# Chapter 6

# JuMPeR: Algebraic Modeling for Robust and Adaptive Optimization

The histories of computing and the solution of mathematical optimization problems are intricately linked, as rapid increases from the 1950s onwards in the availability and power of computers were applied widely to solving a variety of planning problems in industry and military applications [Orchard-Hays, 1984]. However, as the capability to solve ever-larger problems grew, a gap developed between the ability to solve larger problems and the ability to describe and manipulate them. Input formats for solvers were painful and slow for people to use, and were proving a major barrier to the further adoption of optimization technology [Fourer, 2012].

*Algebraic modeling languages* (AMLs) were developed in the late 1970s as a solution to this problem. They enabled users to express their optimization problems in a natural format that is similar to the original mathematical expressions, and automate the translation to a lower-level format suitable for solvers. Two of the first AMLs that made a significant impact on the field of optimization, and are still in use today, are the commercial packages GAMS [Brooke et al., 1999] and AMPL [Fourer et al., 2003] (created in 1978 and 1985 respectively). For the most part, AMLs have focused on two broad classes of optimization problems: mixed-integer linear optimization (MILO) and quadratic optimization (MIQO) problems, and constrained nonlinear optimization problems. For MILO problems, solvers typically expect the problem to presented in "standard computational form" (i.e., $\min_{\mathbf{x}\geq\mathbf{0}} \mathbf{c}^T\mathbf{x},\ \ \mathbf{A}\mathbf{x} = \mathbf{b}$), so the task of an AML is primarily to generate the sparse $\mathbf{A}$ matrix and vectors $\mathbf{c}$ and $\mathbf{b}$. While there may be some problem transformations (e.g., "presolve"), in general there is normally a simple mapping between the user's input and the solver's input – but this mapping is one that would be painful to perform manually. In the majority of AMLs there is no capability for more advanced transformations or interactions with the solver – for example, in a scenario-based approach to solve a stochastic program, the user is responsible for manually iterating over the scenarios to express the recourse decision variables and constraints.

Robust optimization (RO) problems, which we address in this chapter, require the user to either manually reformulate their optimization problem using duality (necessitating the inconvenient introduction of auxiliary variables and constraints) or implementing a cutting plane method from scratch. While performing these operations even once can be time-consuming and error prone, small changes in the problem structure (e.g., changing the uncertainty set) can require substantial effort on the users part to update the model. We suggest that, much as the lack of AMLs hindered uptake of mathematical optimization on computers in the past, the lack of AMLs for more complex settings such as robust and adaptive optimization prevents uptake by practitioners and introduces inefficiencies for researchers.

In this chapter, we present JuMPeR, an extension to the JuMP modeling language [Lubin and Dunning, 2015, Dunning et al., 2015]. It is available for download with the Julia package manager. It extends the modeling capabilities of JuMP by adding primitives for uncertain parameters, adaptive decisions, and uncertainty sets, enabling a rich variety of RO problems to modeled in a high-level fashion independent of the particular solution method. It interfaces with a wide variety of solvers, inherits all the general-purpose programming capabilities of its host language Julia [Bezanson et al., 2014], and is designed to be extensible for new developments in RO and adaptive RO (ARO).

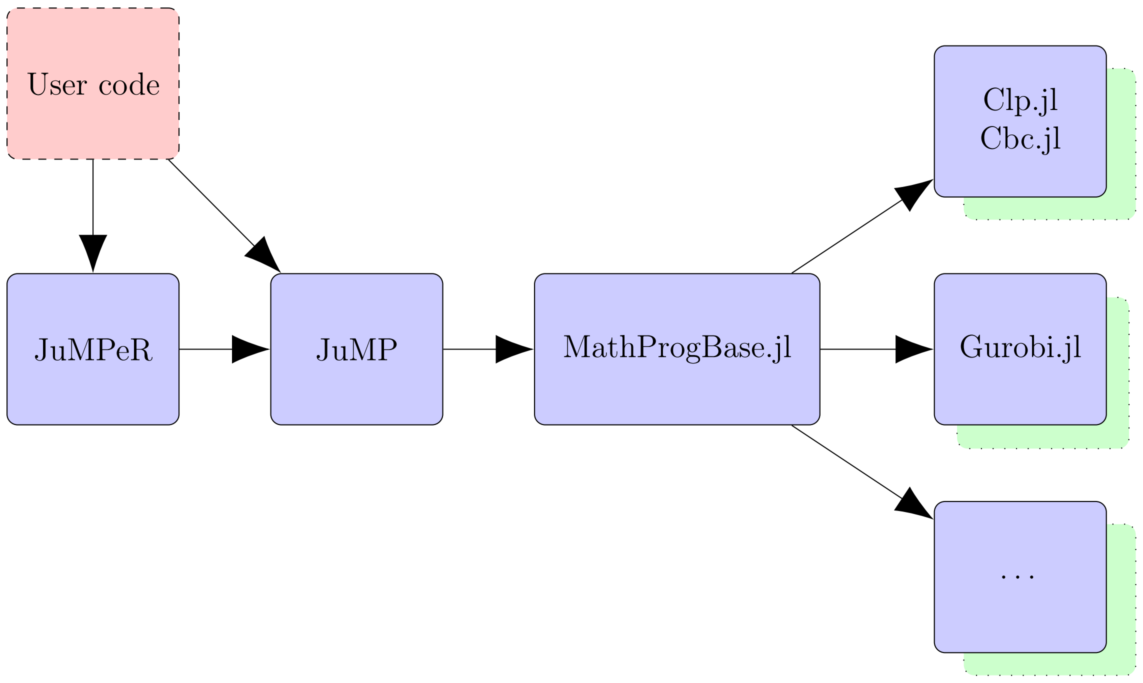

Figure 6-1: Overview of how JuMPeR interacts with related packages. User code (red, dashed) depends explicitly on both JuMPeR and JuMP. JuMP depends on the MathProgBase package which provides an abstraction over solvers. Each underlying solver library (green, loose dashes) has a thin Julia wrapper package.

