A Guide to Factor ToolBox

April 28, 2014

Contents

1 Introduction **1**

2 Build A New Factor **2**

2.1 Set Path 3
2.2 Create Class File 3
Understand Frequency 4
2.3 Write Implementation Code 5
2.4 Test 7
2.5 Register Factor 8
2.6 Populate 8
2.7 More Examples 8
5-Day Price Reversal (REVSAL5D) 8
Blended EPS Yield (BEPSY) 10

3 Database and Related Functions **11**

3.1 Structure Overview 11
3.2 Data Cache 12
How we process PIT data 13
3.3 Factor Database 14
3.4 Loading Data to Matlab 14
LoadIndexHoldingTS 14
LoadSecInfo 15
LoadRawItemTS 15
LoadRawItemPIT 16
LoadIndexItemTS 17
LoadFactorInfo 18
LoadFactorTS 18
LoadFXTS 19
LoadQSAggTS 20
LoadQSSecTS 20

3.5 Saving Data to Database . . . 21
Save2DB . . . 21
Factory.Register2DB . . . 22
3.6 Data Process Functions . . . 22
CashFlowDecomp . . . 22
CashFlowDecompPIT . . . 23
genDateSeries . . . 23
LatestDataDate . . . 24
Live2PIT . . . 25

4 Use of myfints 25
4.1 Aligning Data . . . 26
aligndata . . . 26
aligndates . . . 27
alignfields . . . 27
isaligneddata, isalignedfields, isaligneddates . . . 28
4.2 Filling Missing Data . . . 28
backfill . . . 28
forwardfill . . . 29
4.3 Factor Backtesting . . . 29
optimize . . . 29
normalize . . . 33
neutralize . . . 34
csregress . . . 34
csqtrtn . . . 36
csqtspread . . . 37
tsplot . . . 38
mat2fts . . . 42
xls2fts . . . 43
4.4 Comparison Operators: >=, >, ==, ∼=, <=, < . . . 44
4.5 Ordinary Matrix Algebra Operators: `+, -, *, /, .*, ./` . . . 44
4.6 Indexing . . . 44
4.7 Cross-Sectional Statistics . . . 46
cscorr, csrankcorr, cscorrwt . . . 46
cscov . . . 47
csmax, csmean, csmedian, csstd, cssum, csnorm . . . 47
ftsmovavg, ftsmovstd, ftsmovsum . . . 48
4.8 Third-Dimensional Statistics . . . 48
ftsnanmean, ftsnanstd, ftsnansum . . . 48
4.9 Performing Specified Functions on `myfints` . . . 49
uniftsfun . . . 49

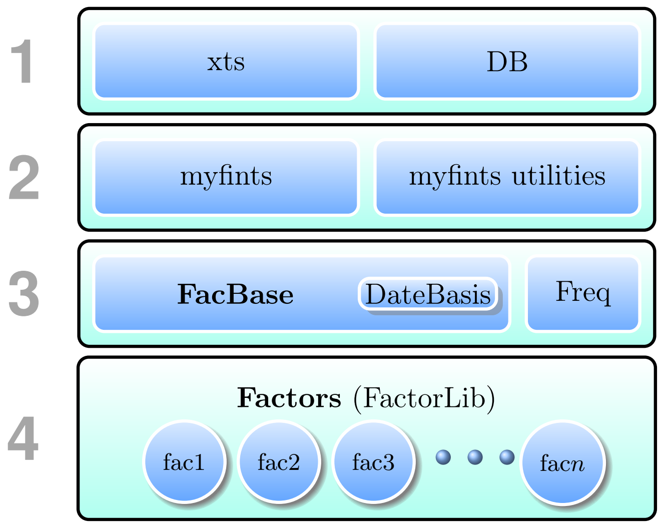
biftsfun 50
multiftsfun 51
ftsmovfun 52
bsxfun 53
4.10 Field Operations 55
4.11 Resampling 55
convertto, toannual, tosemi, toquarterly, tomonthly, toweekly, todaily 55
4.12 List of Other Functions 57

5 Mutliple Dimensional Time Series: xts 58
5.1 Common Functions with slightly different usage 58
5.2 Common Functions 58
5.3 Functions only applicable for myfints 63
5.4 Indexing 63
5.5 Packing Matrix into `xts` 64
mat2xts 64

1 Introduction

The Factor ToolBox defines a framework for constructing factors and provides a bunch of functions to support and facilitate the process, including acquiring, sampling, aligning and manipulating basic data items (financial like EPS and DPS and market-related like prices and shares) upon which factors are built, and testing, visualizing and reporting tools. These functions cover a wide range of functionalities and reach every part of our quantitative infrastructure, across from the very first stage of quantitative model development work to the last stage of generating the real trading data. Most of our other tools, such as factor (linear and nonlinear) combination, factor analysis and monitoring, performance attribution, model backtest and portfolio construction, are built on this toolbox. Figure 1 shows you the whole picture of it.

Figure 1: Structure of Factor ToolBox



1. The very first layer is quite fundamental, consisting of `xts` – multi-dimensional time series class representing the basic data form used in MATLAB, and `DB` – interface class to Database for accessing data. `xts` can be regarded as labeled matrices with first dimension for time, others labeled with some meaningful names. Labeled matrices greatly facilitates data sampling and aligning which are tedious chores and easily error-prone.

   `DB` isolates and wraps most of interactions with database. There are another group of functions (name started with `'Load'`) in `myfintsutility` which are actually another wrapping layer for `DB` class.

2. This layer is `myfints` and related things in `myfintsutility` folder. `myfints` is 2-D `xts`, derived from `xts` (therefore having all methods from `xts`), plus things specific and more meaningful (e.g., `tsplot`) (e.g., functions with names started with `'cs'`) in 2-D situation. Many useful tools are also placed here, like

   - `Options`, collecting operations related to processing named function arguments (name-value pairs),

- `grep`, regular search and/or replace string in files,
- `PDFDoc`, generating PDF documents with figures and tables,
- `TRACE`, logging facilities (to screen, files, and/or database),

to name a few.

3. `FacBase` is derived from `myfints` and provides a framework on how to define a factor. Factors, associated with a security, is something that related to the future movements of the security prices; in other words, they have certain predictability. A typical factor, for example, is Book to Price, a ratio of company's book value to its share price, usually positively correlated to the company's stock price. `FacBase` take care of the details of such things as: data processing (e.g., back filing missing data), sampling (frequency internally used by items for calculation and frequency in factors returned to user), and aligning (both in time and other dimensions). It also provides methods for period calculation.

   Inside `FacBase`, there is a member of type `DateBasis` to deal with different (internal) frequencies, with the help from class `Freq`.

4. This layer consists of classes of factors derived from `FacBase`. Since the base class `FacBase` deals with majority routine common things, concrete factor classes should only focus on the definition of factors. That is why code in factors usually is so neat.

   There also exists a class `Factory` in charge of populating factors to database.

In this guide book, we will first (section 2) present you the factor construction framework, detailing the steps how to create a factor. Then move to discuss functions consisting of this toolbox by category. Section 3 briefs the database structure and related functions. Since in most cases, we encounter and deal with 2-D matrices, so section 4 discusses `myfints`, which is a 2-D version of `xts`, with many unique things and being most useful in quantitative research. After that, `xts` will be introduced with comparison with `myfints`.

2 Build A New Factor

In this section, we take *Book to Price Ratio* as an example to show how to construct a new factor and add it to our (existing) factor library.

Book to price ratio is a popular multiple to value stocks. It is defined as

$$\text{Book to Price} = \frac{\text{Total Equity Reported}}{\text{Price} \times \text{Shares Outstanding}} \tag{1}$$

A higher B/P ratio could mean that the stock is undervalued, or, the higher the values of the factor, the better the associated stock.

Now follow the steps discussed below (subsection (2.1)-(2.6) to create this factor.

2.1 Set Path

Set the path so that MATLAB can find our toolbox:

```
basepath = ['Y:\' getenv('USERNAME') '\QuantStrategy\Analytics\'];
addpath([basepath 'Utility']);            % for xts, myfints
addpath([basepath 'date']);               % for DateBasis and Freq
addpath([basepath 'DB']);                 % DataBase related stuff
addpath([basepath 'myfintsUtility']);     % for bunch of things
addpath([basepath 'FactorLib\updated']);  % all up-to-date factors code
```

These pathes is not only for creating factors, but also for developing quantitative models.

