MATLAB to AxiomaPortfolio™ API Tutorial

April 28, 2014

Contents

1 Introduction

We will present you a simple example throughout this document to show how to build optimized portfolio using Axioma**Portfolio**™ from MATLAB. Basically, you need to follow the following steps:

1. Set up a dataset (create an `AXDataSet` object) which should include your alpha information.
2. Create a set of objectives and a set of constraints, which make up an `AXStrategy` object.
3. Create an `AXRebalancing` object which specifies some basic parameters (like riskmodel) used in portfolio optimization.
4. Create an `AXOptimization` object from the `AXDataSet` in step 1, `AXStrategy` in step 2, and `AXRebalancing` in step 3.
5. Run the `AXOptimization` object to get optimization result (also some analytical stuff).

We will discuss these steps in more detail in the next few subsections.

2 Set Up a Dataset

In MATLAB, a *dataset* is a collection of `xts` or `xts` variants (i.e., `Model` objects).

2.1 Types of Data Elements

Dataset is represented by `AXDataSet` class, which contains 5 types of data:

- **Asset Group.** A set of assets with associated values (e.g., weight, price, country, etc.) Value can be numerical or text (char-string). For numerical group, it can be Account or Benchmark or just simple Group. For text-type group, it can be used as a mapping between assets and text values. So the total possible types of a group can be:

  `'ACCOUNT'`
  : Numerical, representing portfolio account. To specify a group as a `'ACCOUNT'`, Axioma**Portfolio**™ will impose some restrictions on it; for example, if portfolio is long-only, then the value associated with each asset can not be negative.

  `'BENCHMARK'`
  : Numerical, representing benchmark weight associated with assets. Axioma**Portfolio**™ will impose some restrictions on benchmark; e.g., summation of all values should be 1.

  `'TEXTGROUP'`
  : String, representing some properties of associated assets, such as countries, sectors, etc.

  `'ASSETMAP'`
  : String, representing some properties of associated assets, but require values and assets must be 1-to-1 mapped; i.e., the same value can be assigned to different assets.

  `'GROUP'`
  : Numerical, just giving a value to each assets in the group.

  Asset group is created as a `myfints` and added to dataset by specifying it type. See examples below.

- **Metagroup.** A metagroup consists of multiple simple groups. You can simply specify a `myfints` object as `'METAGROUP'`. Later when MATLAB passing it to Axioma**Portfolio**™, MATLAB will create simple groups based on the values of the `myfints` (i.e., assets with the same value consists of a simple group) and these simple groups consists of the metagroup.

- **Set.** An asset set can be made up of assets, other asset sets, groups, and/or metagroups. You create an asset set by setting up a `myfints` whose value are either 1s or 0s indicating if corresponding thing is in this asset set, where corresponding thing refers to the field name with which the value is associated. So the field names of this kind of `myfints` may be different from others, for it can be asset names, group names or other asset names already exist.

- **Risk Model.** An object of class `RiskModel` or its descendants. Currently we implemented 3 risk models: `RiskBarra`, `RiskEMA`, and `RiskDummy`. Suppose T is the number of periods, N is the number of assets, and M is the number of factors, a risk model object contains the following fields you can access (but can not modified once it's created):

  `faccov`
  : `xts`, $T \times M \times M$, factor covariance matrix for each period.

  `exposure`
  : `xts`, $T \times N \times M$, asset exposure to factors.

  `specrisk`
  : `myfints`, $T \times N$, asset specific risks.

  `beta`
  : `myfints`, $T \times N$, asset beta.

  `facids`
  : M cell vector of strings, containing factor names (the same as the field names from `faccov` or `exposure` (3rd dimension).)

  Suppose `rm` is a risk model object, the following code calculates covariance matrix of factors for period t:

```
exposure = squeeze(rm.exposure(t,:,:));
sigma = exposure * squeeze(rm.faccov(t,:,:)) * exposure';
```

  If we are given asset weights as `w`, then we can calculate portfolio risk at period t as:

```
specrisk = fts2mat(w .* rm.specrisk(t,:));
w = fts2mat(w);
rsk(t) = sqrt(w * sigma * w' + specrisk * specrisk');
```

  In MATLAB, you should set up a risk model this way:

```
RiskBarra('RISK_BARRA', ds);  % ds is an AXDataSet object, 'RISK BARRA' is the ID name of risk model
```

  where `ds` is the `AXDataSet` object to which you want to add this risk model, `'RISK BARRA'` is just a name served as identifier in the dataset. Once the risk model is added to the dataset, you can access it by referencing its id.:

```
rm = ds('RISK BARRA'); % 'RISK BARRA' is the id of the risk model
```

- **Transaction Model.** An object of class `TCModel` or its descendants. Currently we implemented 3 transaction models: `TCFlat`, `TCSimple`, and `TCQSG`. `TCFlat` applies a fixed transaction cost percent to all transactions, `TCSimple` applies piece-wise constant transaction cost percent based on transaction volume size, and `TCQSG` uses a non-linear transaction cost model from QSG (a vendor name?)

  You create a transaction model in a similar manner as risk model, like

```
TCQSG('TCM_QSG', ds, 0.0005, 'USD');
```

  where the first 2 parameters are the same as that in risk model, the 3rd parameter is a fixed transaction cost percent required by an internal store procedure, the 4th is curreny symbol also required by the store prrocedure. You can access the transaction model by indexing its id, just like way for risk model, though it rarely you really need to do so.

2.2 Create and Setup DataSet

Having introduced element type inside a dataset, it's time to create a dataset object, as shown below

```
ds = AXDataSet(true, dates, '00053');
```

where the 1st parameter indicates if this is live or backtest, the 2nd is a vector of dates, and the 3rd is the `aggids` made up the universe. Note that it is allowed to have multiple `aggids` to be passed like

```
ds = AXDataSet(false, dates, {'00053', '000524248'});
```

Once you have an `AXDataSet` object on hand, the benchmark weight for each `aggid` is automatically added to the dataset, with id of something like `'X00053'`; that is, you can get benchmark weight by

```
bm = ds('X00053');
```

Other two elements automatically added to dataset once it's created are `'TICKER'` (as an `'ASSETMAP'`) and `'ROUNDLOTSIZE'` (as a `'GROUP'`).

Then now you can add other things into the dataset just created. `AXDataSet` provides following methods for loading data of different types:

```
fts = loadSecTS(o, itemid, backfilldays, unit)  % load data from quantstaging.dbo.secTS
fts = loadRawTS(o, itemid, backfilldays, unit)  % load data using DataQA.API
fts = loadFactor(o, facid, backfilldays, unit)  % load factors populated
fts = loadAlpha(o, strategyid)                  % load alpha
addAccountRestricted(o,prefix,accList)          % add restricted accounts to dataset o
addCustomRestricted(o,prefix,cusType,typeName)  % add customized restricted accounts to o
addAccount(o, id, strategyid, acctid)           % add an account to the dataset o
addAlpha(o, id, strategyid)                     % add alpha to the dataset o
```

These functions fall into categories: the `loadxxxx` category just load data and return it in a `myfints` object, while the `addxxxx` category load and add data to the dataset which is the first parameter(`o`). Basically, these functions are same as those found in Factor Toolbox, except they go further by allowing user specifying `backfilldays` and `unit`. You don't need to specify the frequency of the data to be loaded, since dataset itself keep a date series and when needed, the data in the dataset will be aligned to the date series. **Unit** is a concept introduced by Axioma**Portfolio**™ and adapted by `xts` and `myfints`. Here's its definition in MATLAB:

```
classdef Unit
    enumeration
        CURRENCY, NUMBER, PERCENT, PRICE, SHARES, TEXT
    end
end
```

You should specify a unit using syntax like:

```
Unit.CURRENCY
```

Don't overlook the role of unit. If you specify a wrong syntax, Axioma**Portfolio**™ may complain and fail you program. For instance, Axioma**Portfolio**™ checks data with Unit.SHARES should be integer type.

