Distributionally Robust Optimization with ROME (part 2)

Joel Goh   Melvyn Sim

Department of Decision Sciences
NUS Business School, Singapore

18 Jun 2009
NUS Business School
Guest Lecture

Outline

Outline

Introduction

Basic Structure of a ROME Program

ROME Features

Example 1: Simple Test

Example 2: Inventory Management

Conclusion

What is ROME?

- ROME: Robust Optimization Made Easy
- Algebraic modeling language in the MATLAB environment for modeling Robust Optimization (RO) problems
- Primarily designed for RO problems within the DRO framework
- ROAM Design Goals
  - Environment for rapid prototyping of new RO ideas
  - Ease the transition from theory to practice
  - Ease numerical studies of RO models

What ROME is NOT

- ROME is NOT a solver engine
  - ROME calls 3rd party solvers to do actual solving (e.g. CPLEX, MOSEK, SDPT3)
  - ROME serves as an intermediary to translate an uncertain optimization program from a mathematically intuitive form into a solver-understandable form
- ROME is NOT a large-scale solver platform
  - ROME can handle medium-sized problems reasonably well, and some large problems
  - Best to write specialized code (e.g. in C, C++)

Why ROME?

- RO programs, especially with more complex decision rules (e.g. BDLDR), can be extremely complex, typically involving the following steps
  - Constructing robust counterparts
  - Finding deflected components (non-anticipative)
  - Constructing robust bounds
- Fortunately, most of these steps are mechanical in nature
  - Potential to be automated

Outline

Ingredients of a Robust Optimization Program

- Uncertainties: Methods to input distributional properties
- Variables: Deterministic variables, recourse variables
- Arithmetic operations on Uncertainties, Variables
- Conic Constraints
- Objective
- Optimization Results

The ROME Environment

- The ROME environment is started by a call to `roam_begin`
    - Sets up internal ROAM structures (invisible)
- A model in ROAM is created by a call to `roam_model`
    - Returns a handle which you can use to access model data, or call model member functions
- Input uncertainties, variables, constraints, and objective
- Call solve to complete the model and run solver.
- Can access optimization results after solve
- `roam_end` clears the ROME environment and frees memory in ROAM structures

Variables and Uncertainties

- Declared with the `newvar` keyword with different options
  - `<nothing>` : deterministic variables
  - `uncertain` : uncertainties
  - `linearrule>` : LDR recourse variables
- Can set distributional properties on uncertainty variables
- Can be construted with various sizes
  - Scalars, vectors, matrices, multi-dimensional arrays

ROME Structure Code (I)

```
% ROME_STRUCTURE.m
% File to illustrate ROME's structure. No real model to be solved.

% PREAMBLE
% Begins ROME Environment and creates model object
rome_begin;                          % Begins the ROME Environment
h = rome_model('Dummy File');        % Creates a model object

% MAIN CODE BODY
% Declare variables, objective and constraints
% Declare uncertain variables first, and set their distributional properties,
% mean, covariance, directional deviations, support. Ends with call to solve.
% Declare variables:
newvar z(5, 1) uncertain;            % Declare z as 5 x 1 uncertainty vector
z.set_mean(0);                       % Set mean of z
z.Covar = 1;                         % Set covariance of z

newvar x(3, 1);                      % Declare 3 x 1 variable x
newvar y(3, 4, z) linearrule;        % Declare 3 x 4 LDR variable y
```

Operators

- Most of MATLAB's arithmetic operators have been overloaded to work with declared ROME variables
  - Addition / Subtraction
  - Array and Matrix Multiplication
  - Subscripted Reference and Assignment (e.g. `A(1, 2)`)
  - Array shape manipulation commands (`size`, `reshape`)
- Idea: whatever you can do with MATLAB matrices, you should be able to do with ROME variables
- Non-standard operation: mean (for uncertainties and LDR variables)

Constraints and Objective

- Constraints
  - Declared using the `rome_constraint` function
  - Accepts ROME expressions containing a single inequality or equalities (examples later)
  - For LDR variables, constraints will be translated into the Robust Counterpart *automatically*
- Objective
  - One objective per model
  - Specified using `rome_minimize` or `rome_maximize`