2.2 Create Class File

Every factor is defined as a class and organized in a class file under the folder `$/QuantStrategy/Analytics/FactorLib/Dev`) where $ usually is you home folder. The class file should have the same name as the class name (in current case, it is `BKTOPRICE`) and be structured as

```
classdef BKTOPRICE < FacBase
    methods (Access = protected)
        function factorTS = build(o, secIds, startDate, endDate)
        ...
        end

        function factorTS = buildLive(o, secIds, runDate)
        % ONLY needed if live version largely different from nonlive (build) version.
        ...
        end
    end
end
```

Every factor class must include method `build`, which takes four parameters: [1]

- **o** (lowercase letter) the object of the class. [2] Bearing in mind when program running here, aside members derived from `myfints`, the following members already be set and can be accessed:
  - `DateBasis` keeps the frequency used internally of data items and details related to frequency conversion. See the subsection *Understand Frequency* for detailed explanation. *You can change this member if necessary, but should do this before loading any data items.*
  - `isLive` is a *read only* member indicating if we are running in live or back-test mode.

[1] These parameters will be passed by `create` method in `FacBase` which unified for all factor classes.

[2] In most cases and as sort of convention, we use `o` to represent the object inside a class file.

- `freq`, also *read only*, is the frequency of the factor returned to users. When returning the final calculated factor to users (by `create` of `FacBase`) , `FacBase` will re-sample the factor to this frequency. This member provides a way to let your code to know the final frequency and is mainly controlled by `FacBase`, you should not change its value (though we do not impose this restriction).

secIds

a cell vector of strings representing names of securities (i.e., stocks). The factor will be calculated for these stocks.

startDate

start date of the period in which the factor is calculated.

endDate

end date of the period in which the factor is calculated.

As said in the comments of the code, you should *only* include the live version (`buildLive`) in the class if it is significantly different from the nonlive version (`build`). If they are only marginally different, you can branch your code by checking the value of `o.isLive`.

Note that `buildLive` takes few parameters because those left out are meaningless in live.

Understand Frequency

Two frequencies are used inside a factor class. The first, accessing via `o.freq` where `o` is the factor object, is the frequency of the final factor constructed by the factor class. It is specified as second parameter when users create a factor by calling (`create` can be passed more parameters in name-value pairs following `endDate`.)

```
fac = create(BKTOPRC, 'M', isLive, secIds, startDate, endDate);
```

where `'M'` stands for monthly frequency. The factor returned in `fac` then is monthly sampled. We thereafter refer it as *target frequency*.

The second, almost unnoticeable but equally important, stored as a `DataBasis` object inside the factor, is the frequency of data items involved in calculating the factor value. We refer this frequency as *calc frequency* This frequency can be the same as target frequency. But at least should be higher than the target frequency. The reason is obvious: when the factor returned to user, `FacBase` will convert the factor in calc frequency to the target frequency, and if the calc frequency is more sparser than the target frequency, information is going to be lost.

Then when and why do we need the calc frequency higher than (instead of equal to) the target frequency? Suppose we need a factor both weekly and monthly sampled, it seems we should pick up the weekly calc frequency. This is fine for weekly target frequency, but for monthly target frequency, converting from weekly to monthly is inaccurate, since one month has 30/7 weeks, and it is not an integer. So a better frequency in this case should be daily.

Choice of calc frequency also depends on the definition of the factor. Consider a factor of 5 day momentum of prices, clearly its calc frequency should be set to daily. To force a factor to use a fixed calc frequency (instead of calc frequency passed by `create`), you can set its `dateBasis` member to that frequency before loading any items:

```
    o.dateBasis = DateBasis('BD');
```

where `'BD'` indicates business days.

`DateBasis` is a class in charge of frequency conventions. Aside from `'BD'`, we have other mnemonics:

`'BD'`	business daily
`'BW'`	business weekly
`'BM'`	business monthly
`'D'`	calendar daily
`'W'`	calendar weekly
`'M'`	calendar monthly

You can also define your own calc frequency by

```
    o.dateBasis = DateBasis(freqBasis, nY, nQ, nM, nW, nD, isBusDay);
```

where

freqBasis
: a char of `'D'`, `'W'`, `'M'`, `'Q'` or `'A'`, standing for daily, weekly, monthly, quarterly and annually, respectively. Frequency indicators are interpreted by class `Freq`.

nY, nQ, nM, nW, nD
: are all integers, indicating how many periods are contained in a year, a quarter, a month, a week and a day, respectively.

isBusDay
: indicates if only business days are accounted; or, equivalently, if whe should exclude holidays. `isbusDay` is processed by class `Freq`.

For example, the `DateBasis('BD')` previously actually is defined as:

```
    o.dateBasis = DateBasis('D', 252, 21*3, 21, 5, 1, true);
```

where we abide by the convention that one year has 252 business days, and one month has 21 business days.

2.3 Write Implementation Code

Since `FacBase` does most of routine job, you only need to focus on the definition part of the factor. Taking a look at the `BKTOPRICE` and comparing it with equation (1) on page 2 could give you some sense about this point:

```
classdef BKTOPRICE < FacBase
    methods (Access = protected)
```

```
        function factorTS = build(o, secIds, startDate, endDate)
            % 1 Load data items
            bookValue = o.loadItem(secIds,'D000686133',startDate,endDate,4);
            closePrice = o.loadItem(secIds,'D001410415',startDate,endDate);
            shares = o.loadItem(secIds,'D001410472',startDate,endDate);

            % 2.1 Calculate the mean of last n-quarter book value for each point in time
            bookValueMean = ftsnanmean(bookValue{:});
            bookValueMean = backfill(bookValueMean, o.DCF('3M'),'entry');

            % 2.2 Here the equation 1
            factorTS = (1000000*bookValueMean./shares)./closePrice;
        end
    end
end
```

Basically, the implementation follows a regular procedure:

1. Load data items using `o.loadItem`. [3] If you loading multiple items using this function with the same parameters of `secIds`, `startDate` and `endDate`, then the items you finally get are already aligned along both the time and the field dimension.
2. Plug items into definition equation(s). Some equations may require a bit more complex logic and more code than a few lines, sometimes even need to write some help functions. So this is the key part you should focus on.
3. Do any necessary processing of the calculated factor, like backfilling (though backfilling is not necessary in most cases, since `o.loadItem` will do backfilling according to the natural frequency of the raw data items. Also note that `FacBase` also set all `infs` to `NaNs`.

This pattern is not mandatory. Based on factors, you may employ different ways to implement factors. However, always keep in mind that item aligning, item backfilling, re-sampling the final factor to target frequency, and others mentioned above have been done by the framework.

Sometimes a group of factors may share the same structure, just a few things different. In this case, it is better to abstract a base class (derived from `FacBase`, of course) which do the common work, and then concrete factor classes, each do their own unique things.

If you have decided to add a live version of build (`buildLive`), follow the same guild line. But even you do not have one, the framework will provide it for you (defined in `FacBase` and inherited by your own factor classes), and

```
   o.buildLive(secIds, runDate)      % o.isLive == false
```

is equivalent to

[3] Note not all data items can be loaded this way, see explanation below.

```
o.build(secIds, runDate, runDate) % o.isLive == true, same start date and end date
```

bearing in mind that `isLive` has different values corresponding to live and back-test cases.

2.4 Test

Test you code by calling `create`:

```
1 % When isLive == false, create actually call build
2   fac = create(factor_object, isLive, targetFreq, secIds, startDate, endDate, 'name', val,...)
3 % When isLive == true, create actually call buildLive
4   fac = create(factor_object, isLive, secIds, runDate, 'name', val,...)
```

Parameters `secIds`, `startDate`, `endDate` and `runDate` have the same meanings as in `build` and `buildLive` explained in last step (subsection 2.3).

The first parameter, `factor_object`, is an (usually empty) factor object that can be created either by direct putting the class (default) constructor there: [4]

```
fac = create(BKTOPRICE, ...)
```

or through a function handle to the class constructor

```
fac = create(fun_handle(), ...)  % fun_handle should be @BKTOPRICE
```

The pair of parentheses after the function handle is must, otherwise MATLAB thinks you are passing a function handle instead of the factor object.

The second parameter, `isLive`, decides what parameters should be followed. If it is `true`, it tell `create` to create the live version of the factor and can only be called as shown in line 4 in the above code snippet; if it is `false`, back-test version of factor is going to be created as shown in line 2 in the same code snippet.

From third parameter on, the two versions of build diverge. For `build`, it is target frequency. `buildLive` does not need that parameter since it only need to return values for the `runDate`.

The `'name'`-`val` pairs are used to provide additional information. `'name'` can be of the following:

`'id'`	factor id
`'name'`	factor name
`'type'`	factor type
`'higherTheBetter'`	true if higher the factor value, the better the factor; false the other way around
`'dateBasis'`	for calc frequency, val passed should be type of `DateBasis`

[4] Because we never implement a constructor for factor classes, MATLAB produces a default constructor. Also note that constructor has the same name as the class.

