOptInterface.Nonlinear.ExpressionIndex – Type.

```
ExpressionIndex
```

An index to a nonlinear expression that is returned by add_expression.

Given data::Model and ex::ExpressionIndex, use data[ex] to retrieve the corresponding Expression.

MathOptInterface.Nonlinear.add_expression – Function.

```
add_expression(model::Model, expr)::ExpressionIndex
```

Parse expr into a Expression and add to model. Returns an ExpressionIndex that can be interpolated into other input expressions.

expr must be a type that is supported by parse_expression.

Examples

```
model = Model()
x = MOI.VariableIndex(1)
ex = add_expression(model, :($x^2 + 1))
set_objective(model, :(sqrt($ex)))
```

Parameters

MathOptInterface.Nonlinear.ParameterIndex – Type.

```
ParameterIndex
```

An index to a nonlinear parameter that is returned by add_parameter. Given data::Model and p::ParameterIndex, use data[p] to retrieve the current value of the parameter and data[p] = value to set a new value.

MathOptInterface.Nonlinear.add_parameter – Function.

```
add_parameter(model::Model, value::Float64)::ParameterIndex
```

Add a new parameter to model with the default value value. Returns a ParameterIndex that can be interpolated into other input expressions and used to modify the value of the parameter.

Examples

```
model = Model()
x = MOI.VariableIndex(1)
p = add_parameter(model, 1.2)
c = add_constraint(model, :($x^2 - $p), MOI.LessThan(0.0))
```

Objectives

MathOptInterface.Nonlinear.set_objective – Function.

```
set_objective(model::Model, obj)::Nothing
```

Parse obj into a Expression and set as the objective function of model.

obj must be a type that is supported by parse_expression.

To remove the objective, pass nothing.

Examples

```
model = Model()
x = MOI.VariableIndex(1)
set_objective(model, :($x^2 + 1))
set_objective(model, x)
set_objective(model, nothing)
```

Constraints

MathOptInterface.Nonlinear.ConstraintIndex – Type.

```
ConstraintIndex
```

An index to a nonlinear constraint that is returned by add_constraint.

Given data::Model and c::ConstraintIndex, use data[c] to retrieve the corresponding Constraint.

MathOptInterface.Nonlinear.add_constraint – Function.

```
add_constraint(
    model::Model,
    func,
    set::Union{
        MOI.GreaterThan{Float64},
        MOI.LessThan{Float64},
        MOI.Interval{Float64},
        MOI.EqualTo{Float64},
    },
)
```

Parse func and set into a Constraint and add to model. Returns a ConstraintIndex that can be used to delete the constraint or query solution information.

Examples

```
model = Model()
x = MOI.VariableIndex(1)
c = add_constraint(model, :($x^2), MOI.LessThan(1.0))
```

MathOptInterface.Nonlinear.delete – Function.

```
delete(model::Model, c::ConstraintIndex)::Nothing
```

Delete the constraint index c from model.

Examples

```
model = Model()
x = MOI.VariableIndex(1)
c = add_constraint(model, :($x^2), MOI.LessThan(1.0))
delete(model, c)
```

User-defined operators

MathOptInterface.Nonlinear.OperatorRegistry – Type.

```
OperatorRegistry()
```

Create a new OperatorRegistry to store and evaluate univariate and multivariate operators.

MathOptInterface.Nonlinear.DEFAULT_UNIVARIATE_OPERATORS – Constant.

```
DEFAULT_UNIVARIATE_OPERATORS
```

The list of univariate operators that are supported by default.

MathOptInterface.Nonlinear.DEFAULT_MULTIVARIATE_OPERATORS – Constant.

```
DEFAULT_MULTIVARIATE_OPERATORS
```

The list of multivariate operators that are supported by default.

MathOptInterface.Nonlinear.register_operator – Function.

```
register_operator(
    model::Model,
    op::Symbol,
    nargs::Int,
    f::Function,
    [∇f::Function],
    [∇²f::Function],
)
```

Register the user-defined operator op with nargs input arguments in model.

Univariate functions

- f(x::T)::T must be a function that takes a single input argument x and returns the function evaluated at x. If ∇f and ∇²f are not provided, f must support any Real input type T.
- ∇f(x::T)::T is a function that takes a single input argument x and returns the first derivative of f with respect to x. If ∇²f is not provided, ∇f must support any Real input type T.
- ∇²f(x::T)::T is a function that takes a single input argument x and returns the second derivative of f with respect to x.

Multivariate functions

- f(x::T...)::T must be a function that takes a nargs input arguments x and returns the function evaluated at x. If ∇f and ∇²f are not provided, f must support any Real input type T.
- ∇f(g::AbstractVector{T}, x::T...)::T is a function that takes a cache vector g of length length(x), and fills each element g[i] with the partial derivative of f with respect to x[i].
- ∇²f(H::AbstractMatrix, x::T...)::T is a function that takes a matrix H and fills the lower-triangular components H[i, j] with the Hessian of f with respect to x[i] and x[j] for i >= j.

Notes for multivariate Hessians

- H has size(H) == (length(x), length(x)), but you must not access elements H[i, j] for i > j.
- H is dense, but you do not need to fill structural zeros.