**Chapter structure**

- In Section 6.1, we outline the capabilities of JuMPeR, its relationship to JuMP and solvers, and the key primitives available to build ARO models.

- In Section 6.2, we describe JuMPeR's uncertainty set system and how JuMPeR solves ARO problems.

- In Section 6.3, we present three case studies using JuMPeR: portfolio optimization, multistage inventory control [Ben-Tal et al., 2004], and specialized cutting plane methods.

- Finally, in Section 6.4, we compare and constrast JuMPeR with similar tools.

## 6.1 Overview of JuMPeR

JuMPeR is implemented as a package for the Julia programming language. Julia is a relatively new "high-level, high-performance dynamic programming language for technical computing"[1], with syntax that would be familiar for users of languages like Python or MATLAB. Some of Julia's features are particularly useful for the creation of AMLs: an expressive type system, metaprogramming (i.e., macros) to enable novel syntax, interoperability with shared libraries written in C, and garbage collection (no need for manual memory management). JuMPeR depends on the JuMP package; JuMP is itself a fully-featured AML that can model (mixed-integer) linear, quadratic, second-order cone, semidefinite, and general nonlinear optimization problems. When a user creates a RO model with JuMPeR, they use JuMP explicitly for the deterministic parts of the problem, and implicitly through JuMPeR's internals, which use JuMP to formulate auxiliary cutting plane problems and to add deterministic constraints arising from reformulations (Figure 6-1).

JuMP, and thus JuMPeR, can use the vast majority of both commercial and popular open-source solvers. This is enabled by the MathProgBase.jl package[2], which defines a shared interface that provides an abstraction over most solver idiosyncrasies. This interface is implemented by a variety of "thin" solver-specific packages that wrap solver shared libraries (normally written in C) in Julia code. MathProgBase.jl and these solver wrapper packages are all maintained together under the auspices of the JuliaOpt organization[3].

Both JuMP and JuMPeR work by composing a variety of primitives, or *types*, to construct models (refer to Figure 6-2). The highest-level type, defined by JuMP, is the `Model` type. A `Model` is a collection of decision variables (`Variable`), constraints (`LinearConstraint`, etc.), an objective function, and other metadata such as the last primal and dual solution (if the model has been solved). It also supports an extension mechanism, by which packages can build on JuMP's machinery. JuMPeR defines a

[1]As described on `http://julialang.org` on April 1st, 2016.
[2]Available at `https://github.com/JuliaOpt/MathProgBase.jl`.
[3]See `http://juliaopt.org` for more information.

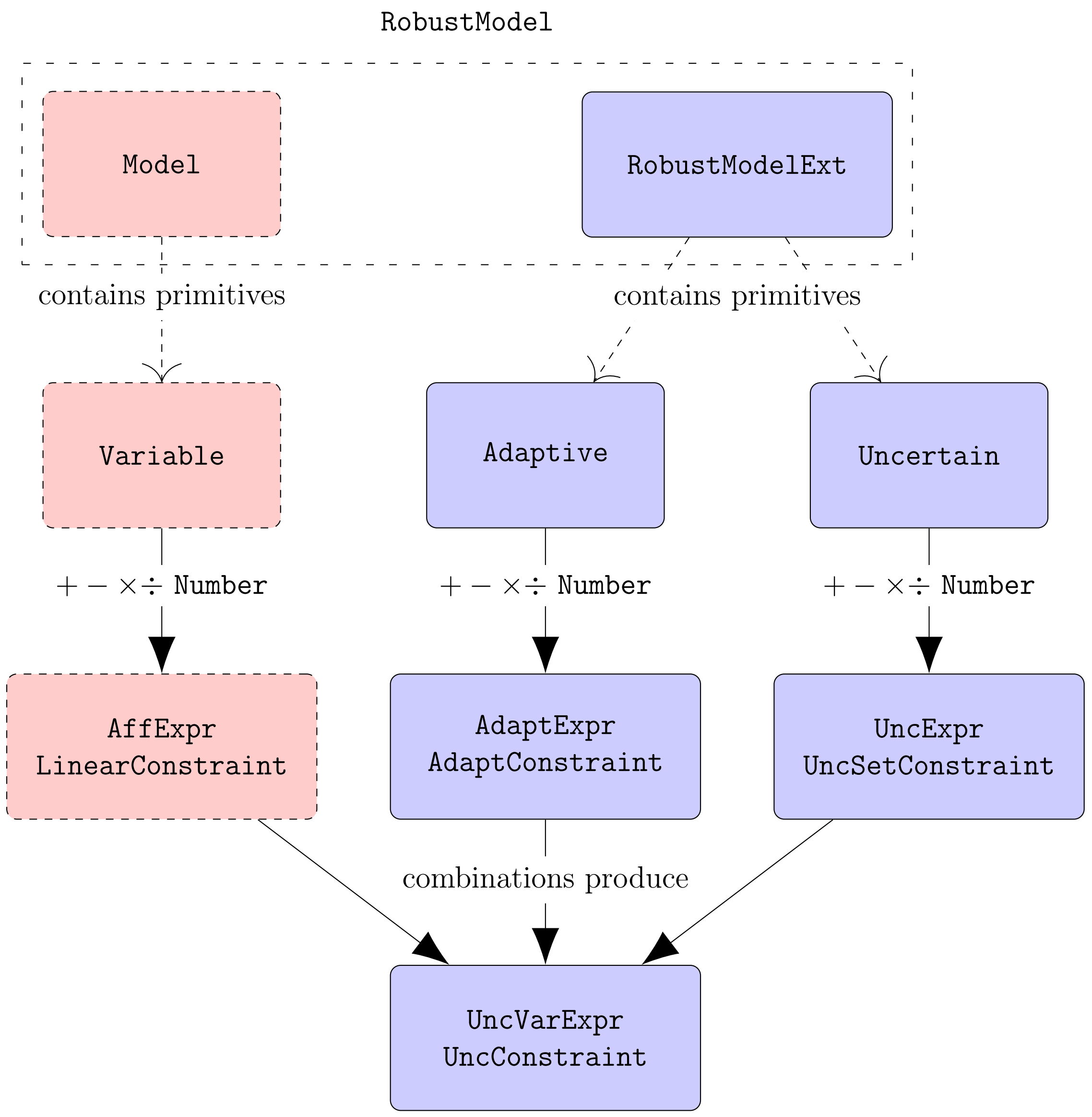

Figure 6-2: Overview of JuMPeR's type system. The red/dashed border types `Model`, `Variable`, `AffExpr` are defined by JuMP, with the remainder defined by JuMPeR. A "`RobustModel`" is a `Model` with an attached extension type `RobustModelExt` containing the RO-specific metadata.