Usually, `'id'`, `'name'`, `'type'` and `'higherTheBetter'` are values from factor registering database (if the factor has been registered). For an example of using these `'name'-val` parameters, see `LoadFactorTS.m` which not only load factor values, but also its associated registered information. For how to register a factor, go to next step.

Finally note that `build` and `buildLive` can not be called directly since factor creation need `create` to do the common routine jobs while `build` to deal with the factor definition.

2.5 Register Factor

Having a factor been done in coding, debugging and testing, you should register it into the database, so that it can be known by and used in models. Here is the format:

```
id = Factory.Register2DB(...
      'Book to Price Ratio' ...                           % name
    , 'Book value per share divided by price per share' ... % description
    , 'BKTOPRICE' ...                                     % name of factor class
    , true ...                                            % is it the higher the better?
    , true ...                                            % is it active currently?
    , true);                                              % can it be used in production?
```

The `id` returned is an unique string of form like `'F00172'` and can be referenced when- and wherever it is needed, like in `LoadFactorTS` when you load populated factors, in factor combination (blending) where you need to specify factors to be combined, etc.

The database table for keeping factor registering information is `quantstrategy.fac.factormstr`.

2.6 Populate

When needing a factor, you can call `LoadFactorTS` providing it with factor id and other information. `LoadFactorTS` can either calculate the factor on the fly or read it from database where pre-calculated values of factors has been populated. There are pros and cons for both ways: The benefits of on-the-fly is obvious: it can always reflect newest changes in data items and easy to maintain the consistence of factors, not to mention it also reduces data redundancy. Equally obvious is its drawback: it is slow in our current database infrastructure.

Populating factor is on the opposite side of on-the-fly. The class `Factory` is used to do the populating and if needed, please refer to the demo code in `runRegistered.m` and `runRegisteredLive.m`.

2.7 More Examples

For helping you get more sense about creating factors, we present two more examples in this subsection, each of which shows different aspects of factor creation.

5-Day Price Reversal (REVSAL5D)

Price reversal is a well-know short-term anomaly. The expectation is that stocks that have performed very well in the last 5 days prior to stock selection days will underperform over the next month and vice versa. Its formula is

$$\text{5-Day Price Reversal} = Z\text{-Score}_{\text{cross sectional}}\left(\frac{\text{Price}_{\text{current}}}{\text{Price}_{\text{5 days lag}}} - 1\right).$$

Support we want monthly factors (i.e., target frequency is `'M'`), what calc frequency should you choose? Since the factor itself is defined based on daily, the calc frequency definitely shoud be daily. In this case, based on the nature the factor should decide its calc frequency instead of setting outside by `create` (Remember the `'name'`-`val` parameter can be used to pass a `'dateBasis'`). That is the first thing to do in the code: changing the `dateBasis` to daily (actually, business daily, since price only quoted in business day) no matter what it is set outside by `create` (and therefore by the framework).

```
1  classdef REVSAL5D < FacBase
2      methods (Access = protected)
3          function factorTS = build(o, secIds, startDate, endDate)
4              o.dateBasis = DateBasis('BD'); % force the calc frequency to be 'BD'
5
6              sDate = datestr(addtodate(datenum(startDate),-1,'M'),'yyyy-mm-dd');
7              closePrice = o.loadItem(secIds,'D001410415',sDate,endDate);
8              closePrice = backfill(closePrice,o.DCF('3D'),'entry');
9              factorTS = Price2Return(closePrice,5);
10         end
11     end
12 end
```

Note also how we use `DCF` function to calculate a given tenor. A tenor is a period or duration represented by something like `'5D'` (5 days), `'2M'` (2 months), `'6W'` (6 weeks), etc. The units can be used in a tenor representation are the same as frequency units (i.e., `'D'`, `'W'`, `'M'`, `'Q'`, `'Y'`). `DCF`, standing for *Date Conversion Factor*, converts the tenor given in the first parameter to an integer number based on the calc frequency; the number is the number of basic period represented by calc frequency; if it is omitted, it is assumed to be 1. For example, if calc frequency is `'BD'`, then `o.DCF('1M')` equals to 21, meaning in one month there are 21 business days. In current example, since we force the calc frequency to be `'BD'`, `o.DCF('3D')` in line 8 is 3. Since the conversion is based on calc frequency represented by `dateBasis` of the factor object `o`, the items involved (`closePrice`) must also be obtained by methods in `FacBase`; to be specific, methods of `loadItem` and `loadBondYield` for now.

For items (`myfints` objects) obtained by other ways, you can set its frequency to be the same as calc frequency; for example:

```
   gics = LoadQSSecTS(secIds, 913, 0, startDate, endDate, o.dateBasis.freqBasis);
```

then you can still use `DCF` as before:

```
   gics = lagts(gics, o.DCF('1M'));
   gics = backfill(gics, o.DCF('1M');
```

Finally, we mention here that `FacBase` provides shortcuts for `lagts` and `leadts`:

```
fts = o.lagfts(fts, '3M')   <==> fts = lagts(fts, o.DCF('3M'));
fts = o.leadfts(fts, '3M')  <==> fts = leadts(fts, o.DCF('3M'));
```

Blended EPS Yield (BEPSY)

Rather than behaving rationally as implied in standard financial theory, investors tend to make systematic cognitive errors which a truly objective investor can exploit. Such errors include overreacting to bad news, confusing a bad company with a bad stock, and assuming poorly performing stocks will continue to behave badly. Each of these can result in an inappropriately low stock price which can lead to investment opportunity.

The formula of the factor is therefore defined as

$$\text{Blended EPS Yield} = \frac{\text{Blended Stock EPS}}{\text{Stock Price}}$$

where blended stock EPS is average of the actual EPS and estimate EPS if both are available, else it is the one available if only one of them is available, otherwise, blended stock EPS is set to null

Here's the code:

```
classdef BEPSY < FacBase
    methods (Access = protected)
        function factorTS = build(o, secIds, startDate, endDate)
            % set the sDate 3 month earlier than the input startDate to ensure some look back
            % for the first observation
            sDate = datestr(addtodate(datenum(startDate),-3,'M'),'yyyy-mm-dd');
            EPS_Act = o.loadItem(secIds,'D000432130',sDate,endDate);
            if o.isLive
                EPS_FY1 = o.loadItem(secIds,'D000448965',sDate,endDate); % QFS FY1 EPS mean
            else
                EPS_FY1 = o.loadItem(secIds,'D000435584',sDate,endDate);
            end
            closePrice = o.loadItem(secIds,'D001410415',sDate,endDate);

            % match the items
            EPS_Act = backfill(EPS_Act, '2M', 'entry');
            EPS_FY1 = backfill(EPS_FY1, '2M', 'entry');

            % calculate factor value
            EPS_Blended = ftsnanmean(EPS_Act, EPS_FY1);
            factorTS = EPS_Blended ./ closePrice;
        end
```

```
23     end
24 end
```

Note that all income statement items are annualized by summing up last four quarter values. But the most notible thing is how we use `o.isLive` to differentiate live and back-test version of build in a single `build` function instead of having them seperately in `build` and `buildLive`.

Finally, we emphasis here that unlike last example you can replace `o.DCF('3D')` by 3 since forcing of calc frequency, here you must stick to `o.DCF('2M')` (lines 16 and 17) because you don't know what the calc frequency is (it is determined outside).

3 Database and Related Functions

3.1 Structure Overview

Quant database is hosted in 2 servers: `iGradeSqlDev1` which is for vendor-sourced data (raw data) and `CommonSQLDEV3` which is for quant-generated data (processed data). The server name `CommonSQLDEV3` is used in quantitative model research and development. `CommonSQLDEV3` is also used for production by mapping it to `CommonSQLProd2`.

Quants have full access permission to `CommonSQLDEV3` and read-only permission to `igradeSqlDev` (via `datainterfaceserver`).

`iGradeSqlDev1`, as a raw data center, holds data from different sources (vendors), each with its own data format, accessing stored procedures, even different security ids. So a group of tables in database `DataQA` are used to manage these differences:

api.DataSourceMstr
: holds a mapping of data sources (vendors),

api.IndexMstr
: holds ids of various indexes and their descriptions.

api.ItemMstr
: holds item ids (related to securities and indexes) and their descriptions.

api.SecMstr
: holds security ids and their associated characteristics. Aside from security id mappings (e.g., quant id to Thomas Reuters Id), `Ticker`, `RoundLotSize`, `GICS` (latest available), `Country`, `MSCICtry`, `ISOCurId` can be obtained from this table.

api.SecTypeMStr
: holds id prefixes which indicate the type of securities. Every security id in `api.SecMstr` is

prefixed with its type id, like `'E0059'`, `'X0005'`, etc.