MathOptInterface.Nonlinear.register_operator_if_needed – Function.

```
register_operator_if_needed(
    registry::OperatorRegistry,
    op::Symbol,
    nargs::Int,
    f::Function;
)
```

Similar to register_operator, but this function warns if the function is not registered, and skips silently if it already is.

MathOptInterface.Nonlinear.assert_registered – Function.

```
assert_registered(registry::OperatorRegistry, op::Symbol, nargs::Int)
```

Throw an error if op is not registered in registry with nargs arguments.

MathOptInterface.Nonlinear.check_return_type – Function.

```
check_return_type(::Type{T}, ret::S) where {T,S}
```

Overload this method for new types S to throw an informative error if a user-defined function returns the type S instead of T.

MathOptInterface.Nonlinear.eval_univariate_function – Function.

```
eval_univariate_function(
    registry::OperatorRegistry,
    op::Symbol,
    x::T,
) where {T}
```

Evaluate the operator op(x)::T, where op is a univariate function in registry.

MathOptInterface.Nonlinear.eval_univariate_gradient – Function.

```
eval_univariate_gradient(
    registry::OperatorRegistry,
    op::Symbol,
    x::T,
) where {T}
```

Evaluate the first-derivative of the operator op(x)::T, where op is a univariate function in registry.

MathOptInterface.Nonlinear.eval_univariate_hessian – Function.

```
eval_univariate_hessian(
    registry::OperatorRegistry,
    op::Symbol,
    x::T,
) where {T}
```

Evaluate the second-derivative of the operator op(x)::T, where op is a univariate function in registry.

MathOptInterface.Nonlinear.eval_multivariate_function – Function.

```
eval_multivariate_function(
    registry::OperatorRegistry,
    op::Symbol,
    x::AbstractVector{T},
) where {T}
```

Evaluate the operator op(x)::T, where op is a multivariate function in `registry`.

MathOptInterface.Nonlinear.eval_multivariate_gradient – Function.

```
eval_multivariate_gradient(
    registry::OperatorRegistry,
    op::Symbol,
    g::AbstractVector{T},
    x::AbstractVector{T},
) where {T}
```

Evaluate the gradient of operator g .= ∇op(x), where op is a multivariate function in `registry`.

MathOptInterface.Nonlinear.eval_multivariate_hessian – Function.

```
eval_multivariate_hessian(
    registry::OperatorRegistry,
    op::Symbol,
    H::AbstractMatrix,
    x::AbstractVector{T},
) where {T}
```

Evaluate the Hessian of operator ∇^2op(x), where op is a multivariate function in `registry`.

The Hessian is stored in the lower-triangular part of the matrix H.

> **Note**
>
> Implementations of the Hessian operators will not fill structural zeros. Therefore, before calling this function you should pre-populate the matrix H with 0.

MathOptInterface.Nonlinear.eval_logic_function – Function.

```
eval_logic_function(
    registry::OperatorRegistry,
    op::Symbol,
    lhs::T,
    rhs::T,
)::Bool where {T}
```

Evaluate (lhs op rhs)::Bool, where op is a logic operator in `registry`.

MathOptInterface.Nonlinear.eval_comparison_function – Function.

```
eval_comparison_function(
    registry::OperatorRegistry,
    op::Symbol,
    lhs::T,
    rhs::T,
)::Bool where {T}
```

Evaluate (lhs op rhs)::Bool, where op is a comparison operator in registry.

Automatic-differentiation backends

MathOptInterface.Nonlinear.Evaluator – Type.

```
Evaluator(
    model::Model,
    backend::AbstractAutomaticDifferentiation,
    ordered_variables::Vector{MOI.VariableIndex},
)
```

Create Evaluator, a subtype of MOI.AbstractNLPEvaluator, from Model.

MathOptInterface.Nonlinear.AbstractAutomaticDifferentiation – Type.

```
AbstractAutomaticDifferentiation
```

An abstract type for extending Evaluator.

MathOptInterface.Nonlinear.ExprGraphOnly – Type.

```
ExprGraphOnly() <: AbstractAutomaticDifferentiation
```

The default implementation of AbstractAutomaticDifferentiation. The only supported feature is :ExprGraph.

MathOptInterface.Nonlinear.SparseReverseMode – Type.

```
SparseReverseMode() <: AbstractAutomaticDifferentiation
```

An implementation of AbstractAutomaticDifferentiation that uses sparse reverse-mode automatic differentiation to compute derivatives. Supports all features in the MOI nonlinear interface.

Data-structure

MathOptInterface.Nonlinear.Node – Type.

```
struct Node
    type::NodeType
    index::Int
    parent::Int
end
```

A single node in a nonlinear expression tree. Used by Expression.

See the MathOptInterface documentation for information on how the nodes and values form an expression tree.