RobustModelExt type that is stored in a Model, and defines a RobustModel function that produces a new model with this extension type already created and attached. We now describe in Section 6.1.1 how uncertain parameters and adaptive variables are defined, and in Section 6.1.2 we describe how these primitives combine into expressions and constraints. A discussion of how uncertainty sets are implemented is deferred to Section 6.2.

### 6.1.1 Uncertain Parameters and Adaptive Variables

JuMP defines the Variable type for constructing models, and JuMPeR introduces two more variable-like types: Uncertain and Adaptive. An Uncertain is an uncertain parameter – it can have lower and upper bounds, and is continuous by default but can be restricted to be binary or integer:

```
rm = RobustModel()  # Creates a new RO model
# Single uncertain parameter
@uncertain(rm, 0 <= capacity <= 100)
# Uncertain parameters indexed by numbers
@uncertain(rm, riskfactor[1:n])
# Uncertain parameters (binary) indexed by arbitrary sets
highways = [:I90, :I93, :I95]
@uncertain(rm, blocked[highways], Bin)
```

An Adaptive is an adaptive decision variable – the value it takes is a function of uncertain parameters. JuMPeR currently provides two simple adaptive policies by default, but can be extended to provide others: Static and Affine. Adaptive variables must be defined in relation to the uncertain parameters that they are a function of. In Section 6.2 we will discuss how Adaptive variables are handled when the model is solved. Here we demonstrate two examples of adaptive variables, including one where we can easily incorporate the temporal structure of a multistage problem.

```
# Creates a set of adaptive variables, where each of
# them is affinely dependent on all the risk factors
```

```
@uncertain(rm, riskfactor[1:n])
@adaptive(rm, allocation[1:n], policy=Affine,
            depends_on=riskfactor)
# Create a set of adaptive variables, where each of
# them is affinely dependent on the demand realized
# up to the time stage when the decision is made
@uncertain(rm, 10 <= demand[1:T] <= 90)
@adaptive(rm, production[t=1:T] >= 0, policy=Affine,
            depends_on=demand[1:t-1])
```

We can read the last expression as "adaptive variable `production[t]` depends on `{demand[1],...,demand[t-1]}`, for all $t \in \{1, \ldots, T\}$".

### 6.1.2 Expressions and Constraints

In both JuMP and JuMPeR affine expressions are stored as lists of tuples, i.e., $\{(c_1, v_1), \ldots, (c_m, v_m)\}$. For example, an expression of numbers and decision variables (e.g., the left-hand-side of a linear constraint), is an `AffExpr`, which is an alias for `GenericAffExpr{Float64,Variable}` (Figure 6-2). A linear constraint (`LinearConstraint`) is then a combination of an `AffExpr`, a sense, and a bound. JuMPeR introduces new aliases and expression-construction machinery for handling the following possibilities:

- `UncExpr/UncSetConstraint`: Affine expression/constraint of numbers and uncertain parameters. This is used to define uncertainty sets, and the coefficients of variables in uncertain constraints.

- `AdaptExpr/AdaptConstraint`: Affine expression/constraint of numbers and adaptive variables, but no explicitly appearing uncertain parameters. These will mostly like become uncertain constraints when the actual adaptive policy is inserted (see Section 6.2). Examples of these can be seen in the multistage inventory case in Section 6.3.2.

- `UncVarExpr`/`UncConstraint`: Affine expression/constraint where the "coefficients" are `UncExpr` (which includes just a number as a degenerate case), and the "variables" are either `Variable` or `Adapt` (or a mix). This expression unifies everything, and is the constraint type that is replaced with deterministic equivalents when the problem is solved.

JuMPeR doesn't support all possible constraint types. Notably, it does not have support for quadratic or semidefinite constraints on uncertain parameters, nor does it allow for constraints with uncertain parameters that are quadratic in the decision variables (i.e., uncertain second-order cone constraints). The lack of these is not inherent to the design of JuMPeR, and could be added in a future version. JuMPeR does have support for expressing simple "norm" constraints on uncertain parameters, including the $1-$, $2-$, and $\infty-$norms, e.g.,

```
@uncertain(rm, riskfactor[1:n])
@constraint(rm, norm(riskfactor, 1) <= 1)
@constraint(rm, norm(riskfactor, 2) <= 1)
@constraint(rm, norm(riskfactor, Inf) <= 1)
```

These are all used in the portfolio optimization case in Section 6.3.1.

## 6.2 Uncertainty Sets

Perhaps the defining feature of JuMPeR is its extensible uncertainty set system. At the heart of this system is the interaction between JuMPeR's core `solve` function, and (possibly user-defined) `UncertaintySet` types that implement the simple `AbstractUncertaintySet` interface. In this section we describe the flow of the `solve` function and its interactions with uncertainty sets.

When a user calls `solve`, the transformation of an RO (or ARO) problem into an deterministic problem begins. We can break this process down into five main stages:

1. `Adaptive` variables are "expanded".

2. Uncertainty sets are notified of which constraints they must reformulate/provide cutting planes for. They are then given the chance to complete any constraint-independent setup.