There are two views for economic data (e.g., interest rates). They are specific to data vendors and not related to equity markets.

api.bbmstr
: economic data items from Bloomberg

api.dslmst
: economic data items from DataStream

3.2 Data Cache

The database structure actually is an image of that vendors provide us. The store procedures used to access data are based on those from vendors too, which are not designed for large scale accessing like in cases of our quantitative research. Full information for some data items (e.g., CIQ point in time data) scatter different tables and acquiring them results huge joining and speed usually is painfully slow. To address this problem, we introduced data cache mechanism.

The idea of data cache pretty simple: preloading data items needed into database tables (these tables are the cache), then whenever we need these items, we load them from the cache tables instead of from scattered tables. Managing and maintaining the cache, however, is not so straightforward and need care.

To cache data, we create a dedicated database, `DataCache`, in server `igradesqldev1`, which contains 3 tables: [5]

api.itemmstr
: holds information of all data items being cached, including time series items and point-in-time items. Any item occurred in this table is assumed to have been cached.

api.TS
: holds data for time series data items.

api.CIQFilingData
: holds data for point-in-time data items.

The number of items registered in `api.itemmstr` should equal to the sum of number of (unique) items in `api.TS` and `api.CIQFilingData`. All the data in cache are in their raw format: when you load them, the data returned are just like loading directly from vendors' database, but much faster.

Note we are only cache items formally used in factors and having accessing bottleneck. Factors that do not have performance problem will not be cached, like those of broker factors (`D0024`).

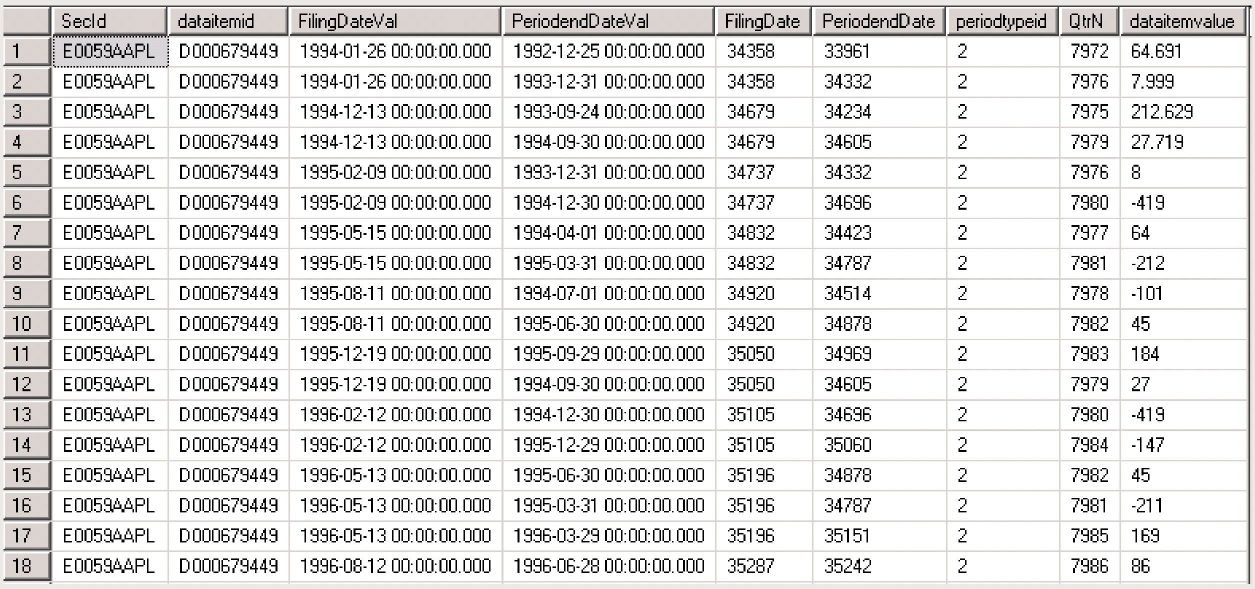
The data cache is transparent to end users: users do not need to know if an item being loaded is cached or not when loading it using `LoadRawItemTS` and `LoadRawItemPIT` functions described later

[5] There are other auxiliary tables for refreshing cache.

in this section; these functions will automatically check cache, and load from cache only if cache is available. See references in subsection **??** to the functions for more details.

How we process PIT data

PIT, standing for point in time, refers to data not only with their corresponding period dates just like in ordinary time series data, but also with their filing (or publishing) dates. At a certain point in time, we can only observe data latest available in relative to filing dates. A typical PIT data item looks like



	SecId	dataitemid	FilingDateVal	PeriodendDateVal	FilingDate	PeriodendDate	periodtypeid	QtrN	dataitemvalue
1	E0059AAPL	D000679449	1994-01-26 00:00:00.000	1992-12-25 00:00:00.000	34358	33961	2	7972	64.691
2	E0059AAPL	D000679449	1994-01-26 00:00:00.000	1993-12-31 00:00:00.000	34358	34332	2	7976	7.999
3	E0059AAPL	D000679449	1994-12-13 00:00:00.000	1993-09-24 00:00:00.000	34679	34234	2	7975	212.629
4	E0059AAPL	D000679449	1994-12-13 00:00:00.000	1994-09-30 00:00:00.000	34679	34605	2	7979	27.719
5	E0059AAPL	D000679449	1995-02-09 00:00:00.000	1993-12-31 00:00:00.000	34737	34332	2	7976	8
6	E0059AAPL	D000679449	1995-02-09 00:00:00.000	1994-12-30 00:00:00.000	34737	34696	2	7980	-419
7	E0059AAPL	D000679449	1995-05-15 00:00:00.000	1994-04-01 00:00:00.000	34832	34423	2	7977	64
8	E0059AAPL	D000679449	1995-05-15 00:00:00.000	1995-03-31 00:00:00.000	34832	34787	2	7981	-212
9	E0059AAPL	D000679449	1995-08-11 00:00:00.000	1994-07-01 00:00:00.000	34920	34514	2	7978	-101
10	E0059AAPL	D000679449	1995-08-11 00:00:00.000	1995-06-30 00:00:00.000	34920	34878	2	7982	45
11	E0059AAPL	D000679449	1995-12-19 00:00:00.000	1995-09-29 00:00:00.000	35050	34969	2	7983	184
12	E0059AAPL	D000679449	1995-12-19 00:00:00.000	1994-09-30 00:00:00.000	35050	34605	2	7979	27
13	E0059AAPL	D000679449	1996-02-12 00:00:00.000	1994-12-30 00:00:00.000	35105	34696	2	7980	-419
14	E0059AAPL	D000679449	1996-02-12 00:00:00.000	1995-12-29 00:00:00.000	35105	35060	2	7984	-147
15	E0059AAPL	D000679449	1996-05-13 00:00:00.000	1995-06-30 00:00:00.000	35196	34878	2	7982	45
16	E0059AAPL	D000679449	1996-05-13 00:00:00.000	1995-03-31 00:00:00.000	35196	34787	2	7981	-211
17	E0059AAPL	D000679449	1996-05-13 00:00:00.000	1996-03-29 00:00:00.000	35196	35151	2	7985	169
18	E0059AAPL	D000679449	1996-08-12 00:00:00.000	1996-06-28 00:00:00.000	35287	35242	2	7986	86

where `SecId` is the security ids, `dataitemid` the id of the PIT data item, `FilingDateVal` and `FilingDate` are filing dates, i.e., the date on and after that the public know corresponding data, `PeriodEndDateVal` and `PeriodEndDate` are the corresponding data periods, i.e. periods the data are reported for. `QtrN` stands for quarter number, is unique to each `PeriodEndDate` for a certain security and data item. We pack PIT raw data into `myfintss` this way:

1. *Determine the latest* `QtrN`. For an observation date, the latest `QtrN` is the one corresponding to the latest `FilingDate` before the observation date, with restriction that `FilingDate` can not be earlier than the observation date more than 360 days. If the latest `FilingDate` is too earlier than the observation date, we treat it as unavailable.

2. *Get data based on the determined latest available* `QtrN`. Suppose, we are loading PIT data 4 quarters back and the latest `QtrN` for current observation date is 7980, then we should load data corresponding to `QtrN` of 7980, 7979,7978, and 7977. If any of them missing, we treat them as unavailable.

The actual process has more involved, in consideration of efficiency.