Getting Optimization Results

- After calling `solve`,
  - Get values of variables using `eval`
  - Can get values of declared variables or even functions of variables

ROME Structure Code (II)

```
% Declare constraints:
rome_constraint(x(2:3) <= 2*x(1:2)); %
rome_constraint(sum(y, 1) == 1);     % Various constraints can be applied.
rome_constraint(y(:, 1) + x >= 2);   % ROME variables can be manipulated by
rome_constraint(norm2(x) <= 1);      % standard arithmetic operations.
rome_constraint(mean(y(2, :)) <= 5); %

% Declare objective:
rome_minimize(sum(x) + sum(y(:, 2))); % objective

% Instruct ROME to solve
h.solve;                             % Terminate the current model with solve

% GET RESULTS
% Get optimization results
y_sol = h.eval(y);                   % Get y
u_sol = h.eval(y(2, 2) + x(1));      % Get function of variables

% CLEANUP
% Clear up ROME memory
rome_end;                            % Complete modeling and deallocate ROME memory
```

Outline

Why Use ROME?

- Uncertainty description
- LDRs as primitives variables in ROAM
- Non-anticipative requirements
- In-built support for BDLDRs
- Numerical analysis of results

Uncertainty Description

- Can set distributional properties of uncertainties
- Stored in ROAM's memory and invoked when necessary
  - e.g. Robust Counterpart for inequalities on LDRs
  - e.g. Robust Bounds

```
newvar z(10, 1)   uncertain;  % create 10 x 1 uncertainty vector

rome_constraint(z.mean == 1); % Sets the mean to be exactly 1

rome_constraint(z >= 0);      % Sets the support of z to be
rome_constraint(z <= 2);      % between 0 and 2 component-wise

z.Covar = eye(10);            % Sets an identity covariance matrix

z.FDev = 1.8*ones(10, 1);     % Sets the forward deviation
z.BDev = 1.8*ones(10, 1);     % Sets the backward deviation
```

LDRs as Primitive Objects

- LDRs are declared and manipulated directly
  - Much more mathematically meaningful
  - Don't have to worry about internal structure of LDR or worry about how to write the robust counterpart
  - Works in concert with the uncertainty description

```
newvar z(5)          uncertain ;  % make uncertainty vector
newvar y1(z)         linearrule;  % create scalar LDR
newvar y2(3, z)      linearrule;  % create 3 x 1 LDR
newvar y3(4, 3, z)   linearrule;  % create 4 x 3 LDR
```

Non-anticipative Requirements

- In ROME, you can make variables with a prescribed dependency pattern on the uncertainties

```
newvar z(N) uncertain;                        % create an uncertainty vector
y = [];                                       % allocate an empty matrix
for ii = 1:N
    newvar tmp(1, z(1:ii)) linearrule;        % create an LDR in each iteration
    y = [y; tmp];                             % append to the output matrix
end
```

- e.g. $\tilde{z}$ where the i^{th} component depends on the first i components of uncertainty
- More directly, can specify dependency pattern at creation

```
pY = [true(N, 1), tril(true(N, N))];          % augmented lower triangular pattern
newvar z(N) uncertain;                        % create an uncertainty vector
newvar y(N, z, 'Pattern', pY) linearrule;     % construct the LDR with the pattern
```

In-built-support for BDLDRs

- As we saw before, BDLDRs improved over LDRs, but were terribly complicated, especially for the non-anticipative case. Recall the steps involved:
  1. Form and solve the two sub-problems
  2. Remove “unnecessary” inequality constraints
  3. Make the BDLDR and construct robust bounds
  4. Solve the final problem
- Design Concept: while the BDLDR is complex, it’s just another decision rule, you shouldn’t have to change your model to use the BDLDR
- Just call `solve_deflected`, instead of `solve`

Numerical Analysis of Results (I)