3. Uncertainty sets are asked to reformulate their associated constraints (and can chose to not do so).

4. Cutting planes:

    - If problem has only continuous variables, solve problem with current constraints. Ask uncertainty sets for any new constraints. If none, terminate. Otherwise, resolve problem and repeat.

    - If problem has any discrete variables, construct lazy constraint callback that queries uncertainty sets for any new constraints. Add lazy constraint callback and solve.

5. If requested, obtain the worst-case uncertain parameters for each constraint from the uncertainty sets.

**Step 1 of 5: "Expanding" Adaptive Variables**

The first step in the solution process is to replace all `Adaptive` variables with a combination of normal `Variable`s and uncertain parameters. This is broken down into two phases. In the first phase, an `UncVarExpr` is created for each `Adaptive` in the model, which will be spliced into the model wherever the adaptive variable appears. For `Static` variables, this simply involves creating a new `Variable` with the same bounds as the `Adaptive`, and setting the expression equal to this new variable. For `Affine` variables, the process is slightly more involved. A new `Variable` $p_i$ is created for each uncertain parameter $u_i$ that the variable depends on, as well as an independent variable $p_0$ – all these variables have no bounds. The expression is then equal to $p_0 + \sum_i u_i p_i$. To handle the bounds on the adaptive variable, up to two constraints on this expression may be added. This is most easily understood through the following snippet, where both versions are equivalent.

```
@defUnc(rm, u[1:n])
# This...
@defAdapt(rm, lb <= x <= ub, policy=Affine, depends_on=u)
# ... is equivalent to the block of code
@defVar(rm, p[1:n])
@defVar(rm, p_indep)
x = p0 + dot(p, u)
@addConstraint(rm, x >= lb)
@addConstraint(rm, x <= ub)
```

The second phase is then, for every constraint with an adaptive variable in it (i.e., all `AdaptConstraint` and some or all of the *UncConstraint*), a new *UncConstraint* is made with these per-`Adaptive` expressions inserted. If the user has a constraint with an affine adaptive variable multiplied with an uncertain parameter, then an error will be thrown at this point as quadratic functions of uncertain parameters and variables are currently unsupported.

### Step 2 of 5: Uncertainty Set Setup

After the `Adaptive` variables have been processed out, we are left with a model containing only uncertain parameters and "normal" decision variables. At this point we set up any uncertainty sets associated with the model. In this context, an uncertainty set is a type that implements the `AbstractUncertaintySet` interface. Each constraint can be explicitly associated with an uncertainty set, but if one is not provided then a default model-wide uncertainty set is used. During this phase of the solution process, the uncertainty set for each `UncConstraint` is determined (either the explicitly provided one, or the default), and the `setup_set` method is called for each uncertainty set once. This is typically used by the set to do any work that can be reused for multiple constraints. For example, the uncertainty set may want to formulate the dual of the cutting plane problem for the purposes of reformulation, or it may want to create an internal JuMP model that, when solved, will produce a cutting plane.

### Step 3 of 5: Reformulation

We now give each uncertainty set the chance to reformulate any of their associated constraints. Some uncertainty sets may not support reformulation, and so will do nothing at this step. Others might support both reformulation and cutting planes, and will only take action at this stage if the user has passed an option to the uncertainty set requesting that it do so – this is the case for the in-built `BasicUncertaintySet`.

While "reformulation" may evoke the duality-based approach commonly described in the literature, the uncertainty set has full freedom in how it approaches this step. The key feature is that it is done before any solving of the RO (or a relaxation of it) takes place. For example, an uncertainty set may be defined by a finite set of scenarios, and at this stage the uncertainty set may choose to add a deterministic constraint for each of them.

### Step 4 of 5: Cutting Planes

We now solve the current version of the RO problem (using, of course, only the deterministic constraints provided initially and by reformulation). If all the uncertainty sets were doing solely reformulation, we would be done with this step. If some sets are using cutting planes, then each uncertainty set is given the chance to return any number of cutting planes it chooses, using the current solution. If any cutting planes are produced, then these cuts are added and the problem is solved again. If none are produced, then we move to the next step.

The exact details of the implementation vary depending on whether there exist any discrete decision variables. If there are none, then the cutting planes are added to the JuMP model and it is resolved, in a loop. This is highly efficient for any solver that supports hot-starting using the dual simplex method, which is the case for the majority of solvers for LO problems. If there are discrete variables, then the above approach would be very inefficient as integer optimization solvers do not have this "hot-start" capability. Instead, we use the "lazy constraint callback" feature that many of these solvers support to integrate cutting planes into the solution process. In this approach, the solver queries JuMPeR at each integer solution whether there any "lazy" constraints that it is not aware of that would make the solution infeasible. JuMPeR then asks this question of each of the uncertainty sets and manages communicating these constraints back to the solver. For more discussion of cutting-plane methods for MIO problems, see Chapter 2.

#### Step 5 of 5: "Active" Scenarios

Some algorithms, including those described in Chapter 3, require access to the worst-case uncertain parameters at optimality. We refer to particular realizations of the uncertain parameters as scenarios, and thus "active" scenarios are the scenarios for which each constraint has minimal slack (or possibly zero slack, which is an active constraint). This final step is an optional one, and is only used if the `active_scenarios` flag is passed to the solve function. If that flag is passed, then each uncertainty set will be asked to produce at most a single scenario for each of its constraints. These can then be accessed on a per-constraint basis by the user after the problem is solved, similar to how dual values are accessed for normal LO problems.

## 6.3 Case Studies

Here we present three case studies that demonstrate different capabilities of JuMPeR:

- In Section 6.3.1, we model a simple single-stage portfolio optimization problem. This demonstrates the basics of JuMPeR, and demonstrates how code can be structured to easily switch between different uncertainty sets with minimal user effort.
- In Section 6.3.2, we model a multi-stage inventory control problem. This demonstrates how we can use affine adaptive variables (linear decision rules).
- In Section 6.3.3, we demonstrate the implementation of a specialized cutting plane generator for "budget" polyhedral uncertainty sets.

