JuMPeR.jl Documentation

Release 0.0

Iain Dunning

March 02, 2014

Contents

CHAPTER 1

Introduction

1.1 What is JuMP?

JuMP is a domain-specific modeling language for mathematical programming embedded in Julia. It currently supports a number of open-source and commercial solvers (Clp, Cbc, GLPK, Gurobi, MOSEK, and CPLEX) via a generic solver-independent interface provided by the MathProgBase package.

This manual will assume you are familiar with JuMP already. If you are not, please read the JuMP manual first.

1.2 What is Robust Optimization?

Robust optimization is one way to address optimization under uncertainty. Uncertainties in the problem are modeled as lying inside convex sets, and feasible solutions to a robust optimization problem must be feasible for all scenarios that lie in these uncertainty sets. There are computationally tractable ways to solve these problems, although implementation of these can be tedious and/or brittle.

1.3 What is JuMPeR?

JuMPeR stands for "Julia for Mathematical Programming - extensions for Robust". It builds on the groundwork laid by JuMP by adding a new type, `Uncertain`, and allowing users to express robust optimization problems in a natural way. JuMPeR handles both reformulations of the problem into certain equivalents, or manages cutting-plane methods to solve the problems iteratively. Users can extend JuMPeR by providing *oracles* that implement reformulations and cutting-plane generation algorithms for new uncertainty sets.

CHAPTER 2

Robust Modeling

2.1 Setup and JuMP

Note: JuMPeR is under active development. Some names may change in the future, and should be considered temporary.

We first introduce a new model type, `RobustModel`, that we will use as we previously used `Model`. We can mostly use the same commands you use with a plain JuMP `Model`: `@defVar`, `setObjective`, `addConstraint`. The main exceptions are that you must use `solveRobust` to solve the model, and `printRobust` to display the model. The `Variable` type can be created and used as it is used in plain JuMP.

2.2 The Uncertain Type

The first major addition is the `Uncertain` type, which represents a coefficient or scalar in the problem that is uncertain. It behaves much like a `Variable`, in that it can be combined with other ‘‘Uncertain‘‘s and ‘‘Variable‘‘s in expressions to form constraints. When modeling robust optimization problems, the following restrictions apply at this time:

- No quadratic terms - e.g. no `Variable * Variable` terms, no `Uncertain * Uncertain` terms. `Uncertain * Variable` is allowed.
- No uncertain terms in the objective. The correct approach is to create an auxiliary variable that represents the objective value and transform the original objective into a constraint. This is subject to change.
- No macros for constraints using uncertainties. For now, the non-macro version, `addConstraint`, must be used. The other implication is that the convenience `sum{}` syntax is not available - instead the Julia `sum([...])` must be used.

Uncertains can be created using `@defUnc` similar to how variables are created, with the main difference being that uncertainties do not have a type (i.e. defining integer-restricted uncertainties is not supported at this time.)

2.3 First Simple Example

Here we will solve a toy model with two variables and two uncertainties. The two uncertains live in a range and are not connected in anyway.:

```
# Load JuMPeR, which also loads JuMP
using JuMPeR

# We need to use RobustModel, not Model
m = RobustModel()

# @defVar is the same as in a plain JuMP model
@defVar(m, x[1:2] >= 0)

# @defUnc is very similar to @defVar
# The bounds on our uncertains completely define the
# uncertainty set in this toy problem
@defUnc(m, 0.3 <= u1 <= 0.5)
@defUnc(m, 0.0 <= u2 <= 2.0)

# We will use the non-macro version of setObjective, although
# we could use the macro version if we wanted to.
# Remember: objectives must be certain.
setObjective(m, :Max, x[1] + x[2])

# We now add constraints as we normally would, with a few
# variations. Notice we are multiplying an uncertain and a
# variable (e.g. u1*x[1]). We also have to use the non-macro
# versions of addConstraint. The solution will ensure
# feasibility for all realizations of u1 and u2
addConstraint(m, u1*x[1] + 1*x[2] <= 2.0)
addConstraint(m, u2*x[1] + 1*x[2] <= 6.0)

# Solve the model - notice solveRobust in place of solve
# Also notice we did not say how it should be solved! We'll
# discuss this later.
status = solveRobust(m)

# Display the solution - just like JuMP
println(getValue(x[1]))  # = 2.6666
println(getValue(x[2]))  # = 0.6666
```

2.4 Expression Types

Note: It is completely possible to use JuMPeR without understanding the details of this section. However, if you wish to develop custom uncertainty sets not handled by an existing *oracle*, you will need to.

In base JuMP we have the `AffExpr` type which represents an affine expression of variables, e.g. `3x + 2y + 3`. To handle uncertainties in this framework we need to introduce new types of affine expression. By combining uncertainties, variables and numbers we obtain the following new types:

- `UAffExpr` - an affine expression containing only numbers and uncertains, e.g. `3u + 4v <= 5` where `u` and `v` are of type `Uncertain`.
- `FullAffExpr` - an affine expression containing both `UAffExpr``s and variables, e.g. ``(3u)*x + (4v + 2)*y + (w + 0) <= 0` where `u, v, w` are uncertain (thus this constraint can be said to have uncertain right-hand-side).