MathOptInterface.Nonlinear.NodeType – Type.

```
NodeType
```

An enum describing the possible node types. Each Node has a .index field, which should be interpreted as follows:

- NODE_CALL_MULTIVARIATE: the index into operators.multivariate_operators
- NODE_CALL_UNIVARIATE: the index into operators.univariate_operators
- NODE_LOGIC: the index into operators.logic_operators
- NODE_COMPARISON: the index into operators.comparison_operators
- NODE_MOI_VARIABLE: the value of MOI.VariableIndex(index) in the user's space of the model.
- NODE_VARIABLE: the 1-based index of the internal vector
- NODE_VALUE: the index into the .values field of Expression
- NODE_PARAMETER: the index into data.parameters
- NODE_SUBEXPRESSION: the index into data.expressions

MathOptInterface.Nonlinear.Expression – Type.

```
struct Expression
    nodes::Vector{Node}
    values::Vector{Float64}
end
```

The core type that represents a nonlinear expression. See the MathOptInterface documentation for information on how the nodes and values form an expression tree.

MathOptInterface.Nonlinear.Constraint – Type.

```
struct Constraint
    expression::Expression
    set::Union{
        MOI.LessThan{Float64},
        MOI.GreaterThan{Float64},
```

```
        MOI.EqualTo{Float64},
        MOI.Interval{Float64},
    }
end
```

A type to hold information relating to the nonlinear constraint f(x) in S, where f(x) is defined by .expression, and S is .set.

MathOptInterface.Nonlinear.adjacency_matrix – Function.

```
adjacency_matrix(nodes::Vector{Node})
```

Compute the sparse adjacency matrix describing the parent-child relationships in nodes.

The element (i, j) is true if there is an edge from node[j] to node[i]. Since we get a column-oriented matrix, this gives us a fast way to look up the edges leaving any node (i.e., the children).

MathOptInterface.Nonlinear.parse_expression – Function.

```
parse_expression(data::Model, input)::Expression
```

Parse input into a Expression.

```
parse_expression(
    data::Model,
    expr::Expression,
    input::Any,
    parent_index::Int,
)::Expression
```

Parse input into a Expression, and add it to expr as a child of expr.nodes[parent_index]. Existing subexpressions and parameters are stored in data.

You can extend parsing support to new types of objects by overloading this method with a different type on input::Any.

MathOptInterface.Nonlinear.convert_to_expr – Function.

```
convert_to_expr(data::Model, expr::Expression)
```

Convert the Expression expr into a Julia Expr.

- subexpressions are represented by a ExpressionIndex object.
- parameters are represented by a ParameterIndex object.
- variables are represennted by an MOI.VariableIndex object.

```
convert_to_expr(
    evaluator::Evaluator,
    expr::Expression;
    moi_output_format::Bool,
)
```

Convert the Expression expr into a Julia Expr.

If moi_output_format = true:

- subexpressions will be converted to Julia Expr and substituted into the output expression.
- the current value of each parameter will be interpolated into the expression
- variables will be represented in the form x[MOI.VariableIndex(i)]

If moi_output_format = false:

- subexpressions will be represented by a ExpressionIndex object.
- parameters will be represented by a ParameterIndex object.
- variables will be represennted by an MOI.VariableIndex object.

> **Warning**
>
> To use moi_output_format = true, you must have first called MOI.initialize with :ExprGraph as a requested feature.

MathOptInterface.Nonlinear.ordinal_index – Function.

```
ordinal_index(evaluator::Evaluator, c::ConstraintIndex)::Int
```

Return the 1-indexed value of the constraint index c in evaluator.

Examples

```
model = Model()
x = MOI.VariableIndex(1)
c1 = add_constraint(model, :($x^2), MOI.LessThan(1.0))
c2 = add_constraint(model, :($x^2), MOI.LessThan(1.0))
evaluator = Evaluator(model)
MOI.initialize(evaluator, Symbol[])
ordinal_index(evaluator, c2)  # Returns 2
delete(model, c1)
evaluator = Evaluator(model)
MOI.initialize(evaluator, Symbol[])
ordinal_index(model, c2)  # Returns 1
```

42.5 Utilities

Overview

The Utilities submodule

The Utilities submodule provides a variety of functions and datastructures 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
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
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 is 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, for example, Integer

2. Typed scalar sets, for example, LessThan

3. Un-typed vector sets, for example, Nonnegatives

4. Typed vector sets, for example, PowerCone

5. Un-typed scalar functions, for example, `VariableIndex`

6. Typed scalar functions, for example, `ScalarAffineFunction`

7. Un-typed vector functions, for example, `VectorOfVariables`

8. Typed vector functions, for example, `VectorAffineFunction`

The tuples can contain more than one element. Typed-sets must be specified without their type parameter, for example, `MOI.LessThan`, not `MOI.LessThan{Float64}`.

Here is an example:

```julia
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
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
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
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, for example, 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 (for example, they support adding variables one-by-one), and solvers that require the entire problem in a single API call (for example, 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
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)
MOI.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.PenaltyRelaxation

Pass `Utilities.PenaltyRelaxation` to `modify` to relax the problem by adding penalized slack variables to the constraints. This is helpful when debugging sources of infeasible models.