Instead of using functions above to load/add data, you can add any data you have to a dataset. Three data types allowed in `AXDataSet`: `xts` (including `myfints`), `AXAttr`, and `Model`. An `AXAttr` actually is an `xts` plus type (`'GROUP'`,`'METAGROUP'`,`'SET'`,...) information. `Model` refers to risk model or transaction model we mentioned earlier. When you add an `xts` object to a dataset, it will be added as a `'GROUP'` type. You can add an `xts` or `AXAttr` object this way (suppose `ds` is an `AXDataSet` object):

```
1 % add myfints_obj to ds as a 'GROUP'.
2 ds('DATA_ID') = myfints_obj;
3 ds('DATA_ID') = AXAttr('DATA_ID', myfints_obj, 'GROUP');
4 ds.add('DATA_ID', myfints_obj, 'GROUP');
```

All the 3 lines are equivalent. In most cases, you just want to add a `myfints` as a `'GROUP'`, so you can use the syntax in line 1. Line 2 and line 3 allow you add data as types other than `'GROUP'`, and line 3 avoid writing data id twice. Here's more examples:

```
ds.add('DATA_ID1', myfints_obj1, 'METAGROUP');
ds.add('DATA_ID2', myfints_obj2, 'SET');
ds.add('DATA_ID3', myfints_obj3, 'TEXTGROUP');
```

You should not use this syntax to add a model to a dataset. Using the method we already introduced before, like

```
RiskBarra('RISK_BARRA', ds); % you can give an id whatever you want
RiskEMA('RISK_EMA', ds);
RiskDummy('RISK_EMA', ds);
```

Finally, a simplified but complete example to build a dataset:

```
dates = genDataSeries('2005-12-31', '2012-04-30', 'M');
ds = AXDataSet(true, dates, '00053');
ds.addAccount('ACCT_FAC_SP500', 'FAC_SP500', 'All');
ds.addAlpha('ALPHA_FAC_SP500', 'FAC_SP500');
RiskBarra('RISK_BARRA', ds);
TCQSG('TCM_QSG', ds, 0.0005, 'USD');

ds('PRICE')  = ds.loadSecTS(1051, 7, Unit.PRICE);  % Adjusted Close Price of Stock, denoted in USD
ds('VWAP')   = ds.loadSecTS(1151, 7, Unit.PRICE);  % Volume Weighted Average Price (VWAP)
ds('SHARES') = ds.loadSecTS(159,  7, Unit.SHARES); % adjusted outstanding shares of stock
ds('TRDVOL') = ds.loadSecTS(158,  7, Unit.SHARES); % adjusted trading volume of stock

ds('VALUE')    = ds.loadFactor('F00001', 7);
ds('MOMENTUM') = ds.loadFactor('F00073', 7);

price  = ds('PRICE');  price.unit = Unit.NUMBER;
trdvol = ds('TRDVOL'); trdvol.unit = Unit.NUMBER;
[price, trdvol] = aligndates(price, trdvol, price.freq);
trdvol = price .* trdvol;

% avg trding volume between 5 busdays ago and this rebalance
ds('VOL5D') = ftsmovavg(trdvol, 5);

% avg trding volume between last rebalance and this rebalance
ds('VOL1P') = aligndates(trdvol, ds.dates, 'CalcMethod', 'simavg');

% forward returns between rebalances
price = aligndates(price, ds.dates);
ds('FWDRTN') = leadts(price,1) ./ price - 1;

% 5 percent of outstanding shares
ds('SHARES5PCT') = ds('SHARES')*0.05;
```

IDs of data in a dataset are going to be used in constructing objectives and constraints.

2.3 Manipulate Dataset

In the previous subsection, you already see how to add/remove/modify data elements of a dataset. There's other functions that may be useful for you to probe a dataset.

The first one is

```
ids = keys(o, dates);
```

which returns all identifiers of data elements in dataset `o` over `dates`. `dates` can be multiple. If not provided, it uses all dates from the dataset.

The second is

```
ids = value(o, type);
```

which returns in a cell vector all data elements of `type` specified as second parameter. `type` must be a char-string of type identifier of data elements, as we discussed in section 2.1, something like `'GROUP'`, `'METAGROUP'`, `'SET'`, `'TEXTGROUP'`, etc, or plus a `'~'` before the type string like `'~GROUP'`, `'~SET'`, etc, which returns all data elements whose type are not `'GROUP'`, `'~SET'`, etc.

Thers are also functions to manipulate the universe:

```
secids = getUniverse(o, dates);  % return all securities in universe over specified dates
                                 % by DEFAULT dates uses all dates in the dataset.
fts = applyUniverse(o, fts);     % set myfints object fts such that non-universe things to be NaNs.
fts = trimUniverse(o, fts);      % fts is a logical myfints object, excluding things with false value
                                 % in fts from universe of the dataset
```

3 Set Up Constraints

Class `AXConstraint` represents a constraint, and a cell vector of `AXConstraint` objects consist of the whole set of constraints used in optimization. Like data elements in a dataset, every constraint must be given a id and type. That's the parameters only needed to create an `AXConstraint` object:

```
c = AXConstraint(id, type);
```

Parameter `id` can be any string but unique among all things that needs an id (i.e., data elements, constraints, objectives). Parameter `type` indicates type of the constraint, corresponding to Axioma**Portfolio**™'s constraint classes which will be described below in detail along with constraint properties. Constraint properties are things associated with a constraint specifying the constraint's scope, unit, parameters, etc.

3.1 Constraint Scopes

`'SCOPE'` indicates the extent to which a constraint is applied in Axioma**Portfolio**™. For example, when a constraint is constructed with a group of assets, the constraint could be applied to the group as a whole or to each individual asset in the group. The constraint scope helps resolve this apparent ambiguity.

There are 4 types of scopes in Axioma**Portfolio**™:

- `ASSET`: a separate constraint is constructed for each asset in the specified group of assets.
- `AGGREGATE`: a single constraint is constructed for all of assets in the selected group or groups of assets.
- `SELECTION`: a separate constraint is constructed for each of the selection.
- `MEMBER`: a separate constraint is constructed for each of the first level children of the selection.

3.2 Constraint Units

Constraint units are specified as a value of class `Unit` and can only take values of `CURRENCY`, `NUMBER`, `PERCENT`, `PRICE`, `SHARES`.

3.3 Other Constraint Properties

There are other properties that may apply to a constraint. Table 1 summaries the properties available for each type of constraint. The first column of the table also gives all possible constraint types.

