

Users' Guide to ROME:  
*Robust Optimization Made Easy*  
Version 1.0.x (beta)

Joel Goh          Melvyn Sim

17 Sept 2009

## Contents

<b>1</b>	<b>Introduction</b>	<b>3</b>
1.1	Introduction to Robust Optimization . . . . .	3
1.2	What ROME Does . . . . .	3
<b>2</b>	<b>Installation Instructions</b>	<b>4</b>
2.1	Installing ROME . . . . .	4
2.2	Underlying Solver Engines . . . . .	4
2.3	OS requirements and MATLAB Version . . . . .	5
<b>3</b>	<b>Basics</b>	<b>5</b>
3.1	Basic Structure of a ROME Program . . . . .	5
3.2	The Model Object . . . . .	6
3.3	Creating New Variables . . . . .	7
3.4	Variable Assignment . . . . .	8
3.5	Common Properties of <code>rome_var</code> objects . . . . .	9
3.6	Operators and Functions on ROME objects . . . . .	9
3.7	Constraints . . . . .	10
3.8	Objective . . . . .	10
<b>4</b>	<b>Uncertainties and Linear Decision Rules</b>	<b>10</b>
4.1	Uncertainties . . . . .	11
4.1.1	Support . . . . .	11
4.1.2	Mean Support . . . . .	11
4.1.3	Covariance Matrix . . . . .	12
4.1.4	Directional Deviations . . . . .	12
4.2	Example: Robust Counterpart . . . . .	13
4.3	Linear Decision Rules . . . . .	13
<b>5</b>	<b>Deflected Linear Decision Rules</b>	<b>15</b>
5.1	Advanced Deflection Options . . . . .	16
5.2	List of Robust Bounds . . . . .	16
5.3	Using Robust Bounds Directly . . . . .	17
5.4	Combining Robust Bounds . . . . .	18

<b>6</b>	<b>Analysis of Optimization Results</b>	<b>18</b>
6.1	Getting the Objective Function Value . . . . .	18
6.2	Using <code>eval</code> to Get Solutions . . . . .	19
6.3	List of <code>rome_sol</code> Members . . . . .	20
6.4	Using <code>insert</code> for Monte Carlo Simulation . . . . .	22
<b>7</b>	<b>Solver Options</b>	<b>23</b>
7.1	Polling the Solvers . . . . .	23
7.2	Choosing a Solver . . . . .	23
7.2.1	At <code>solve</code> Time . . . . .	23
7.2.2	Change Default Solver . . . . .	23
7.2.3	Change Default Solver Forever . . . . .	24
7.3	Controlling Output Verbosity . . . . .	24
<b>8</b>	<b>Acknowledgements</b>	<b>24</b>
<b>A</b>	<b>List of Operators and Functions</b>	<b>26</b>
A.1	Operators . . . . .	26
A.2	Standard Functions . . . . .	26
<b>B</b>	<b>Expert Variable Creation Drivers</b>	<b>28</b>
B.1	Syntax for <code>rome(_rand)_model_var</code> . . . . .	28
B.2	Syntax for <code>rome_linearrule</code> . . . . .	28

# 1 Introduction

## 1.1 Introduction to Robust Optimization

ROME is a MATLAB toolkit for modeling and solving robust optimization (RO) problems. Robust optimization techniques were originally (see Soyster [12], Ben-Tal and Nemirovski [3], and Bertsimas and Sim [4]) used to model and solve optimization problems where the modeler desired to guard the nominal problem against infeasibility caused by uncertainties in the problem data. More modern advances in robust optimization, driven by the introduction of the Affinely-Adjustable Robust Counterpart (AARC) by Ben-Tal et al. [2], allowed robust optimization models to be used as tractable approximations of stochastic programs.

However, in contrast to stochastic programming, where uncertainties are typically modeled as having known distributions, robust optimization models usually contain uncertainties which have distributions that are not precisely known, but are instead confined to some family of distributions based on known distributional properties. To ensure tractability, recourse decisions in robust optimization models are usually restricted to affine functions of the uncertainties, sometimes termed Linear Decision Rules (LDRs) authors. Several recent works (e.g. Chen, Sim, and Sun [6], Chen et al [7], Chen and Zhang [8], See and Sim [11]) introduced more complex decision rules which further improve upon the basic LDR, with varying degrees of modeling and computational complexity.

## 1.2 What ROME Does

While robust optimization problems might be easy to express mathematically, the actual computation of the solution of such problems is seldom trivial. To this end, we designed ROME to be a modeling language embedded in the MATLAB environment, to provide a computational platform for researchers in robust optimization to ease the process of building, testing, and benchmarking of robust optimization models. Using ROME, modelers can easily and flexibly model a variety of robust optimization problems in a mathematically intuitive manner. The current version (1.0.x (beta)) of ROME can model and solve robust optimization problems using standard LDRs as well as more modern (and complex) decision rules such as Deflected LDRs (see Chen et al [7], Goh and Sim [9]).

We emphasize that ROME is *not* a solver on its own, but rather a modeling language, which uses the rules and techniques of robust optimization to convert uncertain optimization problems into deterministic forms, which are then fed as inputs into third-party solver engines to perform the actual computations. While ROME is designed to solve robust optimization problems, i.e. optimization problems contaminated with uncertainties, ROME can also be used to model and solve deterministic convex programs up to a complexity of Second-Order Conic Programs (SOCP). However, ROME is purposefully optimized to solve general deterministic convex programs. We will discuss deterministic convex programs in this guide only to serve as an illustration of ROME's modeling concepts and structures.

## 2 Installation Instructions

### 2.1 Installing ROME

ROME can be freely downloaded from: <http://www.robustopt.com/>.

1. Upon downloading ROME, you should have a zipped file. Unpack the zipped-file into a directory of your choice.
2. Start MATLAB. Upon startup, switch to the directory that you unpacked the files into. This should be the directory that contains the file `rome_setup.m`
3. Run `rome_setup.m`
4. If the installation is successful, you should see a message similar to the one below

```
ROME: Robust Optimization Made Easy  
Version: 1.0.x
```

```
ROME successfully installed!
```

5. This will put the ROME directories in MATLAB's search path for the current session. However, the next time you start MATLAB, you will have to run `rome_setup.m` again. If you are a frequent ROME user, you can configure MATLAB to run this startup file upon beginning each session. Please refer to the MATLAB documentation on how to do this.

### 2.2 Underlying Solver Engines

In this version of ROME, version (1.0.x (beta)), the supported solver engines are the freeware SDPT3 solver (by Toh, Todd, and Tütüncü [13, 14]), as well as the commerical CPLEX and MOSEK solvers.

The solvers will have to be installed separately:

1. SDPT3: <http://www.math.nus.edu.sg/~mattohkc/sdpt3.html>
2. MOSEK: <http://www.mosek.com/>
3. CPLEX: <http://www.ilog.com/products/cplex/>

SDPT3 and MOSEK have pre-built MATLAB interfaces, and upon installing both solvers, you would simply have to add the relevant directories to the MATLAB path. Please see specific installation instructions from the solver authors/vendors. The standard installation of the CPLEX Solver does not come with a MATLAB interface, but ROME comes packaged with CPLEXINT[1], a MATLAB interface for CPLEX. All you will have to do is to install CPLEX on your machine and properly add the directory of the `cplexint.m` file to the MATLAB path.

## 2.3 OS requirements and MATLAB Version

ROME was developed on a 32-bit Windows OS, running Windows XP SP3, and with MATLAB Version 7.8 (2009a). We are currently in the process of testing ROME on various OS platforms and versions of MATLAB. We believe that any version of MATLAB above and including the MATLAB 7.6 releases will work.