```julia
julia> model = MOI.Utilities.Model{Float64}();

julia> x = MOI.add_variable(model);

julia> MOI.set(model, MOI.VariableName(), x, "x")

julia> c = MOI.add_constraint(model, 1.0 * x, MOI.LessThan(2.0));

julia> map = MOI.modify(model, MOI.Utilities.PenaltyRelaxation(Dict(c => 2.0)));

julia> print(model)
Minimize ScalarAffineFunction{Float64}:
 0.0 + 2.0 v[2]

Subject to:

ScalarAffineFunction{Float64}-in-LessThan{Float64}
 0.0 + 1.0 x - 1.0 v[2] <= 2.0

VariableIndex-in-GreaterThan{Float64}
 v[2] >= 0.0

julia> map[c]
0.0 + 1.0 MOI.VariableIndex(2)
```

You can also modify a single constraint using `Utilities.ScalarPenaltyRelaxation`:

```julia
julia> model = MOI.Utilities.Model{Float64}();

julia> x = MOI.add_variable(model);

julia> MOI.set(model, MOI.VariableName(), x, "x")

julia> c = MOI.add_constraint(model, 1.0 * x, MOI.LessThan(2.0));

julia> f = MOI.modify(model, c, MOI.Utilities.ScalarPenaltyRelaxation(2.0));

julia> print(model)
Minimize ScalarAffineFunction{Float64}:
 0.0 + 2.0 v[2]

Subject to:
```

```
ScalarAffineFunction{Float64}-in-LessThan{Float64}
 0.0 + 1.0 x - 1.0 v[2] <= 2.0

VariableIndex-in-GreaterThan{Float64}
 v[2] >= 0.0

julia> f
0.0 + 1.0 MOI.VariableIndex(2)
```

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 parameterizing 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:

```
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
```

```julia
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:

```julia
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:

```julia
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 MOI.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, MOI.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 MOI.VariableIndex should not be given in scalar_functions. The model supports MOI.VariableIndex-in-S constraints where S is MOI.EqualTo, MOI.GreaterThan, MOI.LessThan, MOI.Interval, MOI.Integer, MOI.ZeroOne, MOI.Semicontinuous or MOI.Semiinteger. The sets supported with the MOI.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 MOI.AbstractOptimizer, otherwise, it is a GenericModel, which is a subtype of MOI.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.

`.objective`

`MathOptInterface.Utilities.ObjectiveContainer` – Type.

```
ObjectiveContainer{T}
```

A helper struct to simplify the handling of objective functions in Utilities.Model.

`.variables`

`MathOptInterface.Utilities.VariablesContainer` – Type.

```
struct VariablesContainer{T} <: AbstractVectorBounds
    set_mask::Vector{UInt16}
    lower::Vector{T}
    upper::Vector{T}
end
```

A struct for storing variables and VariableIndex-related constraints. Used in `MOI.Utilities.Model` by default.

`MathOptInterface.Utilities.FreeVariables` – Type.

```
mutable struct FreeVariables <: MOI.ModelLike
    n::Int64
    FreeVariables() = new(0)
end
```

A struct for storing free variables that can be used as the `variables` field of `GenericModel` or `GenericModel`. It represents a model that does not support any constraint nor objective function.

Example

The following model type represents a conic model in geometric form. As opposed to `VariablesContainer`, FreeVariables does not support constraint bounds so they are bridged into an affine constraint in the `MOI.Nonnegatives` cone as expected for the geometric conic form.

```
julia> MOI.Utilities.@product_of_sets(
    Cones,
    MOI.Zeros,
    MOI.Nonnegatives,
    MOI.SecondOrderCone,
    MOI.PositiveSemidefiniteConeTriangle,
);

julia> const ConicModel{T} = MOI.Utilities.GenericOptimizer{
    T,
    MOI.Utilities.ObjectiveContainer{T},
    MOI.Utilities.FreeVariables,
    MOI.Utilities.MatrixOfConstraints{
        T,
        MOI.Utilities.MutableSparseMatrixCSC{
            T,
            Int,
            MOI.Utilities.OneBasedIndexing,
        },
        Vector{T},
        Cones{T},
    },
};

julia> model = MOI.instantiate(ConicModel{Float64}, with_bridge_type=Float64);

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

julia> c = MOI.add_constraint(model, x, MOI.GreaterThan(1.0))
MathOptInterface.ConstraintIndex{MathOptInterface.VariableIndex,
↪   MathOptInterface.GreaterThan{Float64}}(1)

julia> MOI.Bridges.is_bridged(model, c)
true

julia> bridge = MOI.Bridges.bridge(model, c)
MathOptInterface.Bridges.Constraint.VectorizeBridge{Float64,
↪   MathOptInterface.VectorAffineFunction{Float64}, MathOptInterface.Nonnegatives,
↪   MathOptInterface.VariableIndex}(MathOptInterface.ConstraintIndex{MathOptInterface.VectorAffineFunction{Float64
↪   MathOptInterface.Nonnegatives}(1), 1.0)
```

```
julia> bridge.vector_constraint
MathOptInterface.ConstraintIndex{MathOptInterface.VectorAffineFunction{Float64},
↪   MathOptInterface.Nonnegatives}(1)

julia> MOI.Bridges.is_bridged(model, bridge.vector_constraint)
false
```

.constraints

MathOptInterface.Utilities.VectorOfConstraints – Type.

```
mutable struct VectorOfConstraints{
    F<:MOI.AbstractFunction,
    S<:MOI.AbstractSet,
} <: MOI.ModelLike
    constraints::CleverDicts.CleverDict{
        MOI.ConstraintIndex{F,S},
        Tuple{F,S},
        typeof(CleverDicts.key_to_index),
        typeof(CleverDicts.index_to_key),
    }
end
```

A struct storing F-in-S constraints as a mapping between the constraint indices to the corresponding tuple of function and set.