- For deterministic optimization software packages this is trivial
- Recall that recourse decisions are *functions* of uncertainties
- Most general setting in ROME: deflected variable

$$\hat{\boldsymbol{x}}(\tilde{\boldsymbol{z}}) = \left(\boldsymbol{x}^0 + \boldsymbol{X}\tilde{\boldsymbol{z}}\right) + \boldsymbol{P}\left(\boldsymbol{y}^0 + \boldsymbol{Y}\tilde{\boldsymbol{z}}\right)^-$$

- `eval` function returns an object, which encodes these parameters.
- Use `linearpart` and `deflectedpart` functions

Numerical Analysis of Results (II)

- ROME also includes a "prettyprint" functionality to aid debugging and prototyping of small problems, e.g.

```
y_sol =

-0.000 + 1.000*z1 + 1.00(-0.000 + 1.000*z1)^- - 1.00(1.000 - 1.000*z1)^-
```

- Instantiate solution of uncertainty values using `insert`
- Use as a prescriptive tool or for Monte-Carlo simulation

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

Outline

Recall Motivating Problem for BDLDR

$$
\begin{array}{rl}
\min\limits_{y(\cdot)} & \sup\limits_{\mathbb{P}\in\mathbb{F}} \mathrm{E}_{\mathbb{P}}\left(|y(\tilde{z}) - \tilde{z}|\right) \\
& 0 \leq y(\tilde{z}) \leq 1 \\
& y \in \mathcal{Y}\left(1, 1, \{1\}\right) \\
& \Updownarrow \\
\min\limits_{y(\cdot),u(\cdot),v(\cdot)} & \sup\limits_{\mathbb{P}\in\mathbb{F}} \mathrm{E}_{\mathbb{P}}\left(u(\tilde{z}) + v(\tilde{z})\right) \\
\text{s.t.} & u(\tilde{z}) - v(\tilde{z}) = y(\tilde{z}) - \tilde{z} \\
& 0 \leq y(\tilde{z}) \leq 1 \\
& u(\tilde{z}), v(\tilde{z}) \geq 0 \\
& y, u, v \in \mathcal{Y}\left(1, 1, \{1\}\right)
\end{array}
$$

- Where we have used the identities $x = x^+ - x^-, |x| = x^+ + x^-$.

ROME Code for Simple Example (I)

```
% SIMPLE_EXAMPLE.m
% Script to demonstrate several key features and functions in ROME,
% and illustrate overall structure of a ROME program

% ROME MODELING CODE
% Preamble
rome_begin;                          % Begins the ROME Environment
h = rome_model('Simple Example');    % Creates a model object

% Feature 1: Handling Uncertainties
newvar z uncertain;                  % Declare z as a scalar uncertainty
z.set_mean(0);                       % Set distributional properties of z
z.Covar = 1;                         % zero mean and unit variance.

% Feature 2: Declaring LDRs as primitive objects
newvar u(z) v(z) linearrule nonneg;  % nonnegative LDRs u and v
newvar y(z) linearrule;              % LDR y
```

$$
\begin{array}{rl}
\min\limits_{y(\cdot),u(\cdot),v(\cdot)} & \sup\limits_{\mathbb{P}\in\mathbb{F}} \mathrm{E}_{\mathbb{P}}\left(u(\tilde{z}) + v(\tilde{z})\right) \\
\text{s.t.} & u(\tilde{z}) - v(\tilde{z}) = y(\tilde{z}) - \tilde{z} \\
& 0 \leq y(\tilde{z}) \leq 1 \\
& u(\tilde{z}), v(\tilde{z}) \geq 0 \\
& y, u, v \in \mathcal{Y}\left(1, 1, \{1\}\right)
\end{array}
$$

ROME Code for Simple Example (II)

```
% Objective
rome_minimize(mean(u + v));

% Constraints
rome_constraint(u - v == y - z);        % equality constraint
rome_box(y, 0, 1);                      % 0 <= y <= 1 constraint

% Feature 4: Support for BDLDRs
h.solve_deflected;                 % Uses BDLDR as decision rules
% h.solve;                         % Use this instead to use basic LDRs as decision rules

% Feature 5: Extract solution for analysis
y_sol = h.eval(y);                 % Call 'eval' after solve and before rome_end

% Complete modeling and deallocate ROME memory
rome_end;
```