Older versions of MATLAB will not work because ROME uses some object-oriented paradigms which were only fully available in MATLAB 7.6. If you require support for older versions of MATLAB, please contact the authors to discuss alternatives.

## 3 Basics

### 3.1 Basic Structure of a ROME Program

Each ROME block will have to be prefaced with a call of `rome_begin` to properly set up the ROME environment. Next, you will need to construct a model object, and assign a handler to it. The model object stores the state (decision variables, uncertainties, objective, constraints) for the current optimization model, and Section 3.2 details how the model object can be created.

After instantiating the ROME environment and constructing the model object, you will typically input the main body of code, which corresponds to your mathematical model. All variables in ROME are objects of the class `rome_var` and can either be deterministic, uncertain, or linear decision rules (LDR). Section 3.3 describes how variables can be declared, while sections 3.5 and 3.6 discuss common properties, operations, and functions that can be applied to `rome_var` objects.

Uncertain variables and LDRs are variable types which are more peculiar to programming in ROME, and we discuss them separately in section 4. In particular, section 4.1 discusses how uncertain variables can be declared in ROME and how you can set their distributional properties, while section 4.3 discusses LDR creation.

Within most optimization models, it is extremely common to have constraints on some or all of the variables. Deterministic and robust constraints are input in ROME using `rome_constraint`, and are discussed in more detail in section 3.7. Finally, section 3.8 details various ways describe to your model objective in ROME.

To indicate to ROME that your model is complete and that you would like ROME to solve the model, you will have to call the `solve` function, a member function of the `rome_model` class. After the call to `solve`, subsequent calls to the member function `eval` will allow you to access the solution of the optimization. Section 6 furnishes details of this process.

Finally, a call to `rome_end` clears up the environment and the current model object. Any call to access the solution state or the model state must be made before this final step. The final step is optional, as you might want to retain the program state for debugging and other analyses.

In the example which follows, we illustrate how to use ROME to solve a simple deterministic LP. The subsequent sections provide a more detailed reference on the specifics of how to declare variables, input constraints, etc.

## Examples

We will illustrate how to solve a simple linear program with ROME. This example and all examples in this guide can be found in the `+RomeExamples` directory of the package. Let us solve the linear program:

$$\begin{aligned} \max \quad & Z = 12x + 15y \\ \text{s.t.} \quad & x + 2y \leq 40 \\ & 4x + 3y \leq 120 \\ & x, y \geq 0 \end{aligned}$$

This can be done in ROME via the simple program:

```
1 % EX_SIMPLE_LP.m
2 % Script file for solving a simple Linear Program.
3
4 rome_begin; % Set up ROME Environment
5 h = rome_model('Simple LP'); % Create Rome Model
6 newvar x y; % Set up modeling variables
7
8 % set up objective function
9 rome_maximize(12 * x + 15 * y);
10
11 % input constraints
12 rome_constraint(x + 2*y <= 40);
13 rome_constraint(4*x + 3*y <= 120);
14 rome_constraint(x >= 0);
15 rome_constraint(y >= 0);
16
17 % solve and extract solution values
18 h.solve;
19 x_val = h.eval(x);
20 y_val = h.eval(y);
21
22 rome_end; % Clear up ROME environment
```

Code Segment 1: Simple LP

## 3.2 The Model Object

Calling the function `h = rome_model('model_name')` constructs a model object internally, and returns a handle `h` to the model object. `'model_name'` is an optional argument which should be a descriptive model name. If omitted, ROME will automatically assign a name to the model. Subsequent calls to `rome_get_current_model` will return the same handle.

`rome_begin` has been overridden to create a model object by default if supplied with a single string argument instead of the default no-argument call. If supplied with a single string argument, the call to `rome_begin` will output a handle to the newly-created model object. For example, the following code segment:

```
1 rome_begin;
2 h = rome_model('My Model');
```

Code Segment 2: Starting the ROME Environment

is identical to the shortcut call:

```
1 h = rome_begin('My Model');
```

Code Segment 3: Starting the ROME Environment (Contraction)

### 3.3 Creating New Variables

Modeling variables in ROME have to first be declared before use. They can be declared in simple mode and expert mode. Creating variables in expert mode is discussed in Appendix B. In simple mode, all variable declarations are done by the `newvar` command, which in turn, can be used in two ways – (1) as a command, and (2) as a function. In general, the syntax to declare variables is, in command mode:

```
newvar variable_name(variable_size) opt1 opt2 ...;
```

The `variable_size` argument can either be a comma-separated-list, or a row vector containing the size in each dimension. In function mode,

```
variable_name = newvar(variable_size, 'opt1', 'opt2', ...);
```

The allowable options fall into the following 3 categories:

**Type** Declares variable to be of a given type. Allowed values: `'uncertain'`, `'linearrule'`, `'empty'`.

**Cone** Restricts the variable to the specified convex cone. Allowed values: `'nonneg'`, `'lorentz'`.

**Continuity** Restricts continuity of the variable. Allowed values: `'integer'`, `'binary'`.

#### Examples

To declare a deterministic (termed “certain” in ROME) matrix variable `x` of size 20 by 30, we can call either of:

```
1 newvar x(20, 30); % Can declare either way
2 x = newvar(20, 30);
```

Code Segment 4: Declaring Certain Variables

**Note:** We adopt a slightly different convention from standard MATLAB when declaring a variable with a single argument. Consider the example of a variable declared as `newvar x(5)`. In MATLAB, you might expect the declaration to represent a  $5 \times 5$  matrix. In ROME, however, this creates a  $5 \times 1$  *column* vector.

Uncertain variables (discussed in more detail in Section 4) can be created in the same way. To create an uncertain variable `z` which is a column-vector of size 6, we call either of:

```

1 newvar z(6) uncertain;
2 z = newvar(6, 'uncertain');

```

Code Segment 5: Declaring Uncertainties

To create a Linear Decision Rule (LDR) variable, we will first need to have already created a uncertain variable. Suppose  $z$  was created as above with a length of 6. Then, to create a linearrule variable  $y$ , such that

$$y(z) = y^0 + \sum_{i=1}^6 z_i y^i, \quad z \in \mathbb{R}^6, \quad y(z) \in \mathbb{R}^{5 \times 3}$$

We can call either the command or function forms:

```

1 y(5, 3, z) linearrule;
2 y = newvar(5, 3, z, 'linearrule');

```

Code Segment 6: Declaring LDRs

For the rest of this guide, for brevity, we will alternate between the equivalent function and command forms of the `newvar` keyword.

### 3.4 Variable Assignment

Intermediate values of computation can be assigned to variables, which can make code more efficient and more readable. These variables do not have to be declared prior to use. Suppose you wish to express the constraint set

$$P = \{x \in \mathbb{R}^n : l \leq Ax + b \leq u, \|Ax + b\|_2 \leq t\}$$

you could use the rather inefficient lines of code:

```

1 rome_constraint(A*x + b >= l);
2 rome_constraint(A*x + b <= u);
3 rome_constraint(norm2(A*x + b) <= t);

```

Code Segment 7: Declaring Constraints (Inefficient)

or, more efficiently, using variable assignment

```

1 y = A*x + b; % Assign to a variable first
2 rome_constraint(y >= l);
3 rome_constraint(y <= u);
4 rome_constraint(norm2(y) <= t);

```

Code Segment 8: Declaring Constraints (More Efficient)

Notice that this is an *assignment*, in other words, any previous value (numerical values or `rome_var` objects) stored in  $y$  will be overwritten. This is in contrast to the following section of code, which declares an auxilliary variable  $y$  and inserts an equality constraint:

```

1 newvar y(m); % Declare a variable y
2 rome_constraint(y == A*x + b); % Impose equality constraint

```

Code Segment 9: Auxillary Variables

In this case,  $y$  remains as a `rome_var` object, and its value is not overridden by the constraint.