For a constraint, say `c`, you can set its properties this way:

```
c = AXConstraint('AbsHoldingMCap', 'LimitAbsoluteHoldingConstraint'); % 1st is id, 2nd is type
c.SCOPE = 'ASSET';
c.UNIT  = Unit.SHARES;
c.MAX_VALUES_GROUP = 'SHARES5PCT'; % reference to a data element in dataset
```

Actually, you can operate a property of constraint just like a matrix. For example, if a property has been set, like `c.MAX_VALUES_GROUP` in the above code snippet, following code show how to modify/remove it from the constraint.

```
c.MAX_VALUES_GROUP = 'VOL1D';  % change the value to 'VOL1D' from 'SHARES5PCT'
c.MAX_VALUES_GROUP = [];       % remove the property from c
```

Table 1: Constraint Properties

Constraint Type	ALPHA_UNCERTAINTY_MODEL*	BASE_SET	BENCHMARK*	ETA	EXCLUDE_CAP_GAINS	FACTOR_WEIGHT	GRANDFATHER_BOUNDS	INCLUDE_LONG_TERM	INCLUDE_SHORT_TERM	KAPPA	MAX	MAX_VIOLATION	MIN	NUM_OF_DAYS	PENALTY	PENALTY_TYPE	QUALIFICATION	RISK_UNCERTAINTY_MODEL*	RISKMODEL*	SCOPE	SPECIFIC_WEIGHT	UNIT	USE_BUDGET_VALUE	WEIGHTED	SELECTION
BudgetConstraint											X		X				X					X	X		
LimitAbsoluteHoldingConstraint			X								X	X			X	X				X		X			X
LimitAlmostLongTermGainsConstraint												X		X	X	X				X					X
LimitBuyConstraint											X	X			X	X				X		X			X
LimitHoldingConstraint			X				X				X	X	X		X	X				X		X			X
LimitLongHoldingConstraint			X				X				X	X	X		X	X				X		X			X
LimitLongNamesConstraint											X	X	X		X	X				X					X
LimitLongShortRatioConstraint											X		X							X					X
LimitMinimumHoldingPeriod												X		X	X	X				X					X
LimitModelDeviationConstraint			X								X	X			X	X				X		X			X
LimitNamesConstraint											X	X	X		X	X				X					X
LimitNetTaxGainsConstraint					X			X	X		X	X	X		X	X						X		X	
LimitNetTaxLossesConstraint					X			X	X		X	X	X		X	X						X		X	
LimitNumBuyTradesConstraint											X	X			X	X				X					X
LimitNumSellTradesConstraint											X	X			X	X				X					X
LimitNumTradesConstraint											X	X			X	X				X					X
LimitRelativeMarginalContributionToRiskConstrair		X	X								X								X	X					X
LimitSellConstraint											X	X			X	X				X		X			X
LimitShortHoldingConstraint			X				X				X	X	X		X	X				X		X			X
LimitShortNamesConstraint											X	X	X		X	X				X					X
LimitTaxLiabilityConstraint					X						X	X			X	X						X			
LimitTotalRiskConstraint			X			X					X	X			X	X		X	X	X	X	X			X
LimitTradeConstraint											X	X	X		X	X				X		X			X
LimitTurnoverConstraint											X	X			X	X				X		X			X
LimitWeightedAvgConstraint		X	X								X		X							X		X			X
ProbabilisticConstraint	X	X		X						X	X	X	X		X	X				X		X			X
RobustConstraint	X	X	X	X						X	X	X	X		X	X				X		X			X
ThresholdHoldingConstraint							X						X							X		X			X
ThresholdTradeConstraint													X							X		X			X

Aside from properties listed in the table, you can also inquire/set/reset general things of an `AXConstraint` object. Here's the list:

`id` (Only Readable) Identifier of the constraint

`type` (Only Readable) Type of the constraint. See the 1st column of the table.

`desc` (Only Readable) Description of the constraint.

`isEnabled`
If the constraint is enabled/disabled.

`selection`
If the constraint is selection-type (not collective), it's the id of the selected thing (like data element, risk model). *You can not set `selection` field for a collective constraint* (see the next subsection and the last column of table 1.)

`priority`
An integer indicates the constraint's priority in portfolio optimization.

3.4 Constraints

This section will discuss different type of constraints. In Axioma**Portfolio**™, constraints are classified as either *Collective* constraints or as *Selection* constraints. **Collective** constraints are constraints that implicitly apply to all of the assets in a rebalancing account as whole, while **Selection** constraints are constraints that apply to a selected group of assets. The last column of table 1 indicates which constraints are collective and which are selection.

BudgetConstraint

limits the total value of the optimized portfolio. A budget constraint requires that the total value of the optimized portfolio (the sum of all asset values) equals the total value of all holdings in the portfolio (including cash).

The total value of the portfolio usually consists of the initial holdings and potentially cash flows in or out of the portfolio. The budget constraint allows the user to specify exactly the total value of the optimized portfolio. For instance, if a portfolio has value \$100,000 and an additional \$10,000 of cash is added to it then the total value, or the budget, is \$110,000. Conversely, if \$20,000 in cash is to be extracted from the portfolio, then the total value or budget is \$80,000.

LimitAbsoluteHoldingConstraint

limits the absolute (active) holdings for individual assets or a group of assets. This type of constraint can be used to set an upper bound on the absolute (active) asset holdings.

The constraint has the form:

$$\sum_{i \in A} \lambda_i |w_i| \leq rhs,$$

where A is the set of assets, λ_i and w_i represent the weight and holdings for asset i respectively, and rhs is the bound to be imposed. The weights λ_i are required to be non-negative.

If the `BENCHMARK` property is set, then the constraint is over the active holdings with respect to the benchmark, i.e., its left-hand side has the form

$$\sum_{i \in A} \lambda_i |w_i - b_i|$$

where b_i is the benchmark holding for asset i.

LimitAlmostLongTermGainsConstraint

can be used to prohibit trading lots with positive holdings which satisfy the following conditions:

1. Their price has increased since their purchase (i.e. they have unrealized gains).
2. They are still short-term lots.
3. They will become long-term lots within a specified number of days.

LimitBuyConstraint

limits the size of a transaction representing a purchase. A `LimitBuyConstraint` imposes an upper bound on the buy transactions of an asset or a weighted group of assets.

This constraint has the form:

$$\sum_{i \in A} \lambda_i t_t^+ \leq rhs,$$

where A is the set of assets, λ_i and t_i^+ represent the weight and transaction for asset i respectively, and rhs is the bound to be imposed. The weights λ_i are required to be non-negative.

LimitHoldingConstraint

limits the (active) holdings for an individual asset or group of assets. This type of constraint can be used to set a lower bound and/or an upper bound on (active) asset holdings.

Depending on whether this constraint imposes a lower bound or an upper bound on the asset holdings, it has the following forms:

$$\sum_{i \in A} \lambda_i w_i \geq rhs,$$

or

$$\sum_{i \in A} \lambda_i w_i \leq rhs,$$

respectively, where A is the set of assets, λ_i and w_i represent the weight and holdings for asset i respectively, and rhs is the bound to be imposed.

If the `BENCHMARK` property is set, then the constraint is over the active holdings with respect to the benchmark, i.e., its left-hand side has the form

$$\sum_{i \in A} \lambda_i (w_i - b_i),$$

where b_i is the benchmark holding for asset i.

LimitLongHoldingConstraint

limits the long holdings for individual assets or a group of assets. This type of constraint can be used to set a lower bound or an upper bound on long asset holdings.

Depending on whether this constraint imposes a lower bound or an upper bound on the long asset holdings, it has the following forms:

$$\sum_{i \in A} \lambda_i w_i^+ \geq rhs,$$

or

$$\sum_{i \in A} \lambda_i w_i^+ \leq rhs,$$

respectively, where A is the set of assets, λ_i and w_i^+ represent the weight and long holdings for asset i respectively, and rhs is the bound to be imposed.

If the `BENCHMARK` property is set, then the constraint is over the long active holdings with respect to the benchmark, i.e., its left-hand side has the form

$$\sum_{i\in A} \lambda_i (w_i - b_i)^+,$$

where b_i is the benchmark holding for asset i.