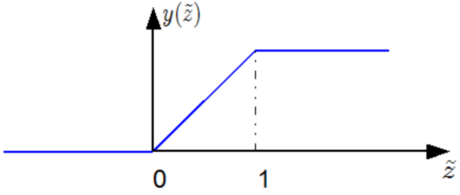
$$
\begin{array}{rl}
\min\limits_{y(\cdot),u(\cdot),v(\cdot)} & \sup\limits_{\mathbb{P}\in\mathbb{F}} \mathrm{E}_{\mathbb{P}}\left(u(\tilde{z}) + v(\tilde{z})\right) \\
\text{s.t.} & u(\tilde{z}) - v(\tilde{z}) = y(\tilde{z}) - \tilde{z} \\
& 0 \leq y(\tilde{z}) \leq 1 \\
& u(\tilde{z}), v(\tilde{z}) \geq 0 \\
& y, u, v \in \mathcal{Y}\left(1, 1, \{1\}\right)
\end{array}
$$

Solution

```
y_sol =

-0.000 + 1.000*z1 + 1.00(-0.000 + 1.000*z1)^- - 1.00(1.000 - 1.000*z1)^-
```

$$
\begin{aligned}
\hat{y}_{sol}(\tilde{z}) &= \tilde{z} + \tilde{z}^{-} - (1 - \tilde{z})^{+} \\
&= \tilde{z}^{+} - (1 - \tilde{z})^{+} \\
&= \begin{cases} \tilde{z} & \text{if } 0 \leq \tilde{z} \leq 1 \\ 0 & \text{if } \tilde{z} \leq 0 \\ 1 & \text{if } \tilde{z} \geq 1 \end{cases}
\end{aligned}
$$



Outline

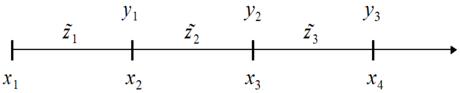
Robust Inventory Model

- Model a distribution-free, multi-period, inventory control problem with service constraints.
- Demand is exogenous, with unknown, but partially characterized distribution in a family $\mathbb{F}$, defined by:
  - Covariance Matrix: Temporal demand correlation can be modeled by a non-diagonal covariance matrix.
  - Mean: Assume fixed mean (for simplicity).
  - Support: Maximum demand in each period.
- Backorders allowed, but in some applications, a penalty cost might not be a good model for stockouts.
  - In our model, we avoid stockouts with a constraint on the fill-rate.
  - Fill-rate constraint acts as a service guarantee to the consumers.

Model Parameters

Parameters

Num Periods	:	$T \in \mathbb{N}$
Order Cost	:	$\boldsymbol{c} \in \Re^T$
Holding Cost	:	$\boldsymbol{h} \in \Re^T$
Min Fill Rate	:	$\boldsymbol{\beta} \in \Re^T$
Max Order Qty	:	$\boldsymbol{x}^{MAX} \in \Re^T$



Uncertainties

Demand	:	$\tilde{\boldsymbol{z}} \in \Re^T$

Decisions

Order Quantity	:	$\boldsymbol{x}(\tilde{\boldsymbol{z}}) : x_t(\tilde{\boldsymbol{z}}) \in \mathcal{L}(1, T, [t-1])$
Inventory Level	:	$\boldsymbol{y}(\tilde{\boldsymbol{z}}) : y_t(\tilde{\boldsymbol{z}}) \in \mathcal{L}(1, T, [t])$

Robust Fill Rate Constraint

$$\text{Fill Rate} = \frac{\text{Expected Sales}}{\text{Expected Demand}} \geq \beta$$

- Using our notation, the robust (worst-case) version:

$$\inf_{\mathbb{P} \in \mathbb{F}} \mathrm{E}_{\mathbb{P}} \left(\min \left\{ \tilde{z}_t, y_{t-1}(\tilde{\boldsymbol{z}}) + x_t(\tilde{\boldsymbol{z}}) \right\} \right) \geq \beta_t \mu_t$$