MathOptInterface.Utilities.StructOfConstraints – Type.

```
abstract type StructOfConstraints <: MOI.ModelLike end
```

A struct storing a subfields other structs storing constraints of different types.

See Utilities.@struct_of_constraints_by_function_types and Utilities.@struct_of_constraints_by_set_types.

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 ith 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 ith 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 ith 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.

Constructors

```
CachingOptimizer(cache::MOI.ModelLike, optimizer::AbstractOptimizer)
```

Creates a CachingOptimizer in AUTOMATIC mode, with the optimizer optimizer.

The type of the optimizer returned is CachingOptimizer{typeof(optimizer), typeof(cache)} so it does not support the function reset_optimizer(::CachingOptimizer, new_optimizer) if the type of new_optimizer is different from the type of optimizer.

```
CachingOptimizer(cache::MOI.ModelLike, mode::CachingOptimizerMode)
```

Creates a CachingOptimizer in the NO_OPTIMIZER state and mode mode.

The type of the optimizer returned is CachingOptimizer{MOI.AbstractOptimizer,typeof(cache)} so it does support the function reset_optimizer(::CachingOptimizer, new_optimizer) if the type of new_optimizer is different from the type of optimizer.

About the type

States

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.

Modes

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 `MOI.copy_to` for more details on which errors can be thrown.

```
MOIU.attach_optimizer(model::GenericModel)
```

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::GenericModel, optimizer::MOI.AbstractOptimizer)
```

Call MOIU.reset_optimizer on the backend of model.

Cannot be called in direct mode.

source

```
MOIU.reset_optimizer(model::GenericModel)
```

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::GenericModel)
```

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 MOI.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 each 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)
```

Penalty relaxation

MathOptInterface.Utilities.PenaltyRelaxation – Type.

```
PenaltyRelaxation(
    penalties = Dict{MOI.ConstraintIndex,Float64}();
    default::Union{Nothing,T} = 1.0,
)
```

A problem modifier that, when passed to MOI.modify, destructively modifies the model in-place to create a penalized relaxation of the constraints.

> **Warning**
>
> This is a destructive routine that modifies the model in-place. If you don't want to modify the original model, use JuMP.copy_model to create a copy before calling MOI.modify.

Reformulation

See Utilities.ScalarPenaltyRelaxation for details of the reformulation.

For each constraint ci, the penalty passed to Utilities.ScalarPenaltyRelaxation is get(penalties, ci, default). If the value is nothing, because ci does not exist in penalties and default = nothing, then the constraint is skipped.

Return value

MOI.modify(model, PenaltyRelaxation()) returns a Dict{MOI.ConstraintIndex,MOI.ScalarAffineFunction} that maps each constraint index to the corresponding y + z as a MOI.ScalarAffineFunction. In an optimal solution, query the value of these functions to compute the violation of each constraint.

Relax a subset of constraints

To relax a subset of constraints, pass a penalties dictionary and set default = nothing.

Supported constraint types

The penalty relaxation is currently limited to modifying MOI.ScalarAffineFunction and MOI.ScalarQuadraticFunction constraints in the linear sets MOI.LessThan, MOI.GreaterThan, MOI.EqualTo and MOI.Interval.

It does not include variable bound or integrality constraints, because these cannot be modified in-place.

To modify variable bounds, rewrite them as linear constraints.

Examples

```julia
julia> model = MOI.Utilities.Model{Float64}();

julia> x = MOI.add_variable(model);

julia> c = MOI.add_constraint(model, 1.0 * x, MOI.LessThan(2.0));

julia> map = MOI.modify(model, MOI.Utilities.PenaltyRelaxation(default = 2.0));

julia> print(model)
Minimize ScalarAffineFunction{Float64}:
 0.0 + 2.0 v[2]

Subject to:

ScalarAffineFunction{Float64}-in-LessThan{Float64}
 0.0 + 1.0 v[1] - 1.0 v[2] <= 2.0

VariableIndex-in-GreaterThan{Float64}
 v[2] >= 0.0

julia> map[c] isa MOI.ScalarAffineFunction{Float64}
true
```

```julia
julia> model = MOI.Utilities.Model{Float64}();

julia> x = MOI.add_variable(model);

julia> c = MOI.add_constraint(model, 1.0 * x, MOI.LessThan(2.0));

julia> map = MOI.modify(model, MOI.Utilities.PenaltyRelaxation(Dict(c => 3.0)));

julia> print(model)
Minimize ScalarAffineFunction{Float64}:
 0.0 + 3.0 v[2]

Subject to:

ScalarAffineFunction{Float64}-in-LessThan{Float64}
 0.0 + 1.0 v[1] - 1.0 v[2] <= 2.0

VariableIndex-in-GreaterThan{Float64}
 v[2] >= 0.0

julia> map[c] isa MOI.ScalarAffineFunction{Float64}
true
```

MathOptInterface.Utilities.ScalarPenaltyRelaxation – Type.