Another way to think about these (and is indeed how it is internally handled) is that we define an affine expression by its coefficient type, and the "things" the coefficients are multiplied with. That is (loosely):

```
AffExpr       === GenericAffineExpression{Number,   Variable }
UAffExpr      === GenericAffineExpression{Number,   Uncertain}
FullAffExpr   === GenericAffineExpression{UAffExpr, Variable }
```

For the precise details, the best reference is the JuMP and JuMPeR source code, as well as the existing oracles.

CHAPTER 3

Robust Optimization

3.1 Solving robust problems

Multiple methods exist in the literature for solving robust optimization problems, and it is beyond the scope of this manual to go into too many details. The main methods are:

- Reformulation: take uncertain constraints and replace them with new constraints and possibly new variables to form a certain problem. Usually exploits duality properties of the uncertainty set.
- Cutting-plane: solve a relaxed version of the problem (i.e. with no uncertain constraints). Given the solution to this relaxed problem, try to generate new constraints that will make the current solution infeasible - usually by solving a optimization problem in the space of the uncertainty set.

Other things to consider are sampling a set of constraints from the uncertainty set, partially generating reformulations, or really whatever you can think of. The important thing is that at the end of the solve, the problem must be feasible with respect to all constraints. In a way we can think of "reformulation" as operations performed before any solving, and "cutting-plane" as operations performed during solve.

3.2 Oracles

JuMPeR seperates the reformulation and cutting-plane logic into *oracles*. Oracles can be thought of as "robustifying operators" that take an uncertain problem and ensure the solution will be robust. At various points during the solve, JuMPeR will interact with the oracles associated with the constraints to coordinate reformulations, cutting-planes, etc. JuMPeR comes with multiple oracles, and users of JuMPeR can make and share their own - allowing users to easily swap and try out new uncertainty sets with little fuss.

Oracles can be associated with one or more constraints. By default all constraints use the inbuilt `PolyhedralOracle` that, as the name suggests, handles reformulations and cutting planes when the uncertainty set is polyhedral. We can alternatively specify a particular constraint by passing a third argument to `addConstraint`, i.e.

```
addConstraint(m, a*x + b*y <= 4, MyNewOracle())
shared_oracle = MyOtherOracle()
addConstraint(m, c*x >= 2, shared_oracle)
addConstraint(m, d*z >= 3, shared_oracle)
```

3.3 Creating an Oracle

To create an oracle one must understand the structure of the solving algorithm inside JuMPeR's `solveRobust`

1. A new model, referred to as the *master*, is created with the originals variables and certain constraints.
2. Each oracle is notified of the constraints it must handle using `registerConstraint`.
3. Each oracle is given time to do any general setup, now that is is aware of what it must do. For example, it may take the dual of the uncertainty set in order to more efficiently reformulate multiple constraints.
4. Oracles that want to will now reformulate their constraints (or, alteratively, generate samples).
5. The master problem will now be solved.
6. Oracles can now use the master solution to generate new constraints (cutting-planes). If no oracles add constraints, we end the solution process. Otherwise, we re-solve.

We will now detail the four functions an oracle must provide.

GO HERE

CHAPTER 4

Built-in Oracles

4.1 PolyhedralOracle

ggg

4.2 BertSimOracle

ggg

CHAPTER 5

Worked Example: Portfolio Optimization

We will now consider a worked example using a polyhedral uncertainty set. We are trying to allocate percentages of the money in our portfolio to different assets. We are provided a matrix where the rows correspond to monthly returns for each asset, and the columns are the returns for each asset. Our goal is solve a robust optimization problem that maximizes the return of the porfolio over the worst case in the uncertainty set. We will obtain less conservate solutions by exploiting the covariance information stored in the returns matrix to construct a polyhedral uncertainty set.

The particular robust optimization model we will solve is as follows:

```
# max   obj
# s.t.  sum(x_i      for i in 1:n) == 1
#       sum(r_i x_i  for i in 1:n) >= obj
#       x_i >= 0
# where x_i is the percent allocation for asset i, and r_i is the
# uncertain return on asset i.
#
# Uncertainty set:
#       r = A z + mean
#       y = |z|
#       sum(y_i for i in 1:n) <= Gamma
#       |z| <= 1, 0 <= y <= 1
# where A is such that A*A' = Covariance matrix
```

Let begin by creating our function, `solve_portfolio`:

```
function solve_portfolio(past_returns, Gamma, pref_cuts)

# Create covariance matrix and mean vector
covar = cov(past_returns)
means = mean(past_returns, 1)

# Idea: multivariate normals can be described as
# r = A * z + mu
# where A*A^T = covariance matrix.
# Instead of building uncertainty set limiting variation of r
# directly, we constrain the "independent" z
A = round(chol(covar),2)
```