- Apply inventory balance equation and re-arrange:

$$\sup_{\mathbb{P} \in \mathbb{F}} \mathrm{E}_{\mathbb{P}} \left(y_t(\tilde{\boldsymbol{z}})^- \right) \leq (1 - \beta_t) \mu_t$$

Model Formulation

- Family of uncertainties:

$$\mathbb{F} = \left\{ \mathbb{P} : \mathrm{E}_{\mathbb{P}}\left(\tilde{\boldsymbol{z}}\right) = \boldsymbol{\mu}, \mathrm{E}_{\mathbb{P}}\left(\tilde{\boldsymbol{z}}\tilde{\boldsymbol{z}}'\right) = \boldsymbol{\Sigma} + \boldsymbol{\mu}\boldsymbol{\mu}', \mathbb{P}\left(0 \leq \tilde{\boldsymbol{z}} \leq \boldsymbol{z}^{MAX}\right) = 1 \right\}$$

- Robust Inventory Model with Fill Rate constraints:

$$\begin{array}{ll} \displaystyle\min_{\boldsymbol{x}(\cdot),\boldsymbol{y}(\cdot)} & \displaystyle\sup_{\mathbb{P}\in\mathbb{F}} \mathrm{E}_{\mathbb{P}}\left(\boldsymbol{c}'\boldsymbol{x}(\tilde{\boldsymbol{z}}) + \boldsymbol{h}'\left(\boldsymbol{y}(\tilde{\boldsymbol{z}})\right)^{+}\right) \\ \text{s.t.} & \displaystyle\sup_{\mathbb{P}\in\mathbb{F}} \mathrm{E}_{\mathbb{P}}\left(\boldsymbol{y}(\tilde{\boldsymbol{z}})^{-}\right) \leq \left(\boldsymbol{I} - \textbf{diag}\left(\boldsymbol{\beta}\right)\right)\boldsymbol{\mu} \\ & \boldsymbol{D}\boldsymbol{y}\left(\tilde{\boldsymbol{z}}\right) - \boldsymbol{x}\left(\tilde{\boldsymbol{z}}\right) = -\tilde{\boldsymbol{z}} \\ & \boldsymbol{0} \leq \boldsymbol{x}\left(\tilde{\boldsymbol{z}}\right) \leq \boldsymbol{x}^{MAX} \\ & x_t \in \mathcal{L}(1, T, [t-1]) \quad \forall t \in [T] \\ & y_t \in \mathcal{L}(1, T, [t]) \quad \forall t \in [T] \end{array}$$

$$\boldsymbol{D} = \begin{bmatrix} 1 & 0 & 0 & \ldots & 0 \\ -1 & 1 & 0 & \ldots & 0 \\ 0 & -1 & 1 & \ldots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \ldots & 1 \end{bmatrix}$$

Model Transformed into DRO Framework

- After linearizing piecewise-linear terms,

$$
\begin{array}{lll}
\min\limits_{\boldsymbol{x}(\cdot),\boldsymbol{y}(\cdot),\boldsymbol{r}(\cdot),\boldsymbol{s}(\cdot)} & \sup\limits_{\mathbb{P}\in\mathbb{F}} \mathrm{E}_{\mathbb{P}} \left(\boldsymbol{c}'\boldsymbol{x}(\tilde{\boldsymbol{z}}) + \boldsymbol{h}'\boldsymbol{r}(\tilde{\boldsymbol{z}})\right) & \\
\text{s.t.} & \sup\limits_{\mathbb{P}\in\mathbb{F}} \mathrm{E}_{\mathbb{P}} \left(\boldsymbol{s}(\tilde{\boldsymbol{z}})\right) \leq (\boldsymbol{I} - \textbf{diag}\,(\boldsymbol{\beta}))\boldsymbol{\mu} & \\
& \boldsymbol{r}\left(\tilde{\boldsymbol{z}}\right) \geq \boldsymbol{y}\left(\tilde{\boldsymbol{z}}\right) & \\
& \boldsymbol{s}\left(\tilde{\boldsymbol{z}}\right) \geq -\boldsymbol{y}\left(\tilde{\boldsymbol{z}}\right) & \\
& \boldsymbol{r}\left(\tilde{\boldsymbol{z}}\right), \boldsymbol{s}\left(\tilde{\boldsymbol{z}}\right) \geq \boldsymbol{0} & \\
& \boldsymbol{D}\boldsymbol{y}\left(\tilde{\boldsymbol{z}}\right) - \boldsymbol{x}\left(\tilde{\boldsymbol{z}}\right) = -\tilde{\boldsymbol{z}} & \\
& \boldsymbol{0} \leq \boldsymbol{x}\left(\tilde{\boldsymbol{z}}\right) \leq \boldsymbol{x}^{MAX} & \\
& x_t \in \mathcal{L}(1, T, [t-1]) & \forall t \in [T] \\
& r_t, s_t, y_t \in \mathcal{L}(1, T, [t]) & \forall t \in [T]
\end{array}
$$