```
ScalarPenaltyRelaxation(penalty::T) where {T}
```

A problem modifier that, when passed to MOI.modify, destructively modifies the constraint in-place to create a penalized relaxation of the constraint.

> **Warning**
>
> This is a destructive routine that modifies the constraint in-place. If you don't want to modify the original model, use JuMP.copy_model to create a copy before calling MOI.modify.

Reformulation

The penalty relaxation modifies constraints of the form $f(x) \in S$ into $f(x) + y - z \in S$, where $y, z \geq 0$, and then it introduces a penalty term into the objective of $a \times (y + z)$ (if minimizing, else $-a$), where a is penalty

When S is MOI.LessThan or MOI.GreaterThan, we omit y or z respectively as a performance optimization.

Return value

MOI.modify(model, ci, ScalarPenaltyRelaxation(penalty)) returns y + z as a MOI.ScalarAffineFunction. In an optimal solution, query the value of this function to compute the violation of the constraint.

Examples

```julia
julia> model = MOI.Utilities.Model{Float64}();

julia> x = MOI.add_variable(model);

julia> c = MOI.add_constraint(model, 1.0 * x, MOI.LessThan(2.0));

julia> f = MOI.modify(model, c, MOI.Utilities.ScalarPenaltyRelaxation(2.0));

julia> print(model)
Minimize ScalarAffineFunction{Float64}:
 0.0 + 2.0 v[2]

Subject to:

ScalarAffineFunction{Float64}-in-LessThan{Float64}
 0.0 + 1.0 v[1] - 1.0 v[2] <= 2.0

VariableIndex-in-GreaterThan{Float64}
 v[2] >= 0.0

julia> f isa MOI.ScalarAffineFunction{Float64}
true
```

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 ith 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}
    nz_added::Vector{Ti}
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. In addition, nz_added is used
to cache the number of non-zero terms that have been added to each column due to the incremental
nature of load_terms.

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 ith 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 ith 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.num_rows – Function.

```
num_rows(sets::OrderedProductOfSets, ::Type{S}) where {S}
```

Return the number of rows corresponding to a set of type S. That is, it is the sum of the dimensions of the sets of type S.

MathOptInterface.Utilities.set_with_dimension – Function.

```
set_with_dimension(::Type{S}, dim) where {S<:MOI.AbstractVectorSet}
```

Returns the instance of S of MOI.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, attr::MOI.AnyAttribute, 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).

When to implement this method for new types X

This function is used by implementations of MOI.copy_to on constraint functions, attribute values and submittable values. If you define a new attribute whose values x::X contain variable or constraint indices, you must also implement this function.

```
map_indices(
    variable_map::AbstractDict{T,T},
    x::X,
)::X where {T<:MOI.Index,X}
```

Shortcut for map_indices(vi -> variable_map[vi], x).

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 copy of the 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{<:Number})
```

Returns x.

```
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.

Set utilities

The following utilities are available for sets:

MathOptInterface.Utilities.AbstractDistance – Type.

```
abstract type AbstractDistance end
```

An abstract type used to enabble dispatch of Utilities.distance_to_set.

MathOptInterface.Utilities.ProjectionUpperBoundDistance – Type.

```
ProjectionUpperBoundDistance() <: AbstractDistance
```

An upper bound on the minimum distance between point and the closest feasible point in set.

Definition of distance

The minimum distance is computed as:

$$d(x, \mathcal{K}) = \min_{y \in \mathcal{K}} ||x - y||$$

where x is point and $\mathcal{K}$ is set. The norm is computed as:

$$||x|| = \sqrt{f(x, x, \mathcal{K})}$$

where f is Utilities.set_dot.

In the default case, where the set does not have a specialized method for Utilities.set_dot, the norm is equivalent to the Euclidean norm $||x|| = \sqrt{\sum x_i^2}$.

Why an upper bound?

In most cases, distance_to_set should return the smallest upper bound, but it may return a larger value if the smallest upper bound is expensive to compute.

For example, given an epigraph from of a conic set, $\{(t, x)|f(x) \leq t\}$, it may be simpler to return δ such that $f(x) \leq t + \delta$, rather than computing the nearest projection onto the set.

If the distance is not the smallest upper bound, the docstring of the appropriate distance_to_set method must describe the way that the distance is computed.

MathOptInterface.Utilities.distance_to_set – Function.

```
distance_to_set(
    [d::AbstractDistance = ProjectionUpperBoundDistance()],]
    point::T,
    set::MOI.AbstractScalarSet,
) where {T}

distance_to_set(
```

```
    [d::AbstractDistance = ProjectionUpperBoundDistance(),]
    point::AbstractVector{T},
    set::MOI.AbstractVectorSet,
) where {T}
```

Compute the distance between point and set using the distance metric d. If point is in the set set, this function must return zero(T).

If d is omitted, the default distance is Utilities.ProjectionUpperBoundDistance.

MathOptInterface.Utilities.set_dot – Function.