LimitLongNamesConstraint

limits the number of long asset names held in a portfolio. This type of constraint can be used to set a lower bound or an upper bound on the number of long asset names held.

A `LimitLongNamesConstraint` bounds the number of long holdings among a set of assets, and depending on whether this constraint imposes a lower bound or an upper bound on the number of long asset names held, it has the following forms:

$$\left|\{i \in S | w_i^+ > 0\}\right| \geq k,$$

or

$$\left|\{i \in S | w_i^+ > 0\}\right| \leq k,$$

respectively, where S denotes the asset set and k denotes the bound.

LimitLongShortRatioConstraint

limits the amount of money held long relative to the amount held short for an individual asset or group of assets. This type of constraint can be used to set a lower bound or an upper bound on the ratio of long and short asset holdings.

The default form of a `LimitLongShortRatioConstraint` is:

$$\frac{\sum_{i\in A} \lambda_i w_i^+}{\sum_{i\in A} \lambda_i w_i^-} \leq rhs,$$

or equivalently

$$\sum_{i\in A} \lambda_i w_i^+ - rhs \sum_{i\in A} \lambda_i w_i^- \leq 0,$$

where A is the set of assets, λ_i, w_i^+ and w_i^- represent the weight, the long, and the short holdings for asset i respectively, and rhs is the bound to be imposed.

The weights λ are required to be non-negative.

LimitMinimumHoldingPeriodConstraint

can be used to prohibit trading tax lots which were recently purchased.

LimitModelDeviationConstraint

limits the deviation of the optimized holdings of a group of assets from the corresponding benchmark weights. This type of constraint can be used to place an upper bound on the distance between the optimized holdings and the benchmark weights.

The distance is computed as the 2-norm of the vector of differences, i.e. the constraint has the form:

$$\|\mathbf{x}\|_2 \leq ths,$$
$$x_i = \lambda_i (w_i - b_i), \quad i \in A,$$

where A is the set of assets in the asset group and $\mathbf{x}$ contains the deviation from the benchmark, multiplied by the weights of the assets given by the asset group.

LimitNamesConstraint

limits the number of asset names held (either long or short) in a portfolio. This type of constraint can be used to set a lower bound or an upper bound on the number of asset names held.

A `LimitNamesConstraint` bounds the number of holdings among a set of assets, and depending on whether this constraint imposes a lower bound or an upper bound on the number of asset names held, it has the following forms:

$$\|\{i \in S | w_i > 0\}\| \geq k,$$

or

$$\|\{i \in S | w_i > 0\}\| \leq k,$$

respectively, where S denotes the asset set and k denotes the bound.

LimitNetTaxGainsConstraint

limits the total tax liability due to long and short term gains resulting from portfolio transactions.

The general form of a `LimitNetTaxGainsConstraint` is:

$$\lambda_1 \text{ ltg} + \lambda_2 \text{ stg} \leq rhs,$$

where the λ_i are the coefficients of the tax parameters for long and short term gains, and rhs is the bound to be imposed.

LimitNetTaxLossesConstraint

limits the total tax liability due to long and short term losses resulting from portfolio transactions.

The general form of a `LimitNetTaxLossesConstraint` is:

$$\lambda_1 \text{ ltl} + \lambda_2 \text{ stl} \leq rhs,$$

where the λ_i are the coefficients of the tax parameters for long and short term losses, and rhs is the bound to be imposed.

LimitNumBuyTradesConstraint

limits the number of names traded among a set of assets by means of transactions representing purchases. This type of constraint can be used to set an upper bound on the number of names traded in order to control transactions costs.

A `LimiNumBuyTradesConstraint` has the following form:

$$\|\{i \in S | t_i > 0\}\| \leq k,$$

where S denotes the asset set, t_i the transaction, and k the bound to be imposed.

LimitNumSellTradesConstraint

limits the number of names traded among a set of assets by means of transactions representing sales. This type of constraint can be used to set an upper bound on the number of names traded in order to control transactions costs.

A `LimiNumSellTradesConstraint` has the following form:

$$\|\{i \in S | t_i < 0\}\| \leq k,$$

where S denotes the asset set, t_i the transaction, and k the bound to be imposed.

LimitNumTradesConstraint

limits the total number of names traded among a set of assets by means of transactions representing sales and purchases. This type of constraint can be used to set an upper bound on the number of names traded in order to control transactions costs.

A `LimiNumTradesConstraint` has the following form:

$$\|\{i \in S | t_i \neq 0\}\| \leq k,$$

where S denotes the asset set, t_i the transaction, and k the bound to be imposed.

LimitRelativeMarginalContributionToRiskConstraint

limits the relative marginal contribution to risk of a weighted group of assets. If a benchmark is specified, the relative marginal contribution to active risk is constrained instead.

Depending on whether a benchmark has been specified or not, the constraint is of the following forms:

$$\frac{(w-b)^T \Lambda Q (w-b)}{(w-b)^T Q (w-b)} \leq rhs,$$

or

$$\frac{w^T \Lambda Q w}{w^T Q w} \leq rhs,$$

respectively, where w represents the holdings, Q is the covariance matrix, b represents the benchmark, Λ is the diagonal matrix of the asset weights with the same dimension as the base set as the specified selection of assets (or base asset set), rhs the bound to be imposed. Note that w and Q are restricted to the base set of assets given by the `BASE_SET` property. The assets and their weights in Λ are given by the selection of the constraint.

LimitSellConstraint

limits the weighted sell transactions of a group of assets:

$$\sum_{i \in A} \lambda_i t_i^- \leq rhs,$$

where A is the set of assets, the λ_i are their weights, t_i^- are their sell transactions, and rhs is the bound to be imposed. The weights λ_i are required to be non-negative.

LimitShortHoldingConstraint

limits the short holdings for individual assets or a group of assets. This type of constraint can be used to set a lower bound or an upper bound on short asset holdings.

A `LimiShortHoldingConstraint` has the following form:

$$\sum_{i \in A} \lambda_i w_i^- \geq rhs,$$

if a lower bound is imposed, or

$$\sum_{i \in A} \lambda_i w_i^- \leq rhs,$$

if an upper bound is imposed, where A is the set of assets, λ_i their weights, w_i their short holdings, and rhs the bound to be imposed.

If the `BENCHMARK` property is set, then the constraint is over the short active holdings with respect to the benchmark, i.e., its left-hand side has the form

$$\sum_{i \in A} \lambda_i (w_i - b_i)^-,$$

where b_i is the benchmark holding for asset i.

LimitShortNamesConstraint

limits the number of short asset names held in a portfolio. This type of constraint can be used to set a lower bound or an upper bound on the number of short asset names held.

A `LimitShortNamesConstraint` bounds the number of short holdings among a set of assets, and depending on whether this constraint imposes a lower bound or an upper bound on the number of short asset names held, it has the following forms:

$$\|\{i \in S | w_i^- > 0\}\| \geq k,$$

or

$$\|\{i \in S | w_i^- > 0\}\| \leq k,$$

respectively, where S denotes the asset set and k denotes the bound.

LimitTaxLiabilityConstraint

limits the total tax liability resulting from a portfolio transactions. This type of constraint can be used to set an upper bound on the tax liability of a portfolio.

The general form is:

$$\lambda_1 \text{ tl} + \lambda_2 \text{ ltg} + \lambda_3 \text{ ltl} + \lambda_4 \text{ stg} + \lambda_5 \text{ stl} \leq rhs,$$

where the λ_i are the coefficients of the tax parameters, tl is tax liability, and rhs is the bound to be imposed.