Note that our function takes three arguments: the matrix of past observed returns `past_returns`, the uncertainty set "size" `Gamma`, and `pref_cuts` which will be a Boolean that determines whether we solve with cuts or reformulation. We will also assume that a constant is defined globally, `NUM_ASSET`, that is the number of assets available to chose from.:

```
# Setup the robust optimization model
m = RobustModel(solver=GurobiSolver(OutputFlag=0))

# Variables
@defVar(m, obj)  # Put objective as constraint using dummy variable
@defVar(m, x[1:NUM_ASSET] >= 0)
```

Here we set up our model with Gurobi selected as the solver, with output suppressed. JuMPeR doesn't support uncertain objectives, so we will make the objective a constraint.:

```
# Uncertainties
@defUnc(m,       r[1:NUM_ASSET]      )  # The returns
@defUnc(m, -1 <= z[1:NUM_ASSET] <= 1 )  # The "standard normals"
@defUnc(m,  0 <= y[1:NUM_ASSET] <= 1 )  # |z|/box

@setObjective(m, Max, obj)

# Portfolio constraint
addConstraint(m, sum([ x[i] for i=1:NUM_ASSET ]) == 1)

# The objective constraint - uncertain
addConstraint(m, sum([ r[i]*x[i] for i=1:NUM_ASSET ]) - obj >= 0)
```

The rest of the model structure follows naturally from the mathematical definition. Note the objective-as-a-constraint, and the use of `sum([])` instead of `sum{}`. Next we will define the uncertainty set, which is notable only in that the constraints only involve uncertainties and numbers - no variables.:

```
# Build uncertainty set
# First, link returns to the standard normals
for asset_ind = 1:NUM_ASSET
    addConstraint(m, r[asset_ind] ==
        sum([ A[asset_ind, j] * z[j] for j=1:NUM_ASSET ]) + means[asset_ind] )
end
# Then link absolute values to standard normals
for asset_ind = 1:NUM_ASSET
    addConstraint(m, y[asset_ind] >= -z[asset_ind] / box)
    addConstraint(m, y[asset_ind] >=  z[asset_ind] / box)
end
# Finally, limit how much the standard normals can vary from means
addConstraint(m, sum([ y[j] for j=1:NUM_ASSET ]) <= Gamma)
```

We now have everything we need to solve the problem. We are using the default `PolyhedralOracle`, and can choose whether we use cutting planes or reformulation. This can done with the `prefer_cuts` option for `solveRobust`. Finally, we will return the portfolios allocation.:

```
solveRobust(m, prefer_cuts=pref_cuts)

return getValue(x)

end  # end of function
```

To evaluate this code, we will generate synthetic correlated data. The following code generates that data use a common market factor random normal combined with idiosyncratic normals for each asset:

```
##########################################################################
# Simulate returns of the assets
# - num_samples is number of samples to take
# - Returns matrix, samples in rows, assets in columns
##########################################################################
```

```
function generate_data(num_samples)
    data = zeros(N, NUM_ASSET)

    # Linking factors
    beta = [(i-1.)/NUM_ASSET for i = 1:NUM_ASSET]

    for sample_ind = 1:num_samples
        # Common market factor, mean 3%, sd 5%, truncate at +- 3 sd
        z = rand(Normal(0.03, 0.05))
        z = max(z, 0.03 - 3*0.05)
        z = min(z, 0.03 + 3*0.05)

        for asset_ind = 1:NUM_ASSET
            # Idiosyncratic contribution, mean 0%, sd 5%, truncated at +- 3 sd
            asset = rand(Normal(0.00, 0.05))
            asset = max(asset, 0.00 - 3*0.05)
            asset = min(asset, 0.00 + 3*0.05)
            data[sample_ind, asset_ind] = beta[asset_ind] * z + asset
        end
    end

    return data
end
```

Let us evaluate the distribution of results for different levels of conservatism. We will generate a "past" dataset we can optimize over, and a "future" dataset we will evaluate on. Let us try the following code:

```
past_returns   = generate_data(1000)
future_returns = generate_data(1000)

function eval_gamma(Gamma)
    x = solve_portfolio(past_returns, 1, true)
    future_z = future_returns * x[:]
    sort!(future_z)
    println("Selected solution summary stats for Gamma $Gamma")
    println("10%:     ", future_z[int(NUM_FUTURE*0.1)])
    println("20%:     ", future_z[int(NUM_FUTURE*0.2)])
    println("30%:     ", future_z[int(NUM_FUTURE*0.3)])
    println("Mean:    ", mean(future_z))
    println("Maximum: ", future_z[end])
end

eval_gamma(0)  # Nominal - no uncertainty
eval_gamma(3)  # Some protection
```

If we evaluate this code, we build the following table:

Gamma	0	3
10%	-5.89%	-2.13%
20%	-2.95%	-1.10%
30%	-0.83%	-0.34%
Mean	2.49%	1.07%
Max	21.86%	10.45%

j

j