### 3.5 Common Properties of rome\_var objects

All variables in ROME, whether explicitly constructed or auxilliary, have properties which can be extracted to provide more detailed information about the particular object. These properties can be gotten using the standard property accessor ('.'). You should *NOT* attempt to change these properties directly. The most commonly accessed properties are listed here:

**TotalSize** Returns a positive integer indicating the total number of elements in the `rome_var` object. Use this instead of the `numel` function.

**Size** Returns the size of the `rome_var` object as a row vector.

**IsCertain** Returns a boolean flag indicating if the `rome_var` object is a pure certain object (no uncertain components)

**IsRand** Returns a boolean flag indicating if the `rome_var` object is a pure uncertain object (no certain components)

**IsLDR** Returns a boolean flag indicating if the `rome_var` object is a linearrule object (contains both uncertain and certain components)

Note that the only type of `rome_var` objects which have true `IsCertain` and `IsRand` properties are `rome_var` objects having constant values. The complete list of properties are listed in the expert drivers in Appendix B.

### 3.6 Operators and Functions on ROME objects

The standard MATLAB arithmetic operators on numeric types have been overloaded to operate on ROME objects. Also, certain functions have been overloaded to work with ROME objects as well. See Appendix A for a complete list of operators, functions and their descriptions.

Non-linear convex functions (such as `norm1`, `exp`, etc) on `rome_var` objects are implemented in epigraph form. For a scalar convex function represented in ROME as `convex_func`, calling `t = convex_func(x)` essentially calls

```
1 newvar t;  
2 rome_constraint(convex_func(x) <= t);
```

Code Segment 10: Convex Functions in ROME

Conversely, non-linear concave functions (such as `hypoquad`) on `rome_var` objects are implemented in hypograph form. A scalar concave function function represented in ROME as `concave_func`, calling `t = concave_func(x)` essentially calls

```
1 newvar t;  
2 rome_constraint(concave_func(x) >= t);
```

Code Segment 11: Concave Functions in ROME

In general, we do not allow products of variables, since we cannot generally guarantee the convexity or concavity of a general product of two variables. Thus, even though we know  $\mathbf{x}' * \mathbf{x}$  to be convex, ROME will reject the statement. Instead, we have provided a function, `sq(x)` which performs the same function, but with the guarantee of convexity.

The only notable exception is the product  $\mathbf{z} * \mathbf{x}$ , or  $\mathbf{z} .* \mathbf{x}$ , where  $\mathbf{z}$  is a (pure) uncertain variable and  $\mathbf{x}$  is a (pure) certain variable. For correctly sized inputs, the product  $f(x, z) = zx$  is actually bilinear in  $(x, z)$  and thus will be accepted by ROME. The return type of the variable will then be a *Linear Decision Rule* (LDR) variable. Please see Section 4 for more details about LDR variables.

### 3.7 Constraints

Constraints in ROME are input via the `rome_constraint` function. This function operates on inequalities and equalities of certain, uncertain, and linearrule variables.

#### Examples

Suppose the certain variable  $\mathbf{x}$ , the uncertain variable  $\mathbf{z}$ , and the linearrule variable  $\mathbf{y}$  are as declared in the preceding section. To input deterministic constraints given by  $\{x \in \mathbb{R}^n : Cx = d, Ax \leq b, x \leq u\}$ , where the inequality signs ( $\leq$ ) represent component-wise inequality, we can issue the following commands in ROME:

```
1  rome_constraint(C * x == d);
2  rome_constraint(A * x <= b);
3  rome_constraint(x <= u);
```

Code Segment 12: Linear Constraints

We can also apply the `rome_constraint` function to describe the support of an uncertain variable  $\mathbf{z}$ ,  $\text{supp}(z) = [z, \bar{z}]$ , by the following lines:

```
1  rome_constraint(z >= z_lo);
2  rome_constraint(z <= z_hi);
```

Code Segment 13: Uncertainty Support

Alternatively, you could call the contraction `rome_box(z, z_lo, z_hi)`.

### 3.8 Objective

The optimization objective can be added to the model via a call to either of `rome_minimize` or `rome_maximize`. The argument to either function can be any well-formed scalar-valued expression in ROME.

## 4 Uncertainties and Linear Decision Rules

ROME defines two other types of variables, uncertain variables and linear decision rule (LDR) variables. These variables behave similarly to deterministic variables, in that the basic arithmetic operations (addition, scalar multiplication, sizing functions) can also be applied to them. They also have some unique functions which are listed in this section

## 4.1 Uncertainties

In ROME, we can specify various distributional properties of declared model uncertainties. In this version of ROME, we can specify the (1) support, (2) mean support, (3) covariance matrix, and (4) directional deviations of the model uncertainties. These distributional properties are used to define a family of distributions, and are used in constructing robust bounds.

### 4.1.1 Support

As stated in Section 3.7 the support of a uncertain variable is defined by supplying inequality constraints using `rome_constraint`. Besides box-type supports, we can also specify more general polytopic supports using linear (in)equalities.

For example, to denote a support of the form  $\mathcal{W} = \{z \in \mathbb{R}^6 : \sum_{i=1}^6 z_i = 1, z_i \geq 0\}$ , we can specify:

```
1 z = newvar(6, 'uncertain', 'nonneg');
2 rome_constraint(sum(z) == 1);
```

Code Segment 14: Specifying the Uncertainty Support

Another common type of support is an ellipsoidal uncertainty set (Ben-Tal and Nemirovski [3]). We can declare this in ROME using the `norm2` function, as

```
1 % Assuming A is a 6 x 6 matrix
2 z = newvar(6, 'uncertain', 'nonneg');
3 rome_constraint(norm2(A*z) <= 1);
```

Code Segment 15: Specifying an Ellipsoidal Uncertainty Set

### 4.1.2 Mean Support

In ROME, we do not require the mean of model uncertainties to be deterministically specified. Instead, users can specify the support of the uncertainty means in a similar manner to its support. The following code segment shows how uncertainty means can be specified to lie within a hypercube between -1 and 1 in each component. The code segment shows two methods of getting the mean of `z`, using the `'dot'` method or using a functional call:

```
1 newvar z(10) uncertain; % declare the uncertainty
2 rome_constraint(z.mean >= -1); % 'dot' method
3 rome_constraint(mean(z) <= 1); % functional call
```

Code Segment 16: Specifying the Mean Support

In addition to the simple box-type constraints above, we can also use polyhedral constraints can also to specify the mean support, for example:

```
rome_constraint(A * z.mean <= b);
```

To specify a fixed numeric mean, we could either use a constraint or a direct assignment (using the `set_mean` function). We can also assign to different components of the uncertainty vector:

```
1 newvar z(4) uncertain; % declare uncertainty
2 rome_constraint(mean(z(1:2)) == 1); % rome_constraint
3 z(3:4).set_mean(2); % using set_mean
```

### 4.1.3 Covariance Matrix

The covariance matrix is specified via assignment to the `covar` member of the `rome_var` object. It can be specified in one of three forms:

1. A symmetric positive semidefinite matrix with each side having a length equal to the total size of the uncertain variable
2. A non-negative vector with length equal to the total size of the uncertain variable. The actual covariance matrix is assumed to be diagonal
3. A non-negative scalar. The actual covariance matrix is assumed to be diagonal and each uncertain variable has the same variance specified by the input scalar.