3.3 Factor Database

We have several tables in database `QuantStratgy` located in server `CommonSQLDEV3` to hold precalculated factors. As we briefly mentioned in section 2.6, factor can be calculated on-the-fly, or populated to database for later use. We also have a comparison of the two ways to get factors in that section. It is a good point to review that section if you are not familiar with these concepts yet.

Following tables are used for managing populated factors:

fac.factormstr

factor master table, holding registered factor information. See subsection 2.5 for how to register a factor.

fac.factortypemstr

factor category table, holding factor types (value, growth, momentum, etc. stuff), referenced by `factorTypeId` in table `fac.factormstr` and `type` property of factor object in MATLAB.

fac.FactorTS_Live

factor time series table used in production

fac.FactorTS_BT

factor time series table used in backtesting. The difference from live is that they were populated with a long history and use `build` version of factor classes (live use `buildLive`). See subsection 2.2 for more information.

3.4 Loading Data to MATLAB

All the database related utility functions are placed in `$/QuantStrategy/Analytics/Utility/myfintsUtility/`, while the core functionalities is organized as a class in `$/QuantStrategy/Analytics/Utility/DB/DB.m`.

LoadIndexHoldingTS

Load index holding (list of component securities and index time series) from database.

Usage

```
[secIds,holdingTS] = LoadIndexHoldingTS(aggId, startDate, endDate, isLive, targetFreq)
```

Input

aggId

the index ID of the stock benchmark defined in quantstaging.dbo.aggmstr

startDate/endDate

two strings specify the range of dates to retrieve

isLive

indicate whether to retrieve live holding (latest based on endDate), or the historical holding between startDate and endDate

targetFreq

target frequency of the resulting time series object, can be

```
1, DAILY, Daily, daily, D, d
2, WEEKLY, Weekly, weekly, W, w
3, MONTHLY, Monthly, monthly, M, m
4, QUARTERLY, Quarterly, quarterly, Q, q
5, SEMIANNUAL, Semiannual, semiannual, S, s
6, ANNUAL, Annual, annual, A, a
```

the same as `convertto()` of MATLAB's fints.

Output

secIds

The list of distinct secid, as defined in quantstaingdbo.secmstr

holdingTS

The holding time series, encapsulated as an object of `myfints`.

LoadSecInfo

Load information of all securities within an index.

Usage

```
secInfo = LoadSecInfo(aggId, colNames, startDate, endDate)}
```

Input

aggId

the index ID of the stock benchmark defined in quantstaging.dbo.aggmstr

colNames

a cell array of the column names to retrieve from quantstaging.dbo.secmstr

startDate/endDate

two strings specify the range of dates to retrieve item value

Output

secInfo

a structure which contains information of stocks.

LoadRawItemTS

Load raw data of specified data items for specified stocks. It only works for the raw data provided by data vendors, not for in-house derived items.

Usage

```
    itemTS = LoadRawItemTS (secIds, itemId, startDate, endDate, targetFreq)
```

Input

secIds

a cell array of security id as defined in quantstaging.dbo.secmstr

itemId

the ID of raw items defined in DataInterfaceServer.dataqa.api.itemmstr

startDate/endDate

two strings specify the range of dates to retrieve item value

targetFreq

target frequency of the resulting time series object, can be

```
1, DAILY, Daily, daily, D, d
2, WEEKLY, Weekly, weekly, W, w
3, MONTHLY, Monthly, monthly, M, m
4, QUARTERLY, Quarterly, quarterly, Q, q
5, SEMIANNUAL, Semiannual, semiannual, S, s
6, ANNUAL, Annual, annual, A, a
```

the same as `convertto()` of MATLAB's fints.

Output

itemTS

a `myfints` object which contains the time series values of items in desired frequency

LoadRawItemPIT

This function retrieves the Point-In-Time raw item value by calling DataQA API.

Usage

```
  ftsArray = LoadRawItemPIT (secIds, itemId, startDate, endDate, numQtrs, targetFreq)
```

Input

secIds

a cell array of security id as defined in quantstaging.dbo.secmstr

itemId

the ID of raw items defined in DataInterfaceServer.dataqa.api.itemmstr

startDate/endDate

two strings specify the range of dates to retrieve item value

numQtrs

number of quarters to look back, as defined for Compustat PIT data.

targetFreq

target frequency of the resulting time series object, can be.

```
1, DAILY, Daily, daily, D, d
2, WEEKLY, Weekly, weekly, W, w
3, MONTHLY, Monthly, monthly, M, m
4, QUARTERLY, Quarterly, quarterly, Q, q
5, SEMIANNUAL, Semiannual, semiannual, S, s
6, ANNUAL, Annual, annual, A, a
```

the same as `convertto()` of MATLAB's fints.

Output

ftsArray

an array of `myfints` objects whose size equals the input `numQtrs`. `ftsArray{i}` is the time series of latest i-th quarter value at each point in time.

LoadIndexItemTS

This function retrieves time series of index level item (e.g. index close price of S&P500.)

Usage

```
indexTS = LoadIndexItemTS (aggId, itemId, startDate, endDate, targetFreq)
```

Input

aggId

the index ID of the stock benchmark defined in quantstaging.dbo.aggmstr

itemId

the ID of index item defined in DataInterfaceServer.dataqa.api.itemmstr

startDate/endDate

two strings specify the range of dates to retrieve item value

targetFreq

target frequency of the resulting time series object, can be

```
1, DAILY, Daily, daily, D, d
2, WEEKLY, Weekly, weekly, W, w
3, MONTHLY, Monthly, monthly, M, m
4, QUARTERLY, Quarterly, quarterly, Q, q
5, SEMIANNUAL, Semiannual, semiannual, S, s
6, ANNUAL, Annual, annual, A, a
```

the same as `convertto()` of MATLAB's fints.

Output

IndexTS

a `myfints` object which contains the time series values of index in desired frequency.

LoadFactorInfo

Load factor information from database.

Usage

```
  factorInfo = LoadFactorInfo (factorIds, colNames)
```

Input

factorIds

a cell array of factorIds, as defined in quantstrategy.fac.factormstr

colName

a cell array of colums in the factormstr table to retrieve.

Output

factorInfo

a structure which contains the information in specified columns

LoadFactorTS

Load factor time series data that is already stored in the DB.

Usage

```
  factorTS = LoadFactorTS (secIds, factorId, startDate, endDate, isLive, targetFreq)
```

Input

secIds

a cell array of security id as defined in quantstaging.dbo.secmstr

factorId

the ID of factor defined in quantstrategy.fac.factormstr

startDate/endDate

two strings specify the range of dates to retrieve item value

isLive

`true` or `false`, to indicate whether to retrieve the backtest factor value or live factor value

targetFreq

target frequency of the resulting time series object, can be

```
1, DAILY, Daily, daily, D, d
2, WEEKLY, Weekly, weekly, W, w
3, MONTHLY, Monthly, monthly, M, m
4, QUARTERLY, Quarterly, quarterly, Q, q
5, SEMIANNUAL, Semiannual, semiannual, S, s
6, ANNUAL, Annual, annual, A, a
```

the same as `convertto()` of MATLAB's fints.

You can access factor information loaded by this function as following:

```
    factorTS.id                 % like 'F00001'
    factorTS.name               % like 'BKTOPRICE'
    factorTS.higherTheBetter    % 1 or 0
    factorTS.isLive             % 1 or 0
    factorTS.type               % 1 to 5
```

Output

FactorTS

a `myfints` object that contains the time series of factors in desired frequency.

LoadFXTS

Load the FX rate time series from `dataqa` and pack it as a `myfints` object.

Usage

```
  FXTS = LoadFXTS(fromCur, toCur, startDate, endDate, targetFreq)
```

Input

fromCur

(string) iso currency id of the currency used to quote (terms currency, numerator).

toCur

(string) iso currency id of the currency to be quoted (base currency, denominator).

startDate

(string) Start time

endDate

(string) End time

targetFreq

(optional) the target freqency of the retrieved financial time series, can be:

```
1, DAILY, Daily, daily, D, d
2, WEEKLY, Weekly, weekly, W, w
3, MONTHLY, Monthly, monthly, M, m
4, QUARTERLY, Quarterly, quarterly, Q, q
5, SEMIANNUAL, Semiannual, semiannual, S, s
6, ANNUAL, Annual, annual, A, a
```

the same as `convertto()` of MATLAB's fints.

Output

FXTS

The FX rate time series.

LoadQSAggTS

Load the specified item time series for all securities in aggIds and pack it as a `myfints` object.