```
set_dot(x::AbstractVector, y::AbstractVector, set::AbstractVectorSet)
```

Return the scalar product between a vector x of the set set and a vector y of the dual of the set s.

```
set_dot(x, y, set::AbstractScalarSet)
```

Return the scalar product between a number x of the set set and a number y of the dual of the set s.

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`.

`MathOptInterface.Utilities.DoubleDicts.outer_keys` – Function.

```
outer_keys(d::AbstractDoubleDict)
```

Return an iterator over the outer keys of the `AbstractDoubleDict` d. Each outer key is a `Tuple{Type,Type}` so that a double loop can be easily used:

```
for (F, S) in DoubleDicts.outer_keys(dict)
    for (k, v) in dict[F, S]
        # ...
    end
end
```

For performance, it is recommended that the inner loop lies in a separate function to gurantee type-stability. Some outer keys `(F, S)` might lead to an empty `dict[F, S]`. If you want only nonempty `dict[F, S]`, use `nonempty_outer_keys`.

`MathOptInterface.Utilities.DoubleDicts.nonempty_outer_keys` – Function.

```
nonempty_outer_keys(d::AbstractDoubleDict)
```

Return a vector of outer keys of the `AbstractDoubleDict` d.

Only outer keys that have a nonempty set of inner keys will be returned.

Each outer key is a `Tuple{Type,Type}` so that a double loop can be easily used

```
for (F, S) in DoubleDicts.nonempty_outer_keys(dict)
    for (k, v) in dict[F, S]
        # ...
    end
end
For performance, it is recommended that the inner loop lies in a separate
function to gurantee type-stability.

If you want an iterator of all current outer keys, use [`outer_keys`](@ref).
```

42.6 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 Test

import MathOptInterface as MOI

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],
)
```

```julia
"""
    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_",
        ],
        # This argument is useful to prevent tests from failing on future
        # releases of MOI that add new tests. Don't let this number get too far
        # behind the current MOI release though. You should periodically check
        # for new tests to fix bugs and implement new features.
        exclude_tests_after = v"0.10.5",
    )
    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 trans-
> form 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_failed` from the REPL to read it.

Periodically, you should re-run excluded tests to see if they now pass. The easiest way to do this is to swap the exclude keyword argument of `runtests` to `include`. For example:

```
MOI.Test.runtests(
    model,
    config;
    exclude = String["test_to_exclude", "test_conic_"],
)
```

becomes

```
MOI.Test.runtests(
    model,
    config;
    include = String["test_to_exclude", "test_conic_"],
)
```

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:

```
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/test_solve.jl:

```
"""
    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 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 agnostic 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
```

Finally, you also need to implement Test.version_added. If we added this test when the latest released version of MOI was v0.10.5, define:

```
version_added(::typeof(test_unit_optimize!_twice)) = v"0.10.6"
```

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,
    infeasible_status::MOI.TerminationStatusCode = MOI.INFEASIBLE,
    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.

- `infeasible_status = MOI.INFEASIBLE`: Set to `MOI.LOCALLY_INFEASIBLE` if the solver cannot prove global infeasibility.

- `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{Union{String,Regex}} = String[],
    exclude::Vector{Union{String,Regex}} = String[],
    warn_unsupported::Bool = false,
    exclude_tests_after::VersionNumber = v"999.0.0",
)
```

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 if an element from `include` occurs in the name of the test.

- If `exclude` is not empty, skip tests if an element from `exclude` occurs in the name of the test.

- `exclude` takes priority over `include`.

- If warn_unsupported is false, runtests will silently skip tests that fail with a MOI.NotAllowedError, MOI.UnsupportedError, or RequirementUnmet error. (The latter is thrown when an @requires statement returns false.) 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.

- exclude_tests_after is a version number that excludes any tests to MOI added after that version number. This is useful for solvers who can declare a fixed set of tests, and not cause their tests to break if a new patch of MOI is released with a new test.

See also: setup_test.

Example

```
config = MathOptInterface.Test.Config()
MathOptInterface.Test.runtests(
    model,
    config;
    include = ["test_linear_", r"^test_model_Name$"],
    exclude = ["VariablePrimalStart"],
    warn_unsupported = true,
    exclude_tests_after = v"0.10.5",
)
```

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.version_added – Function.

```
version_added(::typeof(function_name))
```

Returns the version of MOI in which the test function_name was added.

This method should be implemented for all new tests.

See the exclude_tests_after keyword of runtests for more details.

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 43

Developer Docs

43.1 Checklists

The purpose of this page is to collate a series of checklists for commonly performed changes to the source code of MathOptInterface.

In each case, copy the checklist into the description of the pull request.

Making a release

Use this checklist when making a release of the MathOptInterface repository.

```
## Basic

 - [ ] `version` field of `Project.toml` has been updated
        - If a breaking change, increment the MAJOR field and reset others to 0
        - If adding new features, increment the MINOR field and reset PATCH to 0
        - If adding bug fixes or documentation changes, increment the PATCH field

## Documentation

 - [ ] Add a new entry to `docs/src/changelog.md`, following existing style

## Tests

 - [ ] The `solver-tests.yml` GitHub action does not have unexpected failures.
        To run the action, go to:
        https://github.com/jump-dev/MathOptInterface.jl/actions/workflows/solver-tests.yml
        and click "Run workflow"
```

Adding a new set

Use this checklist when adding a new set to the MathOptInterface repository.