The following three ROME statements are equivalent, to specify an uncertainty vector with uncorrelated components, with variance (not standard deviation) of each component having a value of 4.

```
1 z.Covar = 4*eye(N);
2 z.Covar = 4*ones(N, 1);
3 z.Covar = 4;
```

Code Segment 18: Specifying the Covariance Matrix

ROME also allows the covariance matrix to be specified for a linearly-transformed version of the model uncertainties. This technique is useful if the modeler only has access to the linear-transformed uncertainties, perhaps due to the measurement process, or if the specified uncertainties represent pseudo uncertainties such as segregated uncertainties.

```
1 F = rand(2, 4);
2 z = newvar(4, 'uncertain');
3 zeta = F*z;
4 zeta.Covar = eye(2);
```

Code Segment 19: Covariance for Linearly-Transformed Uncertainties

### 4.1.4 Directional Deviations

We can also specify forward and backward deviation (Chen, Sim, and Sun [6]) information for stochastically independent components of the model uncertainties. These are specified using assignments to the `FDev` and `BDev` member variables of an uncertainty `rome_var` object. Again, the standard binary scalar expansion applies, and we can specify the directional deviations using either

1. A non-negative vector having the same size as the uncertain variable
2. A non-negative scalar

The following code segment illustrates how the directional deviations can be specified for `z`, which is assumed to have independent components

```

1 newvar z(4) uncertain;
2 z.FDev = ones(4, 1); % Assignment by a non-negative vector
3 z.BDev = 1;          % Assignment by binary expansion

```

Code Segment 20: Specifying the Directional Deviations

Similar to the technique of specifying the covariance matrix, directional deviations can be specified for linearly-transformed uncertainties. A standard illustration of when this will occur is when the declared uncertainties represent segregated uncertainties whose original uncertainties are independent. To illustrate, suppose we model a 4-dimensional segregated uncertainty vector  $\mathbf{z}$ , obtained from segregating an original 2-dimensional uncertainty vector into positive and negative half-spaces. Mathematically, suppose we have

$$\tilde{\mathbf{z}} = \begin{bmatrix} \tilde{u}^+ \\ \tilde{w}^+ \\ \tilde{u}^- \\ \tilde{w}^- \end{bmatrix}$$

with  $\tilde{\mathbf{z}}$  independent from  $\tilde{\mathbf{w}}$ . We then can specify directional deviations in the following manner

```

1 F = [eye(2), eye(2)];
2 z = newvar(4, 'uncertain', 'nonneg');
3 new_z = F*z;          % linearly transform uncertainties
4 new_z.FDev = 3;      % specify FDev by binary expansion
5 new_z.BDev = [3;4];  % specify BDev by vector assignment

```

Code Segment 21: Directional Deviations for Linearly-Transformed Uncertainties

We issue the following caveat: ROME does not perform internal consistency checks that the components are independent (at least in this current version). The onus is on the user to verify independence.

## 4.2 Example: Robust Counterpart

In Code Segment 22, we provide a simple example of how uncertain variables can be used to implement the classical robust counterpart (RC) for an uncertain linear program.

## 4.3 Linear Decision Rules

Linear-Decision Rules (LDR), also termed Affinely Adjustable Robust Counterparts (AARC) in the literature, are bi-affine functions in a certain variable  $x \in \mathbb{R}^n$  and an uncertain variable  $z \in \mathbb{R}^N$ , which separately affine in both variables. LDR variables are used to describe decision variables which are dependent upon model uncertainties.

New LDR variables can be constructed from an uncertain variable using the `newvar` command (see Section 3.3). Products between a (pure) certain variable and a (pure) uncertain variable will result in an LDR variable. Affine operations on allocated LDR variables will result in new LDR variables, which can be assigned to named variables in MATLAB, similar to deterministic variables (see Section 3.4). Standard arithmetic operations (addition/subtraction,

```

1 % EX_SIMPLE_LP_RC.m
2 % Script file for solving a simple Linear Program,
3 % with a robust counterpart
4
5 % Set up ROME Environment and instantiate model
6 h = rome_begin('Simple LP - RC');
7 newvar x y; % Set up modeling variables
8
9 % define a scalar uncertainty and its support
10 newvar z uncertain;
11 rome_box(z, 1.5, 2.5);
12
13 % set up objective function
14 rome_maximize(12 * x + 15 * y);
15
16 % input constraints
17 rome_constraint(x + z*y <= 40); % notice product of z*y
18 rome_constraint(4*x + 3*y <= 120);
19 rome_constraint(x >= 0);
20 rome_constraint(y >= 0);
21
22 % solve and extract solution values
23 h.solve;
24 x_val = h.eval(x);
25 y_val = h.eval(y);
26
27 rome_end; % Clear up ROME environment

```

Code Segment 22: Illustration of a Simple Robust Counterpart (RC)

matrix/array multiplying, subscripted referencing/assignment/deletion, array reshaping/replication) are permitted on LDR variables. Similar to certain and uncertain variables, you can apply equality and inequality constraints to LDR Variables by using the `rome_constraint` command. However, in this version of ROME, non-linear functions (e.g. powers, norm2) are not permitted on LDR variables.

A key property unique to the construction of LDR variables in ROME is the ability to define how the LDR depends on the model uncertainties. Specifically, in ROME, we can construct LDRs which only depend on a subset of the model uncertainties. A classic case when this is necessary is in the modeling of sequential decisions, when uncertainties are progressively revealed. The resulting LDR, is termed as *non-anticipative*. We refer interested readers to our working paper [10] for more comprehensive modeling examples involving non-anticipativity. Here, we will illustrate how such non-anticipative LDRs can be constructed in ROME

### Example

We consider the classic example, where an  $N$ -dimensional LDR  $y$ , is dependent on an  $N$ -dimensional uncertainty vector  $z$ . However, to model non-anticipativity, we require that the  $i^{th}$  component of  $y$  should only depend on the first to the  $i^{th}$  component of the uncertainty. We can express this using the loop:

```

1 newvar z(N) uncertain; % Declare Uncertainties
2 y = []; % Make an empty output

```

```

3 for ii = 1:N % Iterate
4     newvar y_tmp(z(1:ii)) linearrrule; % Create a 'temp' LDR
5     y = cat(2,y,y_tmp); % Append to output
6 end

```

Code Segment 23: Creating Non-anticipative Dependencies

We notice that in each iteration of the loop, we declare a separate LDR variable, named `y_tmp`, which depends on the ‘previous’ components of the uncertainties, and we append it to the output LDR `y`. However, this code is rather inefficient, and instead, we can use the ‘Pattern’ option in the construction of the LDR to make this code more efficient, as follows:

```

1 newvar z(N) uncertain; % Declare Uncertainty
2 patternY = [true(N, 1), tril(true(N))]; % Make dependency
3 newvar y(N, z, 'Pattern', patternY); % Create LDR

```

Code Segment 24: Creating Non-anticipative Dependencies (Vectorized)

The specified pattern should be a logical  $M$  by  $(N+1)$  matrix ( $M$ : output dimension of the LDR,  $N$ : dimension of the uncertainties), with 1’s on the  $(i, j)$  entry if the  $i^{th}$  output component of the LDR is allowed to depend on the  $(j - 1)^{th}$  component of the uncertainty.  $j = 1$  represents the deterministic dependency. In the example above, we notice that the pattern matrix, which we called `patternY`, is an augmented lower triangular matrix, as such:

$$\text{patternY} = \begin{bmatrix} 1 & 1 & 0 & 0 & \dots & 0 \\ 1 & 1 & 1 & 0 & \dots & 0 \\ 1 & 1 & 1 & 1 & \dots & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & 1 & 1 & 1 & \dots & 1 \end{bmatrix}$$

## 5 Deflected Linear Decision Rules

Deflected Linear Decision Rules (DLDR) introduced by Chen, Sim, Sun, and Zhang [7], and the recently more general Bi-Deflected Linear Decision Rules (BDLDR) introduced by Goh and Sim [9] have been shown to improve upon basic LDRs for use in modeling and solving robust optimization problems. However, the improvement comes at the cost of additional modeling and (usually) computational complexity, as using Deflected LDRs typically require solving a series of secondary optimization problems.