Usage

```
itemTS = LoadQSAggTS(aggIds, itemId, startDate, endDate, targetFreq)
```

Input

aggIds

A cell array of index id as defined in `quantstaging.dbo.aggmstr`

itemId

The `itemID` of the item defined in `quantstaging.dbo.itemmstr`

startDate

(string) Start time

endDate

(string) End time

targetFreq

(optional) the target freqency of the retrieved financial time series, can be:

```
1, DAILY, Daily, daily, D, d
2, WEEKLY, Weekly, weekly, W, w
3, MONTHLY, Monthly, monthly, M, m
4, QUARTERLY, Quarterly, quarterly, Q, q
5, SEMIANNUAL, Semiannual, semiannual, S, s
6, ANNUAL, Annual, annual, A, a
```

the same as `convertto()` of MATLAB's fints.

Output

itemTS

The item time series

LoadQSSecTS

Load the specified item time series from `quantstaging.dbo.secTS` table for all securities in `SecIds` and pack it as a `myfints` object.

Usage

```
itemTS = LoadQSSecTS(secIds, itemId, seq, startDate, endDate, targetFreq)
```

Input

secIds

A cell array of security id as defined in `quantstaging.dbo.secmstr`

itemId

The `itemID` of the item defined in `quantstaging.dbo.itemmstr`

startDate

(string) Start time

endDate

(string) End time

targetFreq

(optional) the target freqency of the retrieved financial time series, can be:

```
1, DAILY, Daily, daily, D, d
2, WEEKLY, Weekly, weekly, W, w
3, MONTHLY, Monthly, monthly, M, m
4, QUARTERLY, Quarterly, quarterly, Q, q
5, SEMIANNUAL, Semiannual, semiannual, S, s
6, ANNUAL, Annual, annual, A, a
```

the same as `convertto()` of MATLAB's fints.

Output

itemTS

The item time series

3.5 Saving Data to Database

Save2DB

Save the factor time series into database.

Usage

```
Save2DB(factorTS, factorId, isLive)
```

Input

factorTS

a factor object (or `myfints` object) which contains the time series values of factors.

factorId

the ID of factor defined in qunatstrategy.fac.factormstr.

isLive

1 or 0, to indicate whether to retrieve the backtest factor value or live factor value.

Output

None

Factory.Register2DB

Register a factor into database.

Usage

```
factorId = Factory.Register2DB(factorName, factorDesc, factorClass, isHighBetter, isActive, isProd)
```

Input

factorName
the name of the registered factor

factorDesc
the description of the registered factor

factorClass
the MATLAB class name of the factor

isHighBetter
1 or 0, indicate whether higher the factor value, better the stock according to factor rationale

isActive
1 means this factor is currently under active research or production, 0 means this factor is pended or stopped and shouldn't be used for any further research.

isProd
1 means this factor is finalized and will be in production (schedule job), 0 means the factor is still under development.

Output

factorId
the registered factor Id.

3.6 Data Process Functions

CashFlowDecomp

Decompose the rolling cumulative cash flow items to quarterly cash flow.

Usage

```
resultCF = CashFlowDecomp(CF, FQTR)
```

Input

CF A myfints object, cash flow items directly retrieved from compustat (has to be in quarterly or monthly freq)

FQTR

A myfints object, NO. of financial quarters for each stock (has to be in quarterly or monthly freq)

Output

resultCF

The decomposed cash flow

CashFlowDecompPIT

PIT version of `CashFlowDecomp`, decomposing the rolling cumulative cash flow items to quarterly cash flow.

Usage

```
resultCF = CashFlowDecompPIT(CF, FQTR)
```

Input

CF a cell array of `myfints` objects, which are the Compustat Point-In-Time cash flow item directly retrieved by `LoadRawItemPIT` function. `CFi` is the ith-quarter-back cash flow value at each point in time.

FQTR

a cell array of `myfints` object, which are the Compustat Point-In-Time financial quarters for each stock.

Output

resultCF

a cell array of `myfints`, which are the decomposed cash flow.

genDateSeries

Generate a specified date series.

Usage

```
resultDate = genDateSeries(startDate, endDate, targetFreq, paraname, paravalue, ...)
```

Input

startDate

a date string, indicating starting date of the generated date series.

endDate

a date string, indicating ending date of the generated date series.

targetFreq

indicates frequency of the generated date series, which can be

```
1, DAILY, Daily, daily, D, d
2, WEEKLY, Weekly, weekly, W, w
3, MONTHLY, Monthly, monthly, M, m
4, QUARTERLY, Quarterly, quarterly, Q, q
5, SEMIANNUAL, Semiannual, semiannual, S, s
6, ANNUAL, Annual, annual, A, a
```

the same as `convertto()` of MATLAB's fints.

paraname, paravalue,...

Named arguments provide additional information.

Parameter	Default	Meaning	Freqs Appl.
'EOW'	0	0 - 6, Specifies the end-of-week day: 0–Friday, 1–Saturday, 2–Sunday, 3–Monday, 4–Tuesday, 5–Wednesday, 6–Thursday	weekly(2)
'ED'	0	The end of date (of month) in a period. 0 is the last day (or last business day), 1-31 is the specific day (if beyond end of month, adjust to end of month)	monthly(3), quarterly(4), semiannual(5), annual(6)
'EM'	3	Last month of the first period (e.g., quarter, year). All subsequent quarterly dates are based on this month	quarterly(4), semiannual(5), annual(6)
'Busdays'	0	Either 0 or 1, indicating whether business days (non-weekend and non-holidays) are counted. If a sampling day is fallen in a non-business day, it will be adjusted to the next business day, unless there's no business day in the current frequency period (e.g., current month), in which case prior business day will be used.	All frequencies
'Holidays'	NYSE	A vector specifying holidays	All frequencies
'Weekend'	[1 0 0 0 0 0 1]	Specify which day is weekend in order of [Sun Mon Tue Wed Thu Fri Sat]	All frequencies

Output

resultDate

a numeric vector representing generated dates.

LatestDataDate

Return the latest no-data-missing date for each field in the `myfints` object.

Usage

```
result = LatestDataDate(ifts)
```

Input

ifts a `myfints` object.

Output

result

a structure containing two fields:

- `result.field` - a cell array of the data field names of `ifts`
- `result.latestDate` - a cell array of corresponding date string which are the latest no-data-missing date for each field.

Live2PIT

Convert the live fundamental data to Point-In-Time format. It simply stacks the live data (in plain time series format) into multiple `myfints` objects each of which contains only one observation and corresponds to a time point. Using this function could make you write more consistent code for both `buildLive` and `build` which usually contain the same logic.

Usage

```
ftsArray = Live2PIT(ifts, nPeriod)
```

Input

ifts a `myfints` which contains raw fundamental data (have not been backfilled).

nPeriod

(int) number of periods data to convert to PIT.

Output

ftsArray

an cell array of `myfints`, each `myfints` only has one date: the latest date in `ifts`, and the ith `myfints` in this array is the ith period looking back from that date.

4 Use of myfints

This section covers the most important methods specific to `myfints`. It does not enumerate all the methods of `myfints` however. For a complete list of methods, type either `methods(myfints_obj)` or `methods(myfints)`.

4.1 Aligning Data

aligndata

Align inputted `myfints` objects both along time and fields.

Usage

```
[ofts1, ofts2, ...] = aligndata(ifts1, ifts2, ...)
[ofts1, ofts2, ...] = aligndata(ifts1, ifts2, ..., mode)
[ofts1, ofts2, ...] = aligndata(ifts1, ifts2, ..., aligningdates)
[ofts1, ofts2, ...] = aligndata(ifts1, ifts2, ..., mode, aligningdates)
[ofts1, ofts2, ...] = aligndata(ifts1, ifts2, ..., aligningdates, mode)
[ofts1, ofts2, ...] = aligndata(..., 'CalcMethod', calcMethod)
```

Where

ifts1, ifts2,...

`myfints` objects to be aligned. Multiple such objects are allowed.

ofts1, ofts2,...

returned `myfints` objects that have been aligned. Each corresponds to and has the same type of the inputted ones; i.e., if the original one is of type `BKTOPRICE`, the corresponding aligned one is still of the same type.

mode

control how the fields in time series aligned. Values can be one of

- `'union'`. include all (unique) fields occurred in all inputted `myfints` objects. NaNs are filled for original nonexistent fields.
- `'intersect'`. only take the common fields in all inputted `myfints` objects. Fields specific to a certain `myfints` object are removed.