ROME Code for Inventory Problem

```
% model parameters
T = 10;                    % planning horizon
c = 1 *ones(T, 1);         % order cost rate
hcost = 2*ones(T, 1);      % holding cost rate
beta  = 0.50*ones(T, 1); % minimum fillrate in each period
xMax  = 100*ones(T, 1);  % maximum order quantity in each period
alpha = 0.5;               % temporal autocorrelation factor
L = alpha * tril(ones(T), -1) + eye(T); % autocorrelation matrix

% numerical uncertainty parameters
zMax  = 105*ones(T, 1);    % maximum demand in each period
zMean = 30*ones(T, 1);     % mean demand in each period
zCovar = 20*(L * L');      % temporal demand covariance

% differencing matrix
D = eye(T) - diag(ones(T-1 , 1), -1);

% dependency structure
pX = logical([tril(ones(T)), zeros(T, 1)]);
```

ROME Code for Inventory Problem (cont.)

```
% Step 3: BDLDR Method
% --------------------
h = rome_begin('Robust Inventory (BDLDR)');
tic;

% declare uncertainties
newvar z(T) uncertain nonneg;

% define uncertainty parameters
rome_constraint(z <= zMax);   % support
z.set_mean(zMean);            % mean
z.Covar = zCovar;             % covariance

% define LDR variables
newvar x(T, z, 'Pattern', pX) linearrule;  % order quantity
newvar y(T, z) linearrule;                 % inventory level

% define auxilliary variables
newvar r(T, z) s(T, z) linearrule nonneg;
```

$$\mathbb{F} = \left\{ \mathbb{P} : \begin{array}{l} \mathbb{P}\left(0 \le \tilde{\boldsymbol{z}} \le \boldsymbol{z}^{MAX}\right) = 1 \\ \mathrm{E}_{\mathbb{P}}\left(\tilde{\boldsymbol{z}}\right) = \boldsymbol{\mu}, \\ \mathrm{E}_{\mathbb{P}}\left(\tilde{\boldsymbol{z}}\tilde{\boldsymbol{z}}'\right) = \boldsymbol{\Sigma} + \boldsymbol{\mu}\boldsymbol{\mu}' \end{array} \right\}$$

$$\begin{array}{ll} \boldsymbol{r}\left(\tilde{\boldsymbol{z}}\right), \boldsymbol{s}\left(\tilde{\boldsymbol{z}}\right) \ge \mathbf{0} & \\ x_t \in \mathcal{L}(1, T, [t-1]) & \forall t \in [T] \\ r_t, s_t, y_t \in \mathcal{L}(1, T, [t]) & \forall t \in [T] \end{array}$$

ROME Code for Inventory Problem (cont.)

```
% auxilliary constraints
rome_constraint(r >=  y); % since r >= y^+
rome_constraint(s >= -y); % since s >= y^-

% fillrate constraint
rome_constraint(mean(s) <= diag(ones(T, 1) - beta) * zMean);

% inventory balance constraint
rome_constraint(D*y == x - z);

% order quantity constraints
rome_box(x, 0, xMax);

% objective
rome_minimize(c'*mean(x) + hcost'*mean(r));

% solve and display optimal objective
h.solve_deflected;
disp(sprintf('BDLDR Obj = %0.2f, time = %0.2f secs', h.ObjVal, toc));
x_sol_bdldr = h.eval(x)

rome_end;
```