In order to increase the ease of using Deflected LDRs, ROME provides in-built support for the non-anticipative BDLDR of Goh and Sim [9]. To use Deflected LDRs in modeling a problem, we will first need to ensure that the optimization problem fits into the distributionally robust optimization framework presented in our earlier work [9]. The modeling code is *identical* to modeling the problem using LDRs, except that you would call `solve_deflected` instead of calling `solve`. Internally, the call `solve_deflected` will analyze the constraint structure of the model, internally solve the the secondary problems and correspondingly modifies the problem structure to yield the improvement over LDRs.

## 5.1 Advanced Deflection Options

The `rome_model` object contains the following members, `defObjFn`, `defLDRRestrict`, and `defBounds`, whose values can be set to override the default deflection settings. As noted in our earlier work [9], we recommend using the same objective as the nominal problem for the subproblems in the BDLDR, which is also the default in ROME. For more complex problems with linear expectation constraints, it might be desirable to use alternate objectives for the subproblems. This objective can be supplied by assigning the corresponding objective to the `defObjFn` member of the current `rome_model` object.

Furthermore, for deeper analysis of the optimization model or perhaps for code efficiency reasons, you might want to deflect some, but not all, of the LDR variables in the model. This is achieved by assigning `defLDRRestrict` with a vectorized collection of LDRs which you want deflected.

Finally, you might want to control which robust bounds are used to bound the mean positive part of the deflected LDRs. This can be done by assigning the `defBounds` member with a *cell array* of function handles to bounding functions. A list of allowable robust bounds can be found in the subsection 5.2 which follows. By default, the deflection algorithm selects bounds for all specified distributional properties, e.g. if the covariance matrix is not specified, the `rome_covar_bound` is omitted. The following simple example shows how the BDLDR can be used:

### Example

This example was mentioned in our earlier work [9], when we discussed the motivation for designing the BDLDR. For a scalar uncertainty  $\tilde{z}$  with infinite support, zero mean and unit variance, we aim to approximately solve the following problem:

$$\begin{aligned} \min_{y(\cdot), u(\cdot), v(\cdot)} \quad & \sup_{\mathbb{P} \in \mathbb{F}} \mathbb{E}_{\mathbb{P}} (u(\tilde{z}) + v(\tilde{z})) \\ \text{s.t.} \quad & u(\tilde{z}) - v(\tilde{z}) = y(\tilde{z}) - \tilde{z} \\ & 0 \leq y(\tilde{z}) \leq 1 \\ & u(\tilde{z}), v(\tilde{z}) \geq 0 \end{aligned}$$

where the decision rules  $y, u, v$  are general functions of the underlying uncertainty. Generally, this is intractable, so and we apply the standard approximation, and restrict ourselves to considering  $y, u, v$  that are LDRs. Most unfortunately, the LDR approximation in this case is terrible, and the problem above is infeasible. We therefore try a complex decision rule, the BDLDR. To do so, we use the following code in ROME:

## 5.2 List of Robust Bounds

In the current version of ROME, we have provided functions which represent several upper bounds on  $\sup_{\mathbb{P} \in \mathbb{F}} \mathbb{E}_{\mathbb{P}} ((\cdot)^+)$ , the expected positive part of an LDR variable, when the family of distributions  $\mathbb{F}$  are characterized by different distributional properties. These bounds can be supplied as function handles to the deflected LDR routine, `apply_na_bdldr`, as described in the previous section. The theory for deriving these bounds can be found in our earlier work [9]. The

```

1 % EX_BDLDR.m Script to test Bi-deflected LDR
2
3 % Set up ROME Environment and instantiate model
4 h = rome_begin('Simple BDLDR');
5
6 % define a scalar uncertainty and its properties
7 newvar z uncertain;
8 z.set_mean(0);
9 z.Covar = 1;
10
11 % define the decision rules
12 newvar y(z) linearrule;
13 newvar u(z) v(z) linearrule nonneg;
14
15 % set up objective function
16 rome_minimize(mean(u + v));
17
18 % input constraints
19 rome_constraint(u - v == y - z);
20 rome_box(y, 0, 1);
21
22 % solve and extract objective values
23 h.solve_deflected;
24 obj_val = h.objective;
25
26 y_sol = h.eval(y);
27 rome_end; % Clear up ROME environment

```

Code Segment 25: Example of Bi-Deflected LDR

bounding functions implemented and distributed together with this version of ROME are

1. `rome_supp_bound`: Constructs a bound based on support and mean support information.
2. `rome_covar_bound`: Constructs a bound based on covariance and mean support information.
3. `rome_dirdev_bound`: Constructs a bound based on directional deviation and mean support information.

### 5.3 Using Robust Bounds Directly

The robust bounds listed above can also be directly used in modeling problems which contain  $\sup_{\mathbb{P} \in \mathbb{F}} E_{\mathbb{P}} \left( (y_0 + \mathbf{y}'\tilde{\mathbf{z}})^+ \right)$  or  $\sup_{\mathbb{P} \in \mathbb{F}} E_{\mathbb{P}} \left( (y_0 + \mathbf{y}'\tilde{\mathbf{z}})^- \right)$  terms in their objective or constraints (both assumed to be convex). For example, suppose we want to compute an upper bound for  $\sup_{\mathbb{P} \in \mathbb{F}} E_{\mathbb{P}} \left( (1 + \tilde{z})^+ \right)$ , for a scalar uncertainty  $\tilde{z}$  with known distributional properties, and an LDR variable  $\mathbf{x}$ , we can use either of the following objectives in ROME:

```

1 % if mean and support are known
2 rome_minimize(rome_supp_bound(x));
3 % if mean and covariance are known,
4 rome_minimize(rome_covar_bound(x));
5 % if mean and directional deviations are known,

```

```
6 | rome_minimize(rome_dirdev_bound(x));
```

together with the constraint: `rome_constraint(x == 1 + z)`.

## 5.4 Combining Robust Bounds

In most models, we will have a combination of distributional properties, for example, mean, support, as well as covariance. We can construct a better bound on  $\sup_{\mathbb{P} \in \mathbb{F}} (())^+$ , by convolving the individual bounds (see e.g. Chen and Sim [5]). The convolution can be performed in ROME using the in-built function, `rome_create_bound`.

The first argument to `rome_create_bound` is the LDR variable which the bound should be applied for. The next set of arguments is a list of function handles to the individual bounds to be convolved. Code Segment 26 shows how to use all the available distributional information on a scalar variable  $\tilde{z}$  to obtain a bound on  $\sup_{\mathbb{P} \in \mathbb{F}} ((1 + \tilde{z})^+)$ :

```
1 | % EX_COMBINE_BOUNDS.m
2 | % Script file to illustrate combining bounds
3 |
4 | % Set up ROME Environment and instantiate model
5 | h = rome_begin('Combine Bounds');
6 |
7 | % define a scalar uncertainty
8 | newvar z uncertain;
9 |
10 | z.set_mean(2);           % mean
11 | rome_box(z, 1.5, 2.5); % support
12 | z.Covar = 1;           % covariance
13 | z.FDev = 0.9;         % forward deviation
14 | z.BDev = 0.8;         % backward deviation
15 |
16 | newvar x(z) linearrule; % define a scalar LDR
17 | rome_constraint(x == 1 + z); % input constraint
18 |
19 | % assign bound to an intermediate value
20 | bnd_x = rome_create_bound(x, ...
21 |     @rome_supp_bound, ...
22 |     @rome_covar_bound, ...
23 |     @rome_dirdev_bound);
24 | rome_minimize(bnd_x);
25 |
26 | h.solve;
27 | rome_end; % Clear up ROME environment
```