### 6.3.1 Portfolio Construction

Our first case is a simple single-stage portfolio construction problem. We have $n$ assets in which we can invest, and our decision is what fraction $x_i \geq 0$ of our wealth to invest in each asset $i$. The return on asset $i$ is an uncertain parameter $r_i$, which we model as being drawn from an uncertainty set $\mathcal{U}$. This leads to the RO problem

$$\begin{aligned} \max_{\mathbf{x},z} \; & z \\ \text{subject to } & z \leq \mathbf{r}^T\mathbf{x} \qquad \forall \mathbf{r} \in \mathcal{U} \\ & \mathbf{1}^T\mathbf{x} = 1 \\ & \mathbf{x} \geq \mathbf{0}, \end{aligned} \tag{6.1}$$

where we have introduced an auxiliary variable $z$ that moves the uncertain objective function $\min_{\mathbf{r}\in\mathcal{U}} \left\{\mathbf{r}^T\mathbf{x}\right\}$ into an epigraph constraint.

We will consider two different "data-driven" uncertainty sets that differ only in their choice of norm. We assume that we have historical data for the returns of each asset, allowing us to estimate their mean $\boldsymbol{\mu}$ and covariance $\boldsymbol{\Sigma}$ (a matrix). The two uncertainty sets we consider are the polyhedral set

$$\mathcal{U}_P = \left\{(\mathbf{r}, \boldsymbol{\xi}) \mid \mathbf{r} = \boldsymbol{\Sigma}^{\frac{1}{2}}\boldsymbol{\xi} + \boldsymbol{\mu},\ \|\boldsymbol{\xi}\|_1 \leq \Gamma_P,\ \|\boldsymbol{\xi}\|_\infty \leq 1\right\}, \tag{6.2}$$

and the ellipsoidal set

$$\mathcal{U}_E = \left\{(\mathbf{r}, \boldsymbol{\xi}) \mid \mathbf{r} = \boldsymbol{\Sigma}^{\frac{1}{2}}\boldsymbol{\xi} + \boldsymbol{\mu},\ \|\boldsymbol{\xi}\|_2 \leq \Gamma_E\right\}, \tag{6.3}$$

where $\boldsymbol{\Sigma}^{\frac{1}{2}}$ can be obtained by the Cholesky decomposition of $\boldsymbol{\Sigma}$, and where $\Gamma_P$ and $\Gamma_E$ control the conservatism of each of the sets. We can consider these $\boldsymbol{\xi}$ to be underlying market factors that induce correlations amongst the returns of the assets. If we take a reformulation approach to solving the RO problem, then the deterministic problem will be a linear optimization (LO) problem in the case of $\mathcal{U}_P$, and a second-order cone optimization (SOCO) problem in the case of $\mathcal{U}_E$.

To demonstrate solving this problem with JuMPeR, we will create a function that takes the past returns (as a matrix with one column per asset), the uncertainty set type, and the value of Γ. We first load JuMP and JuMPeR, but will not explicitly load any solver – instead, one will be selected automatically from the solvers that have been installed depending on the problem class.

```
using JuMP, JuMPeR
```

We now start our function and initialize the deterministic portion of the problem, which is a direct translation of the mathematical description in Equation (6.1). Note the use of non-Latin characters, such as Γ: this is fully supported by Julia and is commonly used for mathematical code such as this. Apart from `RobustModel`, which is defined by JuMPeR, this following lines are using the functionality of JuMP.

```
function solve_portfolio(n, past_returns, set_type, Γ)
m = RobustModel()
@variable(m, 0 <= x[1:n] <= 1)
@constraint(m, sum(x) == 1)
@variable(m, z)
@objective(m, Max, z)
```

Before constructing the uncertainty set we need to extract $\boldsymbol{\mu}$ and $\boldsymbol{\Sigma}$ from the return data, and calculate $\boldsymbol{\Sigma}^{\frac{1}{2}}$ (which is called `L` in the code).

```
μ = vec(mean(past_returns, 1)) # Column mean, as vector
Σ = cov(past_returns)
L = full(chol(Σ))' # Lower Cholesky factor, as matrix
```

We now use JuMPeR to create the uncertainty set, which has four components. First, we define the uncertain parameters $\mathbf{r}$ that appear directly in the model. Second, we define the underlying factor uncertain parameters $\boldsymbol{\xi}$. Third, we connect $\mathbf{r}$ and $\boldsymbol{\xi}$ through $\boldsymbol{\mu}$ and $\boldsymbol{\Sigma}^{\frac{1}{2}}$. Finally, we apply norm constraints on the factors.

```
@uncertain(m, r[1:n])
@uncertain(m, ξ[1:n])
```

```
@constraint(m, r .== L*z + μ)
if set_type == :Polyhedral
    @constraint(m, norm(z,   1) <= Γ) # ‖z‖₁ ≤ Γ
    @constraint(m, norm(z, Inf) <= 1) # ‖z‖∞ ≤ Γ
else
    @constraint(m, norm(z,   2) <= Γ) # ‖z‖₂ ≤ Γ
end
```

We then link the uncertain parameters $\mathbf{r}$ with decision variables $\mathbf{x}$ and $z$, and solve the model. Internally, JuMPeR will reformulate the problem to a deterministic problem and hand it off to a solver. We return the asset allocation and end the function.

```
@constraint(m, z <= dot(r, x))
solve(m)
return getvalue(x)
end  # function
```

By arranging the code in a function we are able to easily evaluate the model for a variety of parameters, and embed the robust optimization model into complex simulations and software. This also demonstrates the smooth transition between a deterministic model and a robust model: we can simply replace the uncertainty set code with $\mathbf{r} = \boldsymbol{\mu}$ and vice versa, staying with the same modeling and solver infrastructure. This is in contrast to RO-only modeling tools like ROME.