DEFAULT is `'intersect'`.

aligningdates

A vector of dates (numeric vector or cell vector of date string) to be aligned, OR

a char vector (string) or scalar number indicating aligned frequency which can be

```
1, DAILY, Daily, daily, D, d
2, WEEKLY, Weekly, weekly, W, w
3, MONTHLY, Monthly, monthly, M, m
4, QUARTERLY, Quarterly, quarterly, Q, q
5, SEMIANNUAL, Semiannual, semiannual, S, s
6, ANNUAL, Annual, annual, A, a
```

the same as `convertto()` of MATLAB's fints.

- All the inputted `myfints` objects will be aligned against this date vector or specified frequency. Returned objects being aligned have the `aligningdates` (when it is a date vector) or equivalent (when it is a frequency) as their date field.

- For returned objects, on a certain date, data are the most recent available (known) data between the date (inclusive) and its previous date (exclusive).
  If no data in inputted objects available in a period (specified or divided by `aligneddates`), `NaN`s will be filled.
- If NOT provided, aligned dates will be the common dates in all inputted objects.

calcMethod

Following a char-type string indicator parameter `'CalcMethod'` (case-insensitive), specifies which method is used to calculate resampling sample data. Methods could be one of

- `'CumSum'` Returns the cumulative sum of the values between each sampling period (e.g., week, month, etc.) Data for missing dates are given the value 0.
- `'Exact'` Returns the exact value at the end-of-sampling-period date. No data manipulation occurs.
- `'Nearest'` (*default*) Returns the values located at the end-of-sampling-period date. If there is missing data, Nearest returns the nearest data point preceding the end-of-sampling-period date.
- `'SimpAvg'` Returns an averaged value over each sampling period that only takes into account dates with data (non NaN) within each quarter.

aligndates

aligns inputted `myfints` objects along time.

Usage

```
[ofts1, ofts2, ...] = aligndates(ifts1, ifts2, ...)
[ofts1, ofts2, ...] = aligndates(ifts1, ifts2, ..., aligningdates)
[ofts1, ofts2, ...] = aligndates(..., 'CalcMethod', calcMethod)
```

This function is similar to `aligndata`, except it aligns data only along date dimension. See the explanation of arguments there.

alignfields

aligns inputted `myfints` objects along fields.

Usage

```
[ofts1, ofts2, ...] = alignfields(ifts1, ifts2, ...)
[ofts1, ofts2, ...] = alignfields(ifts1, ifts2, ..., mode)
```

This function is similar to `aligndata`, except it aligns data only along fields dimension. See the explanation of arguments there.

isaligneddata, isalignedfields, isaligneddates

Determine if `myfints` objects given in arg list are aligned both/either in time and fields.

Usage

```
isaligned = isaligneddata(fts1, fts2, ...)
isaligned = isalignedfields(fts1, fts2, ...)
isaligned = isaligneddates(fts1, fts2, ...)
```

Where

`tfs1`, `tfs2`, etc. are `myfints` objects to be determined. If they are aligned both in time and fields, returns true; otherwise false.

Note that this function is different from `iscompatible(fts1,fts2)` which is inherited from MATLAB's `fints` and returns true as long as the two objects have the same fields and date irrespective of field order.

4.2 Filling Missing Data

backfill

Fill time series backwards where the values are `NaN`s with previous latest available data.

Usage

```
fts = backfill(old_fts, lookbackperiod, mode)
```

Input

old_fts

a `myfints` object to be filled.

lookbackperiod

number of periods to look back for filling. If data at a time is `NaN`, the function will look back at most `lookbackperiod` periods and find the latest available data in these past periods to fill the `NaN`. If you want to look back through to the very beginning of time series, just put this value as `inf`.

mode

can be `'row'` or `'entry'`, meaning filling occurs only when the whole row are `NaN`s or whenever an element is a `NaN`, respectively.

Output

fts the backfilled `myfints` object.

forwardfill

Fill time series forwards where the values are NaNs with most recent available data in the future.

Usage

```
fts = forwardfill(old_fts, lookfwdperiod, mode)
```

Input

old_fts

a myfints object to be filled.

lookfwdperiod

number of periods to look forward (into future) for filling. If data at a time point is NaN, the function will look forward at most lookfwdperiod periods and find the most recent available data in future these periods to fill the NaN. If you want to look forward through to the last period, just put this value as inf.

mode

can be 'row' or 'entry', meaning filling occurs only when the whole row are NaNs or whenever an element is a NaN, respectively.

Output

fts the forward filled myfints object.

4.3 Factor Backtesting

optimize

Calculates optimal weights for multiple alpha sources (factors) based on their historical performance (in terms of IC).

In the following, we suppose there are K factors and N securities, over a total of T periods. For each factor, we may also include their lagged values in current period optimization. (Current factor value always be included.) We denote by L_k the number of lagged factors involved for factor k, where $k = 1, \dots, K$ indexing factor. So the total number of factors involved in optimization is $K + \sum_{k=1}^{K} L_k$. By optimization, we are trying to obtain optimal weights for each factor (and their lagged ones) over each period. Denote factor weights at period t by

$$\begin{aligned}\mathbf{w}_t' = (&w_{1,t}, w_{1,t-1} \ldots, w_{1,t-L_1}, \\ &\ldots\ldots, \\ &w_{k,t}, w_{k,t-1} \ldots, w_{k,t-L_k}, \\ &\ldots\ldots, \\ &w_{K,t}, w_{K,t-1} \ldots, w_{K,t-L_K}),\end{aligned} \tag{2}$$

where $w_{k,t-\ell_k}, k = 1, \ldots, K, \ell_k = 1, \ldots, L_k$ indicates optimal weights at period t for kth factor at lag ℓ_k. Correspondingly, the factor vector at that period is denoted by

$$\begin{aligned}\mathbf{F}_t = (&F_{1,t}, F_{1,t-1} \ldots, F_{1,t-L_1}, \\ &\ldots\ldots, \\ &F_{k,t}, F_{k,t-1} \ldots, F_{k,t-L_k}, \\ &\ldots\ldots, \\ &F_{K,t}, F_{K,t-1} \ldots, F_{K,t-L_K})'.\end{aligned} \tag{3}$$

Both $\mathbf{w}_t$ and $\mathbf{F}_t$ are $(K + \sum_{k=1}^{K} L_k) \times 1$ vectors.

Finally, for late reference, we define

$$\begin{aligned}\mathbf{C}_t &= \mathrm{Cov}(\mathbf{F}_t), \\ \mathbf{D}_t &= \mathrm{Cov}(\mathbf{F}_t, \mathbf{F}_{t-1}) \equiv \mathrm{E}\left\{[\mathbf{F}_t - E(\mathbf{F}_t)]\,[\mathbf{F}_{t-1} - E(\mathbf{F}_{t-1})]\right\}.\end{aligned}$$

Usage

```
function [W, IR_optm, alpha, IC, exitflag] = ...
         optimize(factors, stockRet, 'option_name', option_value, ...)
```

Input

factors

(a cell vector of `myfints`) Cell vector of factors of size K, each factor (a `myfints`) is a $T \times N$ `myfints` object. All the `myfints` objects in `factors` must be aligned.

stockRet

(`myfints`) Securities returns, $T \times N$. Must be aligned to `myfints` objects in `factors`. [6]

'option_name', option_value, ...

Pairs acting as named arguments providing additional information to the optimization process.

[6] When doing forecasting for next-period, one-period forward returns make sense.

Options	Values	Default	Meanings
`method`	'*Obj*:*Con*'	`'M/V:RHO'`	A string consisting of 2 parts (objective and contraint) delimited by ':'. See explanation below.
`lag`	≥ 0, int	11	Indicates how many lagged factors involved in the optimization. Should be a scalar or a vector of length equal to K (number of factors). If it is a scalar, the same lag will be applied to all factors; if a vector, each element applies to the corresponding factor in `factors`.
`rho`	$[0, 1]$	1.0	Target value of the constaint specified in `method`. Since all available constaints for now are some kind of correlation, this parameter shoud be in range of 0 to 1. See notes below for more details.
`lambda`	≥ 0	0	This parameter appears in objective. if `M/V`, it controls weight mass distribution among weights; if `M-V`, it plays a risk-aversion parameter role.
`ICWin`	≥ 0, int	[36 0]	Specifies the period-window used for calculating mean and std of ICs, consisting of 2 values. The first indicates how many past-period ICs used in optimization. 0 means *expanding*, others the *actual* window size; should > lag+2 (or 0) since we have `lag` lagged factors involved and at the least 2 ICs needed to get a meaningful standard deviation. The second specifies how many forward-period ICs included in calculating IC. If want to include all future periods, set it to `Inf`.
`lb`	≥ 0	0	Lower bound of all weights. Note that the upper bound of all weights is 1.
`IC`	$T \times (K + \sum_{k=1}^{K} L_k)$ `myfints`	empty	IC matrix provided by user. If ommitted or provided as empty (`[]`), `optimize` will calculate IC from `factors` and `stockRet`.