Code Segment 26: Combining Bounds

## 6 Analysis of Optimization Results

### 6.1 Getting the Objective Function Value

The solutions to the optimization problem in ROME can be extracted anywhere after the call to `solve` and prior to the call to `rome_end`. The objective after

optimization can also be extracted by calling the `objective` member function of the current `rome_model` object. This works for when the objective contains uncertainties as well. In that case, the `objective` function returns the worst-case objective. For example, consider the following problem:

$$Z = \min_{\mathbf{x}} \{z' \mathbf{x}, \forall z \in \mathcal{W}\}$$

$$\text{s.t. } \mathbf{x} \in \mathcal{X}$$

To find the value of  $Z$ , we can use this Code Segment:

```

1  h = rome_begin('Objective Test');
2  newvar z(5) uncertain; % set uncertainty
3  newvar x(5);          % set decision variable
4  rome_minimize(z' * x); % set objective
5  ...
6  ... % Set constraints of x and support of z
7  ...
8  h.solve;              % call to solve
9  Z = h.objective;     % gets worst-case objective
10 rome_end;
```

Code Segment 27: Extracting the worst-case objective

## 6.2 Using `eval` to Get Solutions

The solutions are obtained from the model object using a call to the `eval` member function of the current `rome_model` object. The `eval` function takes as its argument a `rome_var` object which is to be evaluated, and returns the solution which is embedded in an object from the `rome_sol` class. If deflections are used and if the target `rome_var` object contains deflected components, it will also be stored in the `rome_sol` object. We note at this point that `eval` can be supplied with an argument that is not a primitive declared variable, i.e. the result of any expression of type `rome_var` can be supplied as an argument to `eval`. For example, the following statements are valid:

```

1  h = rome_begin('Eval Test');
2  newvar z(5) uncertain;
3  newvar x(5, 1);
4  newvar y(4, z) linearrule;
5  ...
6  ... % h.solve is called somewhere here
7  ...
8  x_sol = h.eval(x);
9  z_sol = h.eval(z);
10 y_sol = h.eval(y);
11 xz_sol = h.eval(x .* z);
12 expr_sol = h.eval(y + 3 + x(1:4));
```

Code Segment 28: Using `eval` to extract solution

The `rome_sol` object basically encodes the solution in two components, the linear component, and the deflected component. A MATLAB statement which assigns to a `rome_sol` object, and which is not terminated by a semi-colon, will display a prettyprint of the full solution, with both (deflected and linear components). This is particularly useful for debugging and prototype code. For larger problems with many deflected components, the prettyprint will be too long and

complex for any useful analysis. The linear and deflected components, can, however, be numerically extracted from the solution object, using the `linearpart` and `deflectedpart` functions. The list below lists the member functions of `rome_sol` object and their return values:

### 6.3 List of `rome_sol` Members

- `size`: returns the output dimensions of the object.
- `numdepvars`: returns the number of dependent uncertain variables.
- `linearpart`: returns the LDR component of the object as a matrix of dimensions `[prod(size) x numdepvars]`;
- `deflectedpart`: returns the deflected components of the object, in two output argument. The first argument is a matrix of the coefficients of the deflected components, while the second argument stores a matrix of the values of the deflected components.
- `isdeflected`: returns true if the solution object contains deflected components and false otherwise.
- `insert`: instantiates current solution object with realization of uncertainties.

### Example

Admittedly, the interpretation of outputs from the `rome_sol` object can be quite challenging. Here, we present a very simple example, to illustrate how to interpret the various parts of the `rome_sol` object, and hopefully motivate our choice of such an output format.

To set up the illustration, suppose we have completed the solution of an optimization problem, and we have on hand a solution object `x_sol`. We give an example of how the linear and deflected parts can be extracted, and how we can interpret these results mathematically. Consider the following code segment and subsequent outputs: (The actual `x_sol` used in this code segment can be obtained from loading the `output_sol_interpret_data.mat` file in MATLAB): We first notice that `lin_x` has 3 columns, which indicates that the uncertainty vector for the model has 2 components,  $\tilde{z}_1$  and  $\tilde{z}_2$

$$\mathbf{x} = \begin{bmatrix} 10 \\ 12 + 4\tilde{z}_1 + (5 + 6\tilde{z}_1)^- - (4 + 3\tilde{z}_1)^- \\ 11 - 1\tilde{z}_1 + 30\tilde{z}_2 + 2(5 + 6\tilde{z}_1)^- + (1 + 1\tilde{z}_1 + 2\tilde{z}_2)^- - (6 - 8\tilde{z}_2)^- \end{bmatrix}$$

or, perhaps, in better correspondence with the structure of the output, in this

```

1 %
2 % Assume h.solve has been called before
3 %
4 >> lin_x = linearpart(x)
5
6 lin_x =
7
8      10      0      0
9      12      4      0
10     11     -1      3
11
12 >> [def_coeff, def_vals] = deflectedpart(x)
13
14 def_coeff =
15
16      0      0      0      0
17      1     -1      0      0
18      2      0      1     -1
19
20 def_vals =
21
22      5      6      0
23      4      3      0
24      1      1      2
25      6      0     -8

```

Code Segment 29: Sample Output After Using Deflected LDR

form:

$$\begin{aligned}
\mathbf{x} = & \begin{bmatrix} 10 \\ 12 + 4\tilde{z}_1 \\ 11 - 1\tilde{z}_1 + 3\tilde{z}_2 \end{bmatrix} \\
& + \begin{bmatrix} 0 \\ 1 \\ 2 \\ 0 \\ 0 \\ 1 \end{bmatrix} (5 + 6\tilde{z}_1)^- + \begin{bmatrix} 0 \\ -1 \\ 0 \\ 0 \\ 0 \\ -1 \end{bmatrix} (4 + 3\tilde{z}_1)^- \\
& + \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} (1 + 1\tilde{z}_1 + 2\tilde{z}_2)^- + \begin{bmatrix} 0 \\ -1 \\ 0 \\ 0 \\ 0 \\ -1 \end{bmatrix} (5 - 8\tilde{z}_2)^-
\end{aligned}$$

which is, taking the  $(\cdot)^-$  operator componentwise, equivalent to:

$$\mathbf{x} = \begin{bmatrix} 10 & 0 & 0 \\ 12 & 4 & 0 \\ 11 & -1 & 3 \end{bmatrix} \begin{bmatrix} 1 \\ \tilde{z}_1 \\ \tilde{z}_2 \end{bmatrix} + \begin{bmatrix} 0 & 0 & 0 & 0 \\ 1 & -1 & 0 & 0 \\ 2 & 0 & 1 & -1 \end{bmatrix} \left( \begin{bmatrix} 5 & 6 & 0 \\ 4 & 3 & 0 \\ 1 & 1 & 2 \\ 5 & 0 & -8 \end{bmatrix} \begin{bmatrix} 1 \\ \tilde{z}_1 \\ \tilde{z}_2 \end{bmatrix} \right)^-$$

Notice the natural correspondence of each matrix in the expression above with `lin_x`, `def_coeff`, and `def_vals` respectively. In general, every `rome_sol` object has the following structure:

$$\hat{\mathbf{x}}(\tilde{\mathbf{z}}) = (\mathbf{x}^0 + \mathbf{X}\tilde{\mathbf{z}}) + \mathbf{P}(\mathbf{y}^0 + \mathbf{Y}\tilde{\mathbf{z}})^-.$$

In the example above, `lin_x` corresponds to  $[\mathbf{x}^0, \mathbf{X}]$ , `def_coeff` corresponds to the matrix  $\mathbf{P}$ , and `def_vals` corresponds to  $[\mathbf{y}^0, \mathbf{Y}]$ .