LimitTotalRiskConstraint

limits the total risk of a weighted combination of assets. This type of constraint can be used to set an upper bound on either the absolute risk or the active risk of a group of assets. The absolute risk is defined as the standard deviation of the assets' return over the coming period. The active risk (sometimes called the tracking error) is defined as the standard deviation between the set of assets' return and the benchmark return over the coming period. The risk is computed via the asset-asset covariance matrix as given by the risk model of the assets. All parts of the covariance matrix related to assets not in the group of assets are ignored. Denoting the so computed risk as ρ, the `LimitTotalRiskConstraint` has the form:

$$\rho \leq rhs.$$

Expanding ρ we have

$$\sqrt{x^T Q_A x} \leq rhs$$

with

$$x_i = \lambda_i w_i, \quad i \in A,$$

if the absolute risk is being constrained, or

$$x_i = \lambda_i (w_i - b_i), \quad i \in A,$$

if the active risk is being constrained, where A is the set of assets in the group, Q_A is the covariance matrix restricted to assets in A, x contains the optimized asset holdings, b represents the benchmark holdings, and λ is the asset weights specified in the group of assets.

If the risk model is a factor risk model then weights of its factor and specific risk parts can be changed by specifying the optional `FACTOR_WEIGHT` and `SPECIFIC_WEIGHT` properties. In that case, the risk is computed as

$$\sqrt{\nu_f x^T B_A \Omega B_A^T x + \nu_s x^T \Delta_A^2 x}$$

where ν_f is the factor weight, ν_s is the specific weight, B is the exposure matrix, Ω is the factor covariance matrix, $Delta$ is a diagnoal matrix of the specific risks. The values of those two parameters are ignored if the risk model is not a factor model.

LimitTradeConstraint

limits the size of the weighted transactions (either sales or purchases) of an asset or a group of assets. This type of constraint can be used to impose an upper bound or a lower bound on the transactions of an asset or group of assets.

Depending on whether it imposes an upper bound or a lower bound, the constraint takes the following forms:

$$\sum_{i \in A} \lambda_i t_i \leq rhs$$

or

$$\sum_{i \in A} \lambda_i t_i \geq rhs$$

respectively, where A is the set of assets, λ_i their weights, t_i their transactions, and rhs the bound to be imposed.

LimitTurnoverConstraint

limits the turnover (total amount bought or sold) for an individual asset or group of assets. This type of constraint can be used to set an upper bound on an asset's or group of assets' turnover.

$$\sum_{i \in A} \lambda_i |t_i| \leq rhs,$$

where A is the set of assets, λ_i their weights, t_i are their transactions, and rhs is the bound to be imposed. The weights λ_i are required to be non-negative.

LimitWeightedAvgConstraint

imposes a bound on the ratio between the weighted sum of holdings and the absolute value of the sum of holdings of a set of assets. Given the base set of assets A and their weights λ_i, let $\psi(x)$ denote the weighted average of the holdings x, i.e.,

$$\psi(x) := \frac{\sum_{i \in A} \lambda_i x_i}{|\sum_{i \in A} x_i|},$$

the constraint imposes a bound rhs on $\psi(x)$. rhs will be used as an upper bound, lower bound, or both depending on the value of the `MIN` and `MAX` properties. There are three variants of the constraint:

1. **absolute**: this is the variant of the constraint if no benchmark is given and has the form

$$\psi(w) \leq rhs \quad \text{when } \texttt{MAX} \text{ is } rhs.$$

2. **absolute with benchmark**: this is the variant of the constraint if the `BENCHMARK` property is set and the `UNIT` is `Unit.NUMBER`. It has the form

$$\psi(w) - \psi(b) \leq rhs \quad \text{with benchmark } b \text{ and } \texttt{MAX} \text{ set to } rhs.$$

This variant is not supported if the sum of the benchmark holdings over the base set is zero.

3. **relative to benchmark**: this is the variant of the constraint if the `BENCHMARK` property is set and the `UNIT` is `Unit.PERCENT`. It has the form

$$\frac{\psi(w)}{\psi(b)} - 1 \leq rhs \quad \text{with benchmark } b \text{ and } \texttt{MAX} \text{ set to } rhs.$$

> This variant is not supported if the sum of the benchmark holdings over the base set is zero or the weighted average of the benchmark is zero.

The base set is specified by the required `BASE_SET` property. The weights λ are the asset weights in the selection.

ProbabilisticConstraint

imposes a probabilistic linear inequality for a weighted combination of asset holdings, where the weights are not known with certainty but that are assumed to have a multivariate normal distribution. For a given confidence level η, a `ProbabilisticConstraint` ensures that the given inequality holds with a probability of at least η. The base set of assets to be used in the constraint is specified through a `selection` set. The mean of the weights is given as a Group or Metagroup. The confidence level, η, must be $0.5 \leq \eta < 1.0$.

The constraint is of the form

$$\sum_{i \in A} \alpha_i w_i - \Phi^{-1}(\eta) \| Q^{1/2} w \| \geq rhs,$$

which is equivalent to

$$\text{Prob} \left(\sum_{i \in A} \alpha_i w_i \geq rhs \right) \geq \eta,$$

where A is the set of assets in the asset group, α_i are the (estimate?) means of the weights, Q is the covariance matrix, and Φ is the cumulative distribution function of the standard normal distribution. The sign in front of the norm term depends on the sense of the inequality (either a maximum or a minimum).

RobustConstraint

imposes a linear inequality for a weighted combination of asset holdings, where the weights are not known with certainty. For a given uncertainty parameter κ, a `RobustConstraint` ensures that the given inequality holds for all weights in an elliptical uncertainty region whose size is determined by κ. The base set of assets to be used in the constraint is specified through the `BASE_SET` property. The estimate of the mean of the weights is given as a constraint selection. The uncertainty parameter, κ, must $\kappa \geq 0.0$.

A `RobustConstraint` is of the form

$$\sum_{i \in A} \alpha_i w_i - \kappa \| Q^{1/2} (w - z) \| \geq rhs,$$

where A is the base set of assets, α_i are the means of the weights, w_i are the holding variables, Q is the covariance matrix of the estimates and z is the model portfolio. The sign in front of the norm term depends on the sense of the inequality.

ThresholdHoldingConstraint

represents a constraint with the restriction that if an asset is held, the position must be at least some threshold (minimum) amount (either long or short). This type of constraint can be used to set an absolute lower bound L on individual positions, with the caveat that a zero value for holdings is allowed. The constraint ensures that the optimized holdings are either 0 (no holdings), at least L (i.e., for long holding) or at most $-L$ (i.e., for short holdings).

The constraint takes the following form:

$$w = 0 \vee |w| \geq L,$$

where w represents the asset holdings.

ThresholdTradeConstraint

represents a constraint with the restriction that if there is a transaction or trade for an asset or group of assets, then the transaction is for at least a minimum threshold amount (either a buy or a sell). This type of constraints can be used to specify a minimum transaction size.

Note that a zero transaction is permissible. In addition, it is always permissible to trade an amount that will reduce the holdings to zero even if that trade amount does not reach the minimum threshold level specified by the threshold trade constraint.

The constraint takes the following form:

$$t = 0 \vee |t| \geq L \vee w = 0,$$

where w represents the asset holdings, t is the transaction, and L is the lower bound to be imposed.