### 6.3.2 Multistage Inventory Control

In our second case, we show the implementation of the multistage inventory control problem described by Ben-Tal et al. [2004]. In this problem, we must determine production levels at each factory $i$ and time period $t$ for a single product across a $T$ period planning horizon. All demand must be satisfied, and there are constraints on the total amount of production at each time period and the amount of inventory we can store. The deterministic optimization model, in the notation of Ben-Tal et al.

[2004], is

$$
\begin{aligned}
\min_{p_i(t),F} \quad & F \\
\text{subject to} \quad & \sum_{t=1}^{T}\sum_{i=1}^{I} c_i(t)p_i(t) \le F \\
& 0 \le p_i(t) \le P_i(t), \qquad \forall i \in \{1,\dots,I\}, t \in \{1,\dots,T\} \\
& \sum_{t=1}^{T} p_i(t) \le Q_i \qquad \forall i \in \{1,\dots,I\} \\
& V_{min} \le v(1) + \sum_{s=1}^{t}\sum_{i=1}^{I} p_i(s) - \sum_{s=1}^{t} d_s \le V_{max} \qquad \forall t \in \{1,\dots,T\},
\end{aligned}
\tag{6.4}
$$

where $p_i(t)$ is the amount produced of the product at a factory $i$ at time $t$, $c_i(t)$ is the unit cost of production at the same, $P_i(t)$ is the production capacity for factory $i$, $Q_i$ is the cumulative production capacity, and $V_{min}$ and $V_{max}$ are the minimum and maximum total inventory limits. In the robust setting the demand $d_t$ is an uncertain parameter, and the production decisions at time $t$ can be made with full knowledge of the demand realized at time periods $1,\dots,t-1$. In Ben-Tal et al. [2004] the demand belongs to a box uncertainty set, and the adaptive production decisions are approximated with an affine adaptability policy, also known as a linear decision rule:

$$
p_i(t) = \beta_{i,t}^{0} + \sum_{r=1}^{t-1} \beta_{i,t}^{r} d_r,
$$

where $\beta_{i,t}^{r}$ are auxiliary variables that define the policy.

We will solve an instance of this model using the same parameters as in Ben-Tal et al. [2004]. As before we must first load JuMP and JuMPeR. We then define the parameters, which is (JuMPeR-independent) standard Julia code.

```
using JuMP, JuMPeR
I = 3           # Number of factories
T = 24          # Number of time periods
# Nominal demand
```

```
d_nom = 1000*[1 + 0.5*sin(π*(t-1)/12) for t∈1:T]
θ = 0.20        # Uncertainty level
α = [1.0, 1.5, 2.0] # Production costs
c = [α[i]*(1 + 0.5*sin(π*(t-1)/12)) for i∈1:I, t∈1:T]
P = 567          # Maximimum production per period
Q = 13600        # Maximumum production overall
Vmin = 500       # Minimum inventory at warehouse
Vmax = 2000      # Maximum inventory at warehouse
v1 = Vmin        # Initial inventory
```

We can now initialize our model and the uncertain parameters, which belong to a simple box uncertainty set where each uncertain parameter falls in a interval.

```
rm = RobustModel()
@uncertain(rm, d_nom[t]*(1-θ) <= d[t=1:T] <= d_nom[t]*(1+θ))
```

To define the adaptive production decisions $p_i(t)$ we can use `Adaptive` variables, which only require that we define what uncertain parameters the variable should be a function of, and the structure of the policy (in this case, affine).

```
@adaptive(rm, 0 <= p[i=1:I,t=1:T] <= P,
             policy=Affine, depends_on=d[1:t-1])
```

We can read the above line of code as "adaptive variable `p[i,t]`, which belongs to the model `rm`, is an affine function of uncertain parameters `d[1]` through `d[t-1]`". We next define an auxiliary variable $F$ to represent the objective function value, and constrain it as in the Equation (6.4) above.

```
@variable(rm, F)  # Overall cost
@objective(rm, Min, F)
@constraint(rm, F >= sum{c[i,t]*p[i,t], i=1:I, t=1:T})
```

Note that while the objective function constraint doesn't explicitly include any uncertain parameters, as `p[i,t]` is a function of `d` this constraint will be handled as an uncertain constraint when transforming the problem later. The remaining tasks are to constrain the total production to respect the cumulative limit, and to ensure we do not exceed the inventory limits. All these constraints are uncertain constraints, as they include `d` explicitly or implicitly (as part of the production decision).

```
for i in 1:I
    @constraint(rm, sum{p[i,t], t=1:T} <= Q)
end
for t in 1:T
    @constraint(rm, v1 + sum{p[i,s], i=1:I, s=1:t}
                        - sum{d[s], s=1:t} >= Vmin)
    @constraint(rm, v1 + sum{p[i,s], i=1:I, s=1:t}
                        - sum{d[s], s=1:t} <= Vmax)
end
```

Finally, we solve the problem. By default the `BasicUncertaintySet` will be used to reformulate the problem, but if we pass the `prefer_cuts=true` option then cutting planes will be used instead.

```
solve(rm, prefer_cuts=true)
println(getobjectivevalue(rm))
```

## 6.3.3 Specialized Uncertainty Set (Budget)

To this point we have demonstrated the use of common polyhedral or ellipsoidal uncertainty sets. However, more exotic uncertainty sets have been proposed in the literature, including the "data-driven" sets described by Bertsimas et al. [2013a]. Many of those sets have complex descriptions that arise from applying different hypothesis tests to the provided data. The choice of test has a large impact on the computational practicality of the set: some tests correspond to simple box sets, while others require exponential cones (which are supported by few solvers). Intriguingly, some of the sets admit very simple cutting plane generation algorithms – either closed-form formulae, or the solution of a line search problem.