## 6.4 Using insert for Monte Carlo Simulation

We also can use `rome_sol` objects for Monte-Carlo simulation analysis of the optimization results. It is typically desirable to compare the result of the robust optimization solution with benchmarks from alternate algorithms or techniques. We can do this easily in ROME. Since each `rome_sol` object represents a function of the model uncertainties, we can instantiate the `rome_sol` object (using the provided `insert` function) with realizations of uncertainties to obtain concrete numeric decisions, which can be compared *vis-a-vis* decisions obtained from other methods. Code Segment 30 illustrates how to use a loop to instantiate a solution object with a series of standard normal variables. Vectorized instantiation is also permitted by using a flag in the instantiation, as in Code Segment 31.

```
1 x_sol = h.eval(x); % Get the solution object
2 z_vals = randn(N, 100); % Draw standard normal r.v.s
3 x_vals = zeros(M, 100); % Allocate output matrix
4 for ii = 1:100
5     % instantiate solution with r.v.
6     x_vals(:, ii) = x_sol.insert(z_vals(:, ii));
7 end
```

Code Segment 30: Instantiating a solution with uncertainties

```
1 x_sol = h.eval(x); % Get the solution object
2 z_vals = randn(N, 100); % Draw standard normal r.v.s
3 vectorize_flag = true; % Set the vectorize flag to true
4
5 % instantiate solution
6 x_vals = x_sol.insert(z_vals, vectorize_flag);
```

Code Segment 31: Instantiating a solution with uncertainties (Vectorized)

Instantiating solutions with uncertainties is not only useful for Monte-Carlo simulation analysis and benchmarking, but we can use this even as a prescriptive tool for non-anticipative modeling. Suppose we have some solution `y_sol` after an optimization in ROME, which we have explicitly constrained to only depend on the first component of the  $N$ -dimensional uncertainty vector. Now suppose we are at the point in our timeline when we are ready to make the decision corresponding to `y_sol`, i.e. the first component of uncertainty has been revealed, with, say, a numerical value of 10. We can instantiate `y_sol` with an uncertainty vector with 10 as the first component and zeros everywhere else to find the numerical value of the decision corresponding to `y_sol`, since by the explicit non-anticipative constraints, `y_sol` is invariant to all other components of the uncertainty. Code Segment 32 illustrates this in ROME:

```
1 z_val = [10; zeros(N-1, 1)]; % Make instantiation vector
2 y_val = y_sol.insert(z_val) % Instantiate
```

Code Segment 32: Using `insert` to prescribe non-anticipative decisions

## 7 Solver Options

### 7.1 Polling the Solvers

A list of supported solvers in your version of ROME can be accessed using a call to `rome_solver('all')`. The solvers are listed into two categories, “Main” solvers, which have been more extensively tested with ROME, and “Experimental” solvers, which are less well-tested with ROME and more likely to be buggy. Please note that this function returns the solvers that are supported by the your version of ROME, but does *not* check if the solvers are actually installed in your system, or installed correctly.

A call to `rome_solver` without any input arguments returns the currently selected solver as a string. The list of “Main” supported solvers in ROME as at this writing are:

- `'CPLEX'`: Uses the ILOG CPLEX Solver.
- `'MOSEK'`: Uses the MOSEK Solver.
- `'SDPT3DUAL'`: Uses SDPT3 to solve the dual problem.

### 7.2 Choosing a Solver

There are three different ways to choose a solver in ROME, which we describe in this section, in increasing order of permanence.

#### 7.2.1 At solve Time

You can choose a particular solver during the call to `rome_model/solve` or `rome_model/solve_deflected`, by supplying your solver choice as an input string. This overrides any selected solver for the current model *only*. Consider the following Code Segment, which assumes that MOSEK was chosen as the original solver.

```
1  str = rome_solver;           % Output >> str = MOSEK
2  h = rome_begin('Foo');      % Begin model
3  ...                          % Some modeling code
4  h.solve('CPLEX');          % Uses CPLEX to solve this problem only
5  rome_end;                   % End Model
6  str = rome_solver;         % Output >> str = MOSEK
```

Code Segment 33: Choosing a solver at “solve” step

#### 7.2.2 Change Default Solver

If you are solving multiple models, and trying to test which solver works better on your model, it’s quite troublesome to do a search-and-replace on all the arguments to `solve`. Instead, you might want to control the default solver used on all models. This can be done by a call to the `rome_solver` function with a single input argument, using your chosen solver name as the input string. The return argument to this function is the previously chosen solver. Consider the following Code Segment, which assumes that MOSEK was originally chosen as the solver.

```

1  str = rome_solver;           % Output >> str = CPLEX
2  str = rome_solver('MOSEK'); % Output >> str = CPLEX
3  str = rome_solver;           % Output >> str = MOSEK
4  h = rome_begin('Bar');       % Begin model
5  ...                           % Some modeling code
6  h.solve;                     % Uses MOSEK to solve
7  rome_end;                     % End Model
8  str = rome_solver;           % Output >> str = MOSEK

```

Code Segment 34: Changing the default solver

We notice that even after the call to `rome_end`, the solver choice remains as MOSEK. The only way to destroy the persistence of this form of solver choice is to call `clear global`.

### 7.2.3 Change Default Solver Forever

Suppose you only have SDPT3 installed in your system, and you don't have either CPLEX or MOSEK installed. It might be rather cumbersome to have to choose a solver each time you start a MATLAB session to use ROME (even though you'll only have to issue a call to `rome_solver` once per MATLAB session, assuming you don't call `clear global`). To choose a default solver which persists over MATLAB sessions, you'll have to edit the `rome_begin.m` file. Find the line

```
ROME_ENV.DEF_SOLVER='CPLEX';
```

and change it to

```
ROME_ENV.DEF_SOLVER='SDPT3DUAL';
```

## 7.3 Controlling Output Verbosity

The verbosity of the displayed output can be controlled using the function `rome_verbose`. The verbosity level is a non-negative integer, representing increasing output verbosity. In the current versions the allowed verbosity level ranges from 0 to 2. The recommended (and default) setting is with a verbosity level of 1.

If `rome_verbose` is called without arguments, this function will just return the current verbosity level. If `rome_verbose` is called with a single scalar integer argument, the verbosity level will be set to the input argument, and the *previous* verbosity level will be returned.

## 8 Acknowledgements

ROME was not built in a day, and it represents a lot of hard work and support from various people. We gratefully acknowledge the following people who have contributed in various ways to the development of ROME and subsequent improvement of the codebase.

1. Jingxi Wang for helping with the development of several ROME components, including the interfaces to the MOSEK and SDPT3 solvers of the code.

2. Boris Bachelis for reporting bugs in the code and several errors and omissions from this User's Guide.
3. Marcus Ang for providing a test case of a large linear problem.
4. Our families for their unwavering support during the course of this project.

## Appendix A List of Operators and Functions

### A.1 Operators

MATLAB's standard arithmetic operators have been overwritten to work with `rome_var` objects. Accordingly, ROME objects must obey standard rules pertaining to the permissibility of operations, e.g. two objects can only be added if they have the same size, two objects can be (matrix) multiplied if their inner dimensions correspond. The list of overridden operations are:

- Binary addition ( $A + B$ ) and subtraction ( $A - B$ ), unary plus ( $+A$ ) and unary minus ( $-A$ );
- Array multiplication ( $A .* B$ ) and division ( $A ./ B$ ) (division only for a numerical matrix  $B$ ).
- Matrix multiplication ( $A * B$ ) and division ( $A / B$ ) (division only for numerical scalar  $B$ ).
- N-dimensional concatenation, i.e `cat`, `vertcat`, `horzcat` operators.
- Real transpose  $A.'$  and hermitian transpose  $A'$  operators.
- Subscripted reference (`subsref`),  $B = A(2:3, [3, 4\ 5])$
- Subscripted assignment (`subsasgn`),  $A(1:3) = B$ , and subscripted deletion,  $A(1:3) = []$ .
- `size`, `length`, `reshape`, `squeeze` and `end` functions.