4 Set Up Objectives

In Axioma**Portfolio**™, the objective function is expressed as a linear combination of objective terms or goals. For example, the rebalancing objective might be to maximize net return (expected return minus transaction costs). In this case, we would need a term representing the expected return and another term representing the transaction costs. In MATLAB, we use class `AXObjective` to represent an objective consisting of different objective terms. In the following sections, we first introduce the objective functions and then we describe each type of objective terms.

4.1 Create Objective Object

To create a `AXObjective`, just add a line to your code like this:

```
obj = AXObjective('DefaultObj', 'MAXIMIZE'); % 1st param is id, 2nd is objective sense (or type)
```

The constructor of `AXObjective` need 2 parameters, the first is used as id for later reference, the second served as type representing objective sense, which can only be one of

```
'MAXMIZE'   'MINIMIZE'
```

Once the object created, you need to add some objective terms to it. There are many different types of objective terms, each requiring different parameters. `AXObjective` provides a group of function to add objective terms.

```
o = o.addTransactionCostObjective(objectiveTermId, weight, priority);
o = o.addExpectedReturnObjective (objectiveTermId, alphaGroupId, weight, priority);
o = o.addLinearShortObjective    (objectiveTermId, shortCostGroupId, weight, priority);
o = o.addLinearShortSellObjective(objectiveTermId, shortSellCostGroupId, weight, priority);
o = o.addRobust(objectiveTermId, alphaGroupId, kappa, riskModelId, weight, priority);
o = o.addVarianceObjective(objectiveTermId, benchmarkGroupId, riskModelId, weight, priority);
o = o.addRiskObjective(objectiveTermId, benchmarkGroupId, riskModelId, weight, priority);
o = o.addRiskAlphaFactorObjective(objectiveTermId, benchmarkGroupId, riskModelId, alphaFactorVol...
                                , weight, priority);
o = o.addVarianceAlphaFactorObjective(objectiveTermId, benchmarkGroupId, riskModelId, alphaFactorVol...
                                    , weight, priority);
o = o.addMarketImpactObjective(objectiveTermId, buyImpactGroupId, sellImpactGroupId, marketImpactTypeStr...
                             , weight, priority);
```

As you can see, every function have 3 common parameters: the first is `objectiveTermId`, the second to last is `weight`, and the last is `priority`. The reason we put `weight` and `priority` to the last two positions is that they can take default values: **if not provided,** `weight` **takes 1 and** `priority` **takes 0**. [1] Other parameters depend on specific objective terms which we will explain in the next subsection.

4.2 Objective Terms

ExpectedReturnObjective

represents the expected return of a set of assets. The set of involved assets and their associated expected returns are specified by parameter `alphaGroupId`. This objective term contribution to a rebalancing objective is expressed as

$$\nu \sum_{i \in S} \alpha_i w_i,$$

where

- ν is the weight of this objective term in the objective,
- S is the set of assets in the alpha Group,
- α_i is the expected return of asset i as specified in the alpha Group, and
- w_i is the holding of asset i in the portfolio.

LinearShortObjective

represents linear short tilts for a set of assets.

A (short) tilt is a weighting of the (short) holding in an asset.

The set of involved assets and their associated short position tilts are specified by parameter `shortCostGroupId` (can be either asset group or set). If only a set (and no group) of assets is assigned, each asset's linear short tilt is assumed to be 1.

The objective term contribution to a rebalancing objective is expressed as

$$\nu \sum_{i \in A} \lambda_i w_i^-,$$

where

- ν is the weight of this objective term in the objective,
- A is the set of assigned assets,
- λ_i is the short tilt for asset i as specified in the assigned asset group (or 1 if only an asset set is assigned), and
- w_i^- is the short holding amount in asset i (being 0 for non-held or long-held assets).

LinearShortSellObjective

represents linear short sell costs for a set of assets.

A short sell is the part of a sell trade which does not consists of selling initially existing (long held) shares, i.e., the sell amount net of an existing long position in the asset (if any).

The set of involved assets and their associated linear short sell costs are specified by parameter `shortSellCostGroupId` (can be either asset group or set). If only a set (and no group) of assets is assigned, each asset's linear short sell cost is assumed to be 1.

[1] Remember that you can also set priority for `AXConstraint` object by `constraint_obj.priority = val`.

This objective term contribution to a rebalancing objective is expressed as

$$\nu \sum_{i \in A} \lambda_i \max(t_i^- - h_i^+, 0),$$

where

- ν is the weight of this objective term in the objective,
- A is the set of assigned assets,
- λ_i is the linear cost for selling asset i as specified in the assigned asset group (or 1 if only an asset set is assigned),
- t_i^- is the sell volume of asset i (being 0 for non-traded or bought assets), and
- h_i^+ is the long initial holding volume in asset i (being 0 for non-held or short-held assets).

The term $\max(t_i^- - h_i^+, 0)$ represent the short sell amount in asset i.

MarketImpactObjective

represents a market impact function for a set of assets.

The intention of a market impact function is to penalize large trades. The idea is that a large trade is difficult to process or implement on the trading side and should be penalized in the portfolio optimization to discourage it from appearing in the solution. This is sometimes called *market impact* or *illiquidity*. It is modeled using different penalty cost functions in the transaction size. The typical functions used are quadratic, three halves power, or five thirds power. Their increasing shape serves to penalize the transaction more heavily as the transaction size increases. For achieving the penalizing effect, the term weight should have negative sign in a maximization objective, and a positive sign in a minimization objective.

The set of assets with associated buy impact coefficients is specified by parameter `buyImpactGroupId`, and the set of assets with associated sell impact coefficients is specified by parameter `sellImpactGroupId`. The same asset can be contained in both groups. Further, the impact function shape is defined by parameter `marketImpactTypeStr` which can take one of

```
{'FIVE_THIRDS_POWER', 'QUADRATIC', 'THREE_HALVES_POWER'}
```

This objective term contribution to a rebalancing objective is expressed as

$$\nu \left(\sum_{i \in B} \lambda_i^B (t_i^+)^p + \sum_{i \in S} \lambda_i^S (t_i^-)^p \right)$$

where

- ν is the weight of this objective term in the objective,
- B is the set of assets in the buy impact Group,
- λ_i^B is the impact coefficient for buying asset i as specified in the assigned buy impact Group,
- t_i^+ is the buy volume of asset i (being 0 for non-traded or sold assets),
- p is the power of the impact function as defined by `marketImpactTypeStr` (currently supported are the powers 2, 3/2, and 5/3),
- S is the set of assets in the sell impact Group,
- λ_i^S is the impact coefficient for selling asset i as specified in the assigned sell impact Group, and
- t_i^- is the sell volume of asset i (being 0 for non-traded or bought assets).

RiskObjective

represents the risk (in terms of standard deviations) of a set of assets.

In Axioma**Portfolio**™, portfolio risk can be measured as absolute or active risk (compared to a benchmark) and in terms of variance or standard deviation - the latter is usually referred to as 'risk' throughout Axioma**Portfolio**™Portfolio. Consequently, the `RiskObjective` also refers to the standard deviation measure of portfolio risk. For using the variance measure, please see the `VarianceObjective`.

A risk measure is applied to a (weighted) set of assets (can either be a asset group or set). [2] If an asset Group is assigned, the group specifies the weights for the assets to be involved. If only an asset set is assigned, the weight for each asset in the set is assumed to be 1. Optionally, a benchmark Group (`benchmarkGroupId` can be set to make the risk measure active, otherwise it is absolute. Eventually, the RiskModel to apply has to be set by parameter `riskModelId`. The risk model can either be a *DenseRiskModel* defined by a full asset-asset covariance matrix, or a *FactorRiskModel* based on a set of factors and defined by the asset-factor exposures, the factor-factor covariance matrix, and asset-wise specific risk. If a factor model is used, the risk measure can further be adjusted by varying weights for the factor risk and the specific risk portions.