As detailed in Section 6.2, JuMPeR defines a simple interface that a researcher can implement for a new set, and can choose to generate cutting planes however is most efficient. Unfortunately, the implementation of the more interesting uncertainty sets in Bertsimas et al. [2013a] is too long for inclusion here. We instead present the implementation of a specialized `BudgetUncertaintySet`, that provides an efficient cutting plane method for the family of uncertainty sets

$$\mathcal{U}(\boldsymbol{\mu}, \boldsymbol{\sigma}, \Gamma) = \left\{ (\boldsymbol{\xi}, \mathbf{z}) \mid \xi_i = \mu_i + \sigma_i z_i, \ \|\mathbf{z}\|_1 \leq \Gamma, \ \|\mathbf{z}\|_\infty \leq 1 \right\}. \tag{6.5}$$

This set was initially proposed by Bertsimas and Sim [2004] (albeit not in this exact form), and the cutting plane method for this set is described in Chapter 2.

To begin defining a new uncertainty set, we must first construct a Julia type that extends (`<:`) JuMPeR's `AbstractUncertaintySet`. This type stores the parameters $\boldsymbol{\mu}$, $\boldsymbol{\sigma}$, and $\Gamma$ that define the set, as well as the violation $\epsilon$ required to add a new constraint.

```
type BudgetUncertaintySet <: JuMPeR.AbstractUncertaintySet
    Γ::Int
    μ::Vector{Float64}
    σ::Vector{Float64}
    ϵ::Float64
end
```

There is a method that we take no action in, but must be defined to complete the interface for our new type: `setup_set`. This is called by the main `solve` function once per uncertainty set to let it know what constraints it has been associated with. As we do not need to do any per-constraint preparation or setup, we simply define a method that does and returns nothing.

```
setup_set(us::BudgetUncertaintySet, ...) = nothing
```

We can now implement the core of the cutting plane method. We will define a function that, given the uncertainty set and an uncertain constraint, will return the values for the uncertain parameters in that constraint that reduce the slack as much as possible. That is, for a constraint

$$\sum_j \left(\boldsymbol{\xi}^T \mathbf{a}_j + \bar{a}_j\right) x_j + \boldsymbol{\xi}^T \mathbf{a}_j^0 \le b, \tag{6.6}$$

find the values of $\boldsymbol{\xi}$ such that the left-hand-side is maximized. To do so, we first rearrange the terms and accumulate the coefficients for each $\xi_i$ at the current value of $\mathbf{x}$ in the master problem, i.e.,

$$\sum_i \left(a_{0,i} + \sum_j a_{j,i} x_j +\right) \xi_i + \sum_j \bar{a}_j x_j \le b. \tag{6.7}$$

As $\boldsymbol{\xi}_i = \mu_i + \sigma_i z_i$, we can further rearrange the constraint to isolate only the components that involve $\mathbf{z}$ – we say that the rest is the nominal part of the constraint:

$$\sum_i \left(a_{0,i} + \sum_j a_{j,i} x_j +\right) \sigma_i z_i + \sum_i \left(a_{0,i} + \sum_j a_{j,i} x_j +\right) \mu_i + \sum_j \bar{a}_j x_j \le b. \tag{6.8}$$

```
function get_worst_case(us::BudgetUncertaintySet, con)
# Collect the coefficients for each uncertain parameter,
# as well as the nominal part
unc_x_vals = zeros(length(us.μ))
nominal_value = 0.0
# For every variable term in the constraint...
for (unc_expr, var) in linearterms(con.terms)
    # Get the value of x_j in the current solution
    x_val = getvalue(var)
    # unc_expr is ξ^T a_j for x_j. We need
    # to iterate over this expression as well.
    for (coeff, unc) in linearterms(unc_expr)
        nominal_value += coeff * us.μ[unc.id] * x_val
        unc_x_vals[unc.id] += coeff * x_val
    end
    # The deterministic part $\bar{a}_j$
    nominal_value += unc_expr.constant * x_val
end
# The $\boldsymbol{\xi}^T \mathbf{a}_j^0$ term
for (coeff, unc) in linearterms(con.terms.constant)
    nominal_value += coeff * us.$\mu$[unc.id]
    unc_x_vals[unc.id] += coeff
end
nominal_value += con.terms.constant.constant
```

We now scale the $\mathbf{x}$ values by the "deviations" $\boldsymbol{\sigma}$, and take the absolute values of the result as we need to rank the uncertain parameters by the magnitude of this quantity. The uncertain parameters with the largest magnitudes should be set to their upper or lower bounds, as doing so has the largest effect on the left-hand-side of the original uncertain constraint.

```
    scaled_vals = abs(unc_x_vals) .* us.$\sigma$
    # Obtain the permutation vector of the indices as if
    # we had sorted by the magnitudes. Take the top $\Gamma$.
    max_inds = sortperm(scaled_vals)[(end - us.$\Gamma$ + 1):end]
```

Given the uncertain parameters we should set to their bounds, we can easily calculate the effect of doing so. We will use this later to determine if we should add a new constraint or not.

```
    cut_value = nominal_value + sum(scaled_vals[max_inds])
```

Finally, we determine the actual values of the uncertain parameters that achieve this value. We determine which bound to by observing the sign of the coefficients on each $\xi_i$: if they are positive, then setting the uncertain parameter to its upper bound should maximize the left-hand-side, and if they are negative then setting the uncertain parameter to its lower bound should do the same.

```
    unc_values = copy(us.$\mu$)
```

```
for i in max_inds
    if unc_x_vals[i] > 0
        unc_values[i] += us.σ[i] # Push up, LHS goes up
    else
        unc_values[i] -= us.σ[i] # Push down, LHS goes up
    end
end
return cut_value, unc_values
end # function
```