$$
\begin{array}{cl}
\min\limits_{\boldsymbol{x}(\cdot),\boldsymbol{y}(\cdot),\boldsymbol{r}(\cdot),\boldsymbol{s}(\cdot)} & \sup\limits_{\mathbb{P}\in\mathbb{F}} \mathrm{E}_{\mathbb{P}}\left(\boldsymbol{c}'\boldsymbol{x}(\tilde{\boldsymbol{z}}) + \boldsymbol{h}'\boldsymbol{r}(\tilde{\boldsymbol{z}})\right) \\
\text{s.t.} & \sup\limits_{\mathbb{P}\in\mathbb{F}} \mathrm{E}_{\mathbb{P}}\left(\boldsymbol{s}(\tilde{\boldsymbol{z}})\right) \leq \left(\boldsymbol{I} - \mathbf{diag}\left(\boldsymbol{\beta}\right)\right)\boldsymbol{\mu} \\
& \boldsymbol{D}\boldsymbol{y}\left(\tilde{\boldsymbol{z}}\right) - \boldsymbol{x}\left(\tilde{\boldsymbol{z}}\right) = -\tilde{\boldsymbol{z}} \\
& \boldsymbol{r}\left(\tilde{\boldsymbol{z}}\right) \geq \boldsymbol{y}\left(\tilde{\boldsymbol{z}}\right) \\
& \boldsymbol{s}\left(\tilde{\boldsymbol{z}}\right) \geq -\boldsymbol{y}\left(\tilde{\boldsymbol{z}}\right) \\
& \mathbf{0} \leq \boldsymbol{x}\left(\tilde{\boldsymbol{z}}\right) \leq \boldsymbol{x}^{MAX}
\end{array}
$$

Numerical Example (I)

- Use a covariance matrix which represents temporal autocorrelation of demand, $\boldsymbol{\Sigma} = \sigma \boldsymbol{L}(\alpha)\boldsymbol{L}(\alpha)'$

$$\boldsymbol{L}(\alpha) = \begin{bmatrix} 1 & 0 & 0 & \dots & 0 \\ \alpha & 1 & 0 & \dots & 0 \\ \alpha & \alpha & 1 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \alpha & \alpha & \alpha & \dots & 1 \end{bmatrix}$$

- $(z^{MAX}, \beta) = (100, 0.5)$

h	μ	α	$Z_{LDR}{}^{(2)}$	Z_{LDR}	Z_{BDLDR}	h	μ	α	$Z_{LDR}{}^{(2)}$	Z_{LDR}	Z_{BDLDR}
1	10	0.00	590.0	106.0	106.0	3	10	0.00	1490.0	126.0	126.0
1	10	0.20	590.0	108.2	108.1	3	10	0.20	1490.0	131.8	131.5
1	10	0.50	590.0	119.9	117.2	3	10	0.50	1490.0	163.0	155.9
1	10	0.90	590.0	153.3	135.8	3	10	0.90	1490.0	251.8	205.4

Numerical Example (II)

- $(z^{MAX}, \beta) = (200, 0.5)$

h	μ	α	$Z_{LDR}^{(2)}$	Z_{LDR}	Z_{BDLDR}	h	μ	α	$Z_{LDR}^{(2)}$	Z_{LDR}	Z_{BDLDR}
1	10	0.00	$+\infty$	121.0	107.0	3	10	0.00	$+\infty$	166.1	128.0
1	10	0.20	$+\infty$	167.9	109.3	3	10	0.20	$+\infty$	285.9	133.8
1	10	0.50	$+\infty$	313.6	119.4	3	10	0.50	$+\infty$	654.1	159.9
1	10	0.90	$+\infty$	722.3	141.6	3	10	0.90	$+\infty$	1668.7	216.7