If the assigned group or set of assets contains composite assets, note that the risk measure always applies to the underlying assets, i.e., the composite assets are replaced by the appropriately scaled amounts of underlyings as defined by the composite asset's compositions.

This objective term contribution to a rebalancing objective is expressed as

$$\nu \cdot \sqrt{\mathbf{x}^T C_A^T Q_A C_A \mathbf{x}} \quad \text{with} \quad \mathbf{x} = (x_i)_{i\in A} \quad \text{and} \quad x_i = \lambda_i(w_i - b_i) \quad \forall i \in A,$$

- ν is the weight of this objective term in the objective,
- A is the set of involved assets, being the intersection between the set of assigned assets (in the group or set) and the set of assets covered by the assigned RiskModel,
- C_A is the composite asset exposure (or composition) matrix restricted to the set A,
- Q_A is the risk model covariance matrix restricted to the set A of involved assets, where
  - for a DenseRiskModel, Q_A is directly given by the risk model (as part of the dense covariance matrix),
  - for a FactorRiskModel, Q_A is defined as

    $$Q_A = \nu_f B_A \Omega B_A^T + \nu_s \Delta_A^2,$$

    where
    * ν_f is the factor risk weight,
    * B_A is the asset-factor exposure matrix, restricted to the set A,
    * Ω is the factor-factor covariance matrix,
    * ν_s is the specific risk weight,
    * Δ_A is a diagonal matrix with the asset specific risks (restricted to the set A in appropriate order) in the diagonal elements (and 0 else),
- λ_i is the weight of asset i as specified in the assigned asset group (or 1 if only an asset set is assigned),

[2] Currently our MATLAB API assumes this group is the whole portfolio. Improvements will be made in the next version.

- w_i is the holding of asset i in the portfolio.
- b_i is the holding of asset i in the benchmark if one is set, otherwise 0.

Note that the expressed risk is only active if a benchmark is assigned, and otherwise absolute.

Robust

represents the robust expected return of a set of assets.

AxiomaPortfolio™ allows for robust optimization, a deterministic framework to account for data uncertainty information directly within the optimization methodology. The idea is to substitute the particular expectation expressed in the expected returns by a trust region around this mean expectation and to take all possible return realizations within this trust region into account.

For a given set of asset holdings, the value of a Robust term is computed as the worst-case over all possible expected returns in a uncertainty region around the mean. The base set of assets to be used in the objective is specified through an `AssetSet`. The mean of the expected returns is given as the alpha Group (parameter `alphaGroupId`), the covariance is defined through an `AlphaUncertaintyModel` (which can be seen as a risk model for the estimation errors), and the uncertainty region is defined through the uncertainty parameter $\kappa > 0$. The size of the confidence region grows with larger values of κ, whereas its (elliptic) shape is defined by the uncertainty model.

For further information on robust portfolio optimization, please see Chapter 13 of the AxiomaPortfolio™ GUI User Guide, or the article *Incorporating Estimation Errors into Portfolio Selection: Robust Portfolio Construction* by S. Ceria and R. Stubbs, Journal of Asset Management 7, 2006, pp. 109-127.

This objective term contribution to a rebalancing objective is expressed as

$$\nu \left[\sum_{i \in A} \alpha_i w_i \pm \kappa \left\| Q^{1/2}(w - z) \right\| \right],$$

where

- ν is the weight of this objective term in the objective,
- A is the set of assets in the base `AssetSet`,
- α_i is the expected return mean of asset i as specified in the alpha Group,
- w_i is the holding of asset i in the portfolio, and $\mathbf{w}$ is the vector of all these holdings,
- κ is the uncertainty level ,
- Q is the covariance matrix from the assigned `AlphaUncertaintyModel`,
- z is the model portfolio .

The sign in front of the norm term depends on the sense of the optimization ('$-$' for a maximization and '$+$' for a minimization target, to ensure a detracting correction corresponding to the worst-case approach).

TransactionCostObjective

represents transaction costs for a set of assets as described by a piecewise linear transaction cost model.

The `TransactionCostModel` set in the Rebalancing (`AXRebalancing`) is to be used.

This objective term contribution to a rebalancing objective is expressed as

$$\nu \sum_{i \in S} \mathrm{tc}_i(t_i),$$

where

- ν is the weight of this objective term in the objective,
- S is the set of assets in the assigned base `AssetSet`,
- tc_i is the piecewise linear transaction cost function for asset i, as specified by the `TransactionCostModel` in rebalancing object (see next section), and
- t_i is the trade volume in asset i.

VarianceObjective

represents the risk (in terms of variance) of a set of assets.

In Axioma**Portfolio**™, portfolio risk can be measured as absolute or active risk (compared to a benchmark) and in terms of variance or standard deviation - the latter is usually referred to as *risk* throughout Axioma**Portfolio**™. Consequently, the `RiskObjective` also refers to the standard deviation measure of portfolio risk. The `VarianceObjective` described here measures the risk in terms of variance.

A risk measure is applied to a (weighted) set of assets defined by the assigned asset group or set. If an asset Group is assigned, the group specifies the weights for the assets to be involved. If only an AssetSet is assigned, the weight for each asset in the set is assumed to be 1. Optionally, a benchmark Group can be set to make the risk measure active, otherwise it is absolute. Eventually, the `RiskModel` to apply has to be set. The risk model can either be a `DenseRiskModel` defined by a full asset-asset covariance matrix, or a `FactorRiskModel` based on a set of factors and defined by the asset-factor exposures, the factor-factor covariance matrix, and asset-wise specific risk. If a factor model is used, the risk measure can further be adjusted by varying weights for the factor risk and the specific risk portions.

If the assigned group or set of assets contains composite assets, note that the risk measure always applies to the underlying assets, i.e., all composite assets are replaced by the appropriately scaled amounts of underlyings as defined by the composite asset's compositions.

This objective term contribution to a rebalancing objective is expressed as

$$\nu \cdot \sqrt{\mathbf{x}^T C_A^T Q_A C_A \mathbf{x}} \quad \text{with} \quad \mathbf{x} = (x_i)_{i \in A} \quad \text{and} \quad x_i = \lambda_i (w_i - b_i) \quad \forall i \in A,$$

- ν is the weight of this objective term in the objective,
- A is the set of involved assets, being the intersection between the set of assigned assets (in the group or set) and the set of assets covered by the assigned RiskModel,
- C_A is the composite asset exposure (or composition) matrix restricted to the set A,
- Q_A is the risk model covariance matrix restricted to the set A of involved assets, where
  - for a DenseRiskModel, Q_A is directly given by the risk model (as part of the dense covariance matrix),
  - for a FactorRiskModel, Q_A is defined as

    $$Q_A = \nu_f B_A \Omega B_A^T + \nu_s \Delta_A^2,$$

    where

    * ν_f is the factor risk weight,
    * B_A is the asset-factor exposure matrix, restricted to the set A,
    * Ω is the factor-factor covariance matrix,
    * ν_s is the specific risk weight,

  * Δ_A is a diagonal matrix with the asset specific risks (restricted to the set A in appropriate order) in the diagonal elements (and 0 else),

- λ_i is the weight of asset i as specified in the assigned asset group (or 1 if only an asset set is assigned),
- w_i is the holding of asset i in the portfolio.
- b_i is the holding of asset i in the benchmark if one is set, otherwise 0.