```
## Basic

 - [ ] Add a new `AbstractScalarSet` or `AbstractVectorSet` to `src/sets.jl`
 - [ ] If `isbitstype(S) == false`, implement `Base.copy(set::S)`
 - [ ] If `isbitstype(S) == false`, implement `Base.:(==)(x::S, y::S)`
 - [ ] If an `AbstractVectorSet`, implement `dimension(set::S)`, unless the
```

```
      dimension is given by `set.dimension`.

## Utilities

 - [ ] If an `AbstractVectorSet`, implement `Utilities.set_dot`,
       unless the dot product between two vectors in the set is equivalent to
       `LinearAlgebra.dot`
 - [ ] If an `AbstractVectorSet`, implement `Utilities.set_with_dimension` in
       `src/Utilities/matrix_of_constraints.jl`
 - [ ] Add the set to the `@model` macro at the bottom of `src/Utilities.model.jl`

## Documentation

 - [ ] Add a docstring, which gives the mathematical definition of the set,
       along with an `## Example` block containing a `jldoctest`
 - [ ] Add the docstring to `docs/src/reference/standard_form.md`
 - [ ] Add the set to the relevant table in `docs/src/manual/standard_form.md`

## Tests

 - [ ] Define a new `_set(::Type{S})` method in `src/Test/test_basic_constraint.jl`
       and add the name of the set to the list at the bottom of that files
 - [ ] If the set has any checks in its constructor, add tests to `test/sets.jl`

## MathOptFormat

 - [ ] Open an issue at `https://github.com/jump-dev/MathOptFormat` to add
       support for the new set {{ replace with link to the issue }}

## Optional

 - [ ] Implement `dual_set(::S)` and `dual_set_type(::Type{S})`
 - [ ] Add new tests to the `Test` submodule exercising your new set
 - [ ] Add new bridges to convert your set into more commonly used sets
```

Adding a new bridge

Use this checklist when adding a new bridge to the MathOptInterface repository.

The steps are mostly the same, but locations depend on whether the bridge is a Constraint, Objective, or Variable bridge. In each case below, replace XXX with the appropriate type of bridge.

```
## Basic

 - [ ] Create a new file in `src/Bridges/XXX/bridges`
 - [ ] Define the bridge, following existing examples. The name of the bridge
       struct must end in `Bridge`
 - [ ] Check if your bridge can be a subtype of [`MOI.Bridges.Constraint.SetMapBridge`](@ref)
 - [ ] Define a new `const` that is a `SingleBridgeOptimizer` wrapping the
       new bridge. The name of the const must be the name of the bridge, less
       the `Bridge` suffix
 - [ ] `include` the file in `src/Bridges/XXX/bridges/XXX.jl`
 - [ ] If the bridge should be enabled by default, add the bridge to
       `add_all_bridges` at the bottom of `src/Bridges/XXX/XXX.jl`
```

```
## Tests

 - [ ] Create a new file in the appropriate subdirectory of `tests/Bridges/XXX`
 - [ ] Use `MOI.Bridges.runtests` to test various inputs and outputs of the
       bridge
 - [ ] If, after opening the pull request to add the bridge, some lines are not
       covered by the tests, add additional bridge-specific tests to cover the
       untested lines.

## Documentation

 - [ ] Add a docstring which uses the same template as existing bridges.
 - [ ] Add the docstring to `docs/src/submodules/Bridges/list_of_bridges.md`

## Final touch

If the bridge depends on run-time values of other variables and constraints in
the model:

 - [ ] Implement `MOI.Utilities.needs_final_touch(::Bridge)`
 - [ ] Implement `MOI.Utilities.final_touch(::Bridge, ::MOI.ModelLike)`
 - [ ] Ensure that `final_touch` can be called multiple times in a row
```

Updating MathOptFormat

Use this checklist when updating the version of MathOptFormat.

```
## Basic

 - [ ] The file at `src/FileFormats/MOF/mof.X.Y.schema.json` is updated
 - [ ] The constants `SCHEMA_PATH`, `VERSION`, and `SUPPORTED_VERSIONS` are
       updated in `src/FileFormats/MOF/MOF.jl`

## New sets

 - [ ] New sets are added to the `@model` in `src/FileFormats/MOF/MOF.jl`
 - [ ] New sets are added to the `@enum` in `src/FileFormats/MOF/read.jl`
 - [ ] `set_to_moi` is defined for each set in `src/FileFormats/MOF/read.jl`
 - [ ] `head_name` is defined for each set in `src/FileFormats/MOF/write.jl`
 - [ ] A new unit test calling `_test_model_equality` is aded to
       `test/FileFormats/MOF/MOF.jl`

## Tests

 - [ ] The version field in `test/FileFormats/MOF/nlp.mof.json` is updated

## Documentation

 - [ ] The version fields are updated in `docs/src/submodules/FileFormats/overview.md`
```