- $(z^{MAX}, \beta) = (200, 0.8)$

h	μ	α	$Z_{LDR}^{(2)}$	Z_{LDR}	Z_{BDLDR}	h	μ	α	$Z_{LDR}^{(2)}$	Z_{LDR}	Z_{BDLDR}
1	10	0.00	$+\infty$	163.4	129.5	3	10	0.00	$+\infty$	276.6	185.8
1	10	0.20	$+\infty$	285.3	136.0	3	10	0.20	$+\infty$	588.2	202.4
1	10	0.50	$+\infty$	722.1	161.5	3	10	0.50	$+\infty$	1681.6	267.8
1	10	0.90	$+\infty$	3402.0	2563.5	3	10	0.90	$+\infty$	9192.7	7019.9

ROME Output (MOSEK Solver)

```
EDU>> inventory_fillrate_example
Status: OPTIMAL
LDR Obj = 1747.50, time = 0.25 secs

x_sol_ldr =

 97.500
 0.000 + 0.952*z1
-0.000 + 0.007*z1 + 0.945*z2
 0.000 + 0.005*z1 + 0.010*z2 + 0.937*z3
-0.000 + 0.004*z1 + 0.006*z2 + 0.013*z3 + 0.929*z4
 0.000 + 0.003*z1 + 0.005*z2 + 0.008*z3 + 0.015*z4 + 0.921*z5
-0.000 + 0.004*z1 + 0.004*z2 + 0.006*z3 + 0.009*z4 + 0.018*z5 + 0.911*z6
-0.000 + 0.003*z1 + 0.005*z2 + 0.006*z3 + 0.008*z4 + 0.012*z5 + 0.023*z6 + 0.897*z7
 0.000 + 0.004*z1 + 0.005*z2 + 0.006*z3 + 0.009*z4 + 0.011*z5 + 0.016*z6 + 0.031*z7 + 0.870*z8
-0.000 + 0.007*z1 + 0.007*z2 + 0.009*z3 + 0.011*z4 + 0.015*z5 + 0.019*z6 + 0.031*z7 + 0.058*z8 + 0.795*z9

Status: NEAR_OPTIMAL
BDLDR Obj = 300.48, time = 0.58 secs

x_sol_bdldr =

 15.333
-0.000 + 1.003*z1 + 1.00(-0.000 + 1.003*z1)^- - 1.00(100.000 - 1.003*z1)^-
-0.000 + 0.000*z1 + 1.004*z2 + 1.00(-0.000 + 0.000*z1 + 1.004*z2)^- - 1.00(100.000 - 0.000*z1 - 1.004*z2)^-
-0.000 + 0.000*z1 - 0.000*z2 + 1.004*z3 + 1.00(-0.000 + 0.000*z1 - 0.000*z2 + 1.004*z3)^- - 1.00(100.000 - 0.00
-0.000 - 0.000*z1 - 0.000*z2 + 0.000*z3 + 1.004*z4 + 1.00(-0.000 - 0.000*z1 - 0.000*z2 + 0.000*z3 + 1.004*z4)^-
 0.000 - 0.000*z1 - 0.000*z2 - 0.000*z3 + 0.000*z4 + 1.004*z5 + 1.00(0.000 - 0.000*z1 - 0.000*z2 - 0.000*z3 + (
-0.000 + 0.000*z1 - 0.000*z2 + 0.000*z3 + 0.000*z4 - 0.000*z5 + 1.004*z6 + 1.00(-0.000 + 0.000*z1 - 0.000*z2 +
 0.000 - 0.000*z1 - 0.000*z2 + 0.000*z3 + 0.000*z4 - 0.000*z5 - 0.000*z6 + 1.004*z7 + 1.00(0.000 - 0.000*z1 - (
-0.000 + 0.000*z1 - 0.000*z2 + 0.000*z3 + 0.000*z4 + 0.000*z5 - 0.000*z6 - 0.000*z7 + 1.004*z8 + 1.00(-0.000 +
-0.000 + 0.000*z1 + 0.000*z2 - 0.000*z3 + 0.000*z4 + 0.000*z5 + 0.000*z6 + 0.000*z7 + 0.000*z8 + 1.005*z9 + 1.(
```

Outline

Summary

- We have seen how ROME can be used to model Distributionally Robust Optimization problems
- ROME contains some components common to most modeling languages, and unique features for Robust Optimization
- Service-constrained inventory control example