Note that the expressed variance is only active if a benchmark is assigned, and otherwise absolute.

5 Optimization

5.1 Steps

In MATLAB, optimization is represented by `AXOptimization` class. Before you can set up a `AXOptimization` object, you need to

1. have a dataset (`AXDataSet`) object created or loaded,
2. have a objective object (`AXObjective`) which already is added into all the objective terms needed for the optimization,
3. have a bunch of constraints (`AXConstraint` objects) collected in a cell vector.

These are content of previous sections. Now we are almost in a position to create an optimization object.

4. Create a strategy object (`AXStrategy`) like this:

```
strategy = AXStrategy('STRATEGY' ...                  % strategy ID
                    , 'objective',   objective ...    % AXObjective object you have created before
                    , 'constraints', constraints);    % a cell vector of AXConstraint objects
```

5. Create a rebalancing object (`AXRebalancing`) like this:

```
rebal = AXRebalancing('REBAL' ...                          % rebalancing ID
                    , 'priceId',     'PRICE' ...           % price ID in dataset
                    , 'accountId',   'ACCT_FAC_SP500' ...  % account ID in dataset
                    , 'alphaId',     'ALPHA_FAC_SP500' ... % alpha ID in dataset
                    , 'riskmodelId', 'RISK_BARRA' ...      % risk model ID in dataset
                    , 'tcmodelId',   'TCM_QSG' ...         % transaction mode ID in dataset
                    , 'benchmarkId', 'X00053');            % benchmark ID in dataset
```

6. Create and run the optimization object (`AXOptimization`):

```
opt = AXOptimization(ds, strategy, rebal); % all 3 parameter are created in previous steps
results = opt.run(opt.dates, 'TEST'); % dates can be part of opt.dates if you want not run full periods
stats = opt.report(results);          % generate report, stats has analytical statistics for reference
```

Our major work is done here, though, still, some more details you need to know.

5.2 More Parameters to Strategy and Rebalancing

You can provide more information to `AXStrategy` and `AXRebalancing` object either in their constructors or after they are being created.

For `AXStrategy`, the following parameters are provided:

```
isObjectiveHierarchy     % default false
isConstraintHierarchy    % default false
isAllowCrossover         % default false
isAllowGrandfathering    % default false
isAllowShorting          % default false
isIgnoreCompliance       % default false
isIgnoreRoundLots        % default false
included                 % default 'MASTER'; 'MASTER' created by axioma automatically, see below
excluded                 % default empty {};
isMarketNeutral          % default false; not belongs to axioma; from Jeremy code
objective                % MUST set; default empty; an AXObjective object
constraints              % MUST set; default empty; vector of AXConstraint objects
```

Parameters started with `'is'` is of boolean type. The set these parameters in constructor, use this syntax:

```
strategy = AXStrategy('STRATEGY' ...                % strategy ID
                   , parameter_name1, parameter_value1, ...
                   , parameter_name2, parameter_value2, ...);
```

To get/set/modify these parameters after they are created, use this syntax:

```
strategy.parameter_name = parameter_value;
```

Suppose, for instance, if you want to add a constraint to an existing `AXStrategy` object, you can do this:

```
strategy.constraints = [strategy.constraints {new_constraint}];
```

For `AXRebalancing`, similar manners are used. Here's the list of all parameters:

```
benchmarkId     % default 'BENCHMARK'; see below
roundlotsId     % default 'ROUNDLOTSIZE' which automatically added in dataset; see section 2
accountId       % default ''
alphaId         % default ''
prevalphaId     % default ''
betaId          % default ''
priceId         % default ''
riskmodelId     % default ''
tcmodelId       % default ''
cashflow        % default 0
minruntime      % default 1
maxruntime      % default 60
budgetsize      % default NaN, meaning use portfolio size as budgetsize
volumeId        % default '';
```

Parameters ended with `'Id'` should refer to an element either in dataset, or in the following which are created automatically by Axioma**Portfolio**™:

`Accounts`

`'ACCOUNT'`, `'ACCOUNT.LONG'`, `'ACCOUNT.SHORT'`

`Alpha`

`'REBALANCING.ALPHA_GROUP'`

`Benchmark`

`'REBALANCING.BENCHMARK'`

```
Simple Groups
      'REBALANCING.PRICE_GROUP', 'REBALANCING.ROUND_LOT_GROUP', 'PREEMPTIVE_SELLOUT'
Asset Sets
      'MASTER', 'LOCAL_UNIVERSE', 'COMPOSITE', 'NON-CASH ASSETS'
Risk Model
      'REBALANCING.RISK_MODEL'
Transaction Cost Model
      'REBALANCING.COST_MODEL'
```

5.3 Analytical Results and Report

Usually you run `report` method on optimization object with results returned from `run` to obtain an analytical report. It's *rare* you need to check the results in detail yourself. But there are always guys like to delve into details. This subsection gives further information about results from an optimization run.

Results returned from optimization `run` are organized as a `constainers.Map` which contains *key-value pairs* where key is a date and value is a structure like this:

```
          solts: [1x1 struct]
         pfinit: [1x506x5 myfints]
        pffinal: [1x506x10 myfints]
            trd: [1x443x7 xts]
        summary: [1x37x3 myfints]
           vios: [1x1 struct]
     frontierid: NaN     % just ignore if not database-driven
    rebalanceid: NaN     % just ignore if not database-driven
```

As shown above, each `xts` object in the structure is of one-period , to combine them over whole periods across the `Map`, just do this:

```
v = values(results);   % get all values of the Map in order of key (date). v is a cell of all values
v = cell2mat(v);       % convert it to a matrix of structure

summary = cat(1,v.summary);  % now summary is a T-period xts (42x37x3 for above example)
```

You can do the same for other `xts` things. For example, for `pffinal`,

```
pffinal = cat(1, v.pffinal); % equivelent to [v(1).pffinal; v(2).pffinal; ...; v(end).pffinal]
```

If it's possible every period involves different stocks (that one of reason we want to keep each period result separate), then, first align them, then combine:

```
pffinal = {v.pffinal}; % pffinal now is cell vector of one-period xts
[pffinal{:}] = alignfields(pffinal{:}, 'union', 1);
pffinal = cat(1, pffinal{:});
```

`vios` itself is a structure of like

```
    ACTIVENOALPHABET: [1x1x6 xts]
     ACTIVESECTORBET: [1x10x6 xts]
    ACTIVECOUNTRYBET: [1x1x6 xts]
          ACTIVEBETA: [1x1x6 xts]
      TRACKINGERROR: [1x1x6 xts]
    THRESHOLDHOLDING: [1x501x6 xts]
      THRESHOLDTRADE: [1x501x6 xts]
    ABSOLUTESTOCKBET: [1x501x6 xts]
      ABSOLUTESHARES: [1x500x6 xts]
         LONGHOLDONG: [1x1x6 xts]
```

```
            SHORTHOLDONG: [1x1x6 xts]
             CASHHOLDONG: [1x1x6 xts]
                LONGBETS: [1x250x6 xts]
               SHORTBETS: [1x250x6 xts]
                TURNOVER: [1x1x6 xts]
```

each field is an `xts`, so you can combine them in a similar way. For example for `TURNOVER`,

```
vios = {v.vios}; % since structure for each period may be different, we can only use cell here
turnover = cellfun(@(x){x.TURNOVER}, vios); % turnover now is cell vector of one-period xts
turnover = cat(1, turnover{:});
```

6 Tips

1. Get all secids in universe
2. Get all groups in DataSet
3. Modify workspace dynamically