Notice that the `numel` function has not been overridden, and calling `numel` on a `rome_var` object will always yield 1. To find the total number of elements stored in a variable  $x$ , you can use the property `x.TotalSize`.

### A.2 Standard Functions

Certain MATLAB functions for numeric types have also been overwritten, and some new functions have also been added. For the most part, the functions which have been overridden preserve the same conventions as their MATLAB counterparts. e.g. `sum(A, 2)` sums the second dimension of  $A$ . The following functions have been implemented:

#### Affine Functions

- `sum(x, dim)`: Summation over the dimension specified by `dim`.
- `diff(x, n, dim)`:  $n$ -th order differencing over the dimension `dim`. (Actually only 1st order difference been implemented.)

## Convex Functions

- `norm1(x)`, `norm2(x)`:<sup>1</sup>. These functions have been vectorized to take the norm over the first dimension.
- `power(x, n)`, `powercone(x, mu, n)`: Power function,  $x^n$  and power cone function,  $\mu \left(\frac{x}{\mu}\right)^n$ . Only works for positive integers  $n$  at the moment.  $x$  and  $\mu$  must be explicitly constrained to be nonnegative.
- `exp(x)`, `expcone(x, mu)`, `quadexpcone(x, y, mu)`: Exponential function  $\exp(x)$ , exponential cone function  $\mu \exp\left(\frac{x}{\mu}\right)$ , and quadratic exponential cone function,  $\mu \exp\left(\frac{x}{\mu} + \left(\frac{y}{\mu}\right)^2\right)$ .  $\mu$  must be explicitly constrained to be non-negative.

## Concave Functions

- `hypoquad(x, y)`: Hypoquad function. Returns  $w$  such that  $w^2 \leq xy$ .

These functions are internally represented (exactly where possible, and approximately otherwise) as second-order conic (SOC) constraints. Again, ROME frees the modeler from the tedium of performing the conversion into SOCs, and transparently performs the conversions.

---

<sup>1</sup>to be merged together into `norms` in later version

## Appendix B Expert Variable Creation Drivers

While the `newvar` command can be used to create deterministic, uncertain, and linearrule variables, it internally translates the call into one of the following internal variable creation calls:

1. `rome_model_var`: Creates a deterministic variable.
2. `rome_rand_model_var`: Creates an uncertain variable.
3. `rome_linearrule`: Creates an linear decision rule.

For efficiency, you might choose to call these drivers directly in your code instead of `newvar`.

### B.1 Syntax for `rome(_rand)_model_var`

The syntax for the calls to `rome_model_var` and `rome_rand_model_var` are essentially identical, taking the form of

```
x = rome_model_var(size, options);
```

where `size` can either be a multi-dimensional array of positive integers or a comma-separated list. the `options` field requires some discussion. Options are handled entered in key-value pairs, where the keys have to be pre-defined strings. The values are typically constants, most of which are listed as static constants in the `rome_constants` class. The list of allowed key-value pairs and their meanings are listed here. In the present version all options are allowed

Key	Value	Description
'Cone'	NNOC	Variable is confined to the non-negative orthant cone, i.e. component-wise $\geq 0$
	SOC	Variable is confined to the second-order (Lorentz) cone. $\begin{bmatrix} t \\ \mathbf{x} \end{bmatrix} \in SOC \Rightarrow \ \mathbf{x}\ _2 \leq t$
'Continuity'	INTEGER	Integer Variable
	BINARY	Binary Variable

Table 1: Allowed key-value pairs for deterministic and uncertain variables. Values are all members of `roam_constants`, i.e. `roam_constants.NNOC` etc.

for deterministic variables, while only 'Cone' properties can be set for uncertain variables. Note that integer (including binary) variables are only permitted if the chosen underlying solver can handle such variables.

### B.2 Syntax for `rome_linearrule`

The syntax for the call to `rome_linearrule` differs slightly from the simpler calls above. Firstly, we require an uncertain variable to have already been declared, similar to the standard call using `newvar`. We assume this variable is named `z`. The syntax for using `rome_linearrule` is then

```
y = rome_linearrule(size, z, options);
```

Again, the `size` can be either an multi-dimensional array of positive integers or a comma-separated list. `z` is the uncertain variable which we assume to be previously defined, and again, the `options` field consists of key-value pairs. The only allowable option from Table 1 is the NNOC restriction. However, we have an additional option unique to LDR variables, the `'Pattern'` key. This can be used as a more efficient means to define the dependency structure of the LDR variable on the underlying uncertainties (see Section 4.3 for details). In this case, the value is no longer a constant, but a user-defined  $M$ -by- $N$  logical matrix, where  $N$  is the dimension of `z` and  $M$  is the product of the size of the LDR variable `y`.

Key	Value	Description
<code>'Pattern'</code>	logical matrix <code>pY</code>	Defines dependency of LDR variable on uncertainty. <code>pY(i, j) = 1</code> means the $i$ component of the LDR depends on the $j$ component of the uncertainty

Table 2: Allowed key-value pairs for LDR variables.

## References

- [1] M. Baotic and M. Kvasnica. Cplexint - matlab interface for the cplex solver, June 2006.
- [2] A. Ben-Tal, A. Goryashko, E. Guslitzer, and A. Nemirovski. Adjustable robust solutions of uncertain linear programs. *Mathematical Programming*, 99:351–376, 2004.
- [3] A. Ben-Tal and A. Nemirovski. Robust convex optimization. *Mathematics of Operations Research*, 23(4):769–805, 1998.
- [4] D. Bertsimas and M. Sim. The price of robustness. *Operations Research*, 52(1):35–53, 2004.
- [5] W. Chen and M. Sim. Goal-driven optimization. *Operations Research*, 57(2)(2):342–357, 2009.
- [6] X. Chen, M. Sim, and P. Sun. A robust optimization perspective on stochastic programming. *Operations Research*, 55(6):1058–1071, 2007.
- [7] X. Chen, M. Sim, P. Sun, and J. Zhang. A linear decision-based approximation approach to stochastic programming. *Operations Research*, 56(2):344–357, 2008.
- [8] X. Chen and Y. Zhang. Uncertain linear programs: Extended affinely adjustable robust counterparts. *Operations Research*, 2009.
- [9] J. Goh and M. Sim. Distributionally robust optimization and its tractable approximations. *Working Paper*, 2009.
- [10] J. Goh and M. Sim. Distributionally robust optimization with roam: Robust optimization via algebraic modeling. *Working Paper*, 2009.

- [11] C.-T. See and M. Sim. Robust approximation to multi-period inventory management. *Operations Research*, *forthcoming*, 2009.
- [12] A. L. Soyster. Convex programming with set-inclusive constraints and applications to inexact linear programming. *Operations Research*, 21(5):1154–1157, 1973.
- [13] K. Toh, M. Todd, and R. Tütüncü. Sdpt3 — a matlab software package for semidefinite programming. *Optimization Methods and Software*, 11:545–581, 1999.
- [14] R. Tütüncü, K. Toh, and M. Todd. Solving semidefinite-quadratic-linear programs using sdpt3. *Mathematical Programming Ser. B*, 95:189–217, 2003.