Given this function, we can complete the JuMPeR uncertainty set interface. The `generate_cut` function receives a list of constraints and the model, and iterates through these constraints trying to generate new constraints using the function we just defined. Finally, we do not provide reformulation support for this uncertainty set, as it would be no different from the generic duality-based reformulation supported by `BasicUncertaintySet` (which we used by default for the other constraints).

```
function generate_cut(us::BudgetUncertaintySet,
                      rm::Model, idxs::Vector{Int})
# Extract the RobustModelExt from the JuMP model
# This contains all the RO-specific information
rmext = get_robust(rm)::RobustModelExt
# The vector of new constraints we will add
new_cons = Any[]
# For each constraint we need to generate a cut for
for idx in idxs
    # Get the uncertain constraint object
    con = rmext.unc_constraints[idx]
    # Determine worst-case uncertain parameters
    cut_value, unc_values = get_worst_case(us, con)
    # Use a utility function from uncsets_util.jl
    # to check violation status
    if check_cut_status(con, cut_value, us.ϵ) != :Violate
        # No violation, no new cut
        continue  # try next constraint
    end
    # Build a deterministic constraint from the
    # uncertain constraint by filing in values
    new_con = build_certain_constraint(con, unc_values)
    push!(new_cons, new_con)
end
return new_cons
end  # function

generate_reform(us::BudgetUncertaintySet, ...) = nothing
```

## 6.4 Comparisons with Other Tools

JuMPeR is not the first AML that has support for RO. In this section, we describe five alternatives, and contrast their capabilities versus JuMPeR (summarized in Table 6.1). The two most commonly used RO AMLs are YALMIP [Löfberg, 2012] and ROME [Goh and Sim, 2011]. Both are implemented in MATLAB, and support many key features, including polyhedral and ellipsoidal uncertainty set reformulations. There is one commercial package with support for RO, AIMMS, and a further two more experimental frameworks that are implemented in C++: ROPI [Goerigk, 2014] and ROC [Bertsimas et al., 2016].

One of the key features for any AML is usability. AIMMS, as a dedicated standalone modeling language, does well in this regard as it has full flexibility in its syntax. YALMIP and ROME are embedded in MATLAB, but have intuitive syntax that is readable by a novice. Additionally, as MATLAB is a dynamic language, there is no need for complex memory management or a compilation stage. ROPI and ROC are

| Name | JuMPeR | YALMIP | ROME | AIMMS | ROPI | ROC |
|---|---|---|---|---|---|---|
| Availability | Free (MPL2.1) | Free (custom) | Free (GPLv3) | Commercial | Free (MIT) | Free (unknown) |
| Language C++ | Julia | MATLAB | MATLAB | Standalone | C++ | |
| General? | Yes (JuMP) | Yes | No | Yes | No | No |
| Solvers | Many | Many | SDPT3, MOSEK, CPLEX | Many | CPLEX, Gurobi, Xpress | CPLEX |
| Uncertainty sets | Polyhedral, ellipsoidal, custom user-extensible | Polyhedral, ellipsoidal, conic, . . . | Polyhedral, ellipsoidal, DRO | Box, ellipsoidal, convex hull, chance constraints | Scenario sets, “light” robustness | Polyhedral, ellipsoidal, DRO |
| Cutting planes for (MI)LO | Yes | No | No | No | No | No |
| Adaptive optimization | LDR | None | Deflected LDR | LDR | NA | LDR |

Table 6.1: RO modeling language comparison. “Language” refers to the host language for the modeling language. “General” refers to whether the language can be/is intended to be used for a variety of problems, especially deterministic problems. “Cutting planes” refers to whether the language has support for cutting plane approaches to RO, including for robust MIO problems.

both implemented as C++ libraries, which has less elegant syntax, manual memory management, and requires a separate compilation step – all factors which slow prototyping and development. JuMPeR is most similar to YALMIP and ROME, in that it is embedded in a high-level dynamic language and must make some minor syntax concessions as a result. The following snippet demonstrates a simple RO problem in ROME, taken from the ROME documentation:

```
h = rome_begin ('simple');
newvar x y;  % Set up modeling variables
newvar z uncertain;  % scalar uncertainty
rome_box(z , 1.5 , 2.5);  % and its support
rome_maximize(12 * x + 15 * y);  % objective function
rome_constraint(x + z*y <= 40);
rome_constraint(4*x + 3*y <= 120);
rome_constraint(x >= 0);
rome_constraint(y >= 0);
h.solve;
rome_end;
```

The JuMPeR equivalent of this code is as follows:

```
h = RobustModel()
@variable(h, x >= 0)
@variable(h, y >= 0)
@uncertain(h, 1.5 <= z <= 2.5)
@objective(h, Max, 12x + 15y)
@constraint(h, x + z*y <= 40)
@constraint(h, 4x + 3y <= 120)
solve(h)
```

Related to usability is the notion of generality. By this, we mean that a user can use the AML for more than just RO. All the tools here can be used for deterministic problems, but some were made solely for the purpose. We would suggest that a language that is solely made for the purposes of RO will generally be less suitable for deterministic problems than a language that is more general by design. In particular, we claim that YALMIP, JuMPeR and AIMMS are all have this property of generality, with JuMPeR inheriting all the features of JuMP. In all three of these tools, a user can begin with a deterministic model before smoothly transitioning to a RO model. This correlates with solver support: these three languages support a very wide variety of solvers, as they all have a base infrastructure for communicating with them that is painful to build for a dedicated RO modeling tool.

Finally, the six languages all vary significantly in what aspects of RO and adaptive RO they support. JuMPeR is the only language with first-class support for cutting planes so far. All but ROPI support polyhedral and ellipsoidal sets, while only ROME, AIMMS, and ROC have support for "distributionally robust" sets built in. In terms of more complicated uncertainty sets, JuMPeR's general framework for uncertainty sets and YALMIPs very general reformulation capabilities stand apart from the rest. Finally, JuMPeR, ROME, AIMMS, and ROC all support linear decision rules, with ROME going a step further and offering deflected linear decision rules as well.

We conclude from this comparison that JuMPeR is a valuable contribution to the landscape of RO modeling tools. It is particularly suited for problems that are linear in the decision variables, and for adaptive optimization problems. The cutting plane support makes it the best choice for many kinds of RO problems, and a good platform for conducting research into new uncertainty sets.