Output

W

(`myfints`) optimal weights, $T \times (K + \sum_{k=1}^{K} L_k)$ where T is number of periods, K the number of factors and L_k (option `lag`) is the number of lags involved for factor k. Each *row* corresponds to a one-time period optimized weight vector, $\mathbf{w}'_t$, given in equation (2).

IR_optm

(`myfints`) IR values when the `W` is applied to the objective function, $T \times 1$.

alpha

(`myfints`) alphas when apply `W` to blend factors, $T \times N$.

IC (`myfints`) Information Coefficients for each factor (include lagged factors) at every time period, $T \times (K + \sum_{k=1}^{K} L_k)$. If user provided an IC via `'IC'` parameter, it will be the same as the user-provided; otherwise, it will be the one `optimize` calculated from `factors` and `stockRet`.

exitflag

(`myfints`) flags for diagnosing optimization process. 1 indicates success, 0 not convergent, negative values indicate problems. See also `fmincon`. $T \times 1$, each element corresponding to one period optimization.

Notes:

1. `optimize` treats `NaN`s in `stockRet` and `factors` as missing values (excluded from optimization).
2. All matrices and vectors returned have the same number of rows (T). The very first few rows (`max(lag)+2`) are `NaN`s (needed for IC calculation).
3. Optimization method can be specified by named parameter **method** which consists of two parts delimited by a ':'. The first part specifies the type of objective, the second for constraint.
   - *Objective* could be one of the following:
     (a) `'EW'`. Simply equally weight factors. No constraint should be specified.
     (b) `'M/V'`. $\max_{\mathbf{w}_t} \mathrm{IR}_t = \dfrac{\mathbf{w}_t' \,\overline{\mathbf{IC}}_t}{\sqrt{\mathbf{w}_t' \,\boldsymbol{\Sigma}_{\mathbf{IC_t}} \,\mathbf{w}_t}} - \lambda \mathbf{w}_t' \mathbf{w}_t.$ [7]
     (c) `'M-V'`. $\max_{\mathbf{w}_t} \mathrm{IR}_t = \mathbf{w}_t' \,\overline{\mathbf{IC}}_t - \lambda \left(\mathbf{w}_t' \,\boldsymbol{\Sigma}_{\mathbf{IC}} \,\mathbf{w}_t\right).$
   - *Constraint* could be one of the following:
     (a) `'RHO'`. $\dfrac{\mathrm{Cov}\left(\mathbf{w}_t'\mathbf{F}_t,\ \mathbf{w}_t'\mathbf{F}_{t-1}\right)}{\mathrm{Var}\left(\mathbf{w}_t'\mathbf{F}_t\right)} = \dfrac{\mathbf{w}_t'\mathbf{D}_t\mathbf{w}_t}{\mathbf{w}_t'\mathbf{C}_t\mathbf{w}_t} \ge \rho.$ [8]
     (b) `'RA'`. $\dfrac{\mathrm{Cov}\left(\mathbf{w}_t'\mathbf{F}_t,\ \mathbf{w}_{t-1}'\mathbf{F}_{t-1}\right)}{\sqrt{\mathrm{Var}\left(\mathbf{w}_t'\mathbf{F}_t\right)\mathrm{Var}\left(\mathbf{w}_{t-1}'\mathbf{F}_{t-1}\right)}} = \dfrac{\mathbf{w}_t'\mathbf{D}_t\mathbf{w}_{t-1}}{\sqrt{\mathbf{w}_t'\mathbf{C}_t\mathbf{w}_t\mathbf{w}_{t-1}'\mathbf{C}_{t-1}\mathbf{w}_{t-1}}} \ge \rho.$ [9]
     (c) `'RFAC'`, $\mathbf{w}_t' \left(\mathrm{Corr}_t^{k,\ell_k}\right)_{k=1,\ldots,K}^{\ell_k=0,\ldots,L_k} \ge \rho$, where $\mathrm{Corr}_t^{k,l_k} = \mathrm{Corr}(\mathbf{F}_t^{k,\ell_k},\ \mathbf{F}_{t-1}^{k,\ell_k})$ and $\mathbf{F}_t^{k,\ell_k}$ is kth factor with lag ℓ_k, $k = 1, \ldots, K$, $\ell_k = 0, \ldots, L_k$. $\ell_k = 0$ means no lagging.
     (d) `'RHO_EQ'`, `'RA_EQ'`, `'RFAC_EQ'` are corresponding equality constraints of above.

   Depending on different factors, different methods may have different performances.

A final note on `optimize`. `optimize` provide mathematical ways to optimally combine factors. It is easy to substitute our intuition for the mathematics, but this can lead to trouble. We should instead verify that the mathematics and our intuition agree: otherwise, either our intuition is wrong or our model is inadequate. If our intuition and the mathematics of a model do not agree, we should seek a reconciliation before proceeding.

[7] Gradient (omitting subscript t): $\nabla \mathrm{IR}(\mathbf{w}) = (\mathbf{w}'\boldsymbol{\Sigma}_{\mathbf{IC}}\mathbf{w})^{-\frac{1}{2}} \left[\overline{\mathbf{IC}}_t - \mathbf{w}'\overline{\mathbf{IC}}_t(\mathbf{w}'\boldsymbol{\Sigma}_{\mathbf{IC}}\mathbf{w})^{-1}\boldsymbol{\Sigma}_{\mathbf{IC}}\mathbf{w}\right].$

[8] Gradient (omitting subscript t): $\nabla\rho(\mathbf{w}) = (\mathbf{w}'\mathbf{C}\mathbf{w})^{-1} \left[(\mathbf{D}+\mathbf{D}') - \frac{\mathbf{w}'\,\mathbf{D}\,\mathbf{w}}{\mathbf{w}'\,\mathbf{C}\,\mathbf{w}}(\mathbf{C}+\mathbf{C}')\right]\mathbf{w}.$

[9] Gradient (omitting subscript t): $\nabla\rho(\mathbf{w}) = (\mathbf{w}_{t-1}'\mathbf{C}_{\mathbf{t-1}}\mathbf{w}_{t-1}\ \mathbf{w}'\mathbf{C}\mathbf{w})^{-\frac{1}{2}} \left[\mathbf{D}\mathbf{w}_{t-1} - \mathbf{w}'\,\mathbf{D}\,\mathbf{w}_{t-1}(\mathbf{w}'\,\mathbf{C}\,\mathbf{w})^{-1}\mathbf{C}\mathbf{w}\right].$

normalize

Normalize factors. Different methods allowed.

Usage

```
score = normalize(factor, 'option_name', option_value, ...)
```

Input

factor

(`myfints`) Factor to be normalized.

'option_name', option_value, ...

(`myfints`) Options providing additional information for normalization.

method

Specifies which method used to do the normalization. Now two methods are allowed:

- `'norminv'`
- `'zscore'`

Default is `'zscore'`.

mode

Indicates if reverse the factor ranking orders. Two possible values:

- `'ascend'` The normalized result has the same ranking as the original.
- `'descend'` The ranking of the normalized result is reversed.

Default is `'ascend'`, which corresponds to 'the-higher-the-better' factors.

weight

(`myfints` or ordinary matrix compatible to the `factor`) Indicates if corresponding elements in `factor` belong to the universe. Weight value of 0 or `NaN` will be excluded the normalization process. Other weight values treat the same. *Default* all element in `factor` will be involved.

GICS

(`myfints` or ordinary matrix compatible to the `factor`) If provided, normalization will also be *neutralized*. The value of `GICS` must be integers.

level Controls to what level in `GICS` the neutralization is performed.

Standard GICS code consists of 8 digits, a form of like `25203010`. The first 2 (`25` in the previous example) refer to **sector**, the next 2 (`20`) to **industry group**, the 5-6th (`30`) to **industry**, and the last 2 (`10`) to **sub-industry**. Correspondingly, the value of level can be: 0: NO neutralization; 1: sector; 2: industry group; 3: industry; or 4: sub-industry. *Default* is 1. Note that it is only effective when `GICS` being provided.

`level` can also be `'customized'`, used with non-standard `GICS` passed by user. In this case, the neutralization will be performed based on all digits of the non-standard gics code.

neutralize

Neutralizes specified function.

Usage

```
res = neutralize(factor, GICS, fun)
res = neutralize(factor, GICS, fun, level)
```

Input

factor

(`myfints`) Factor on which neutralization is performed.

GICS

(`myfints` or ordinary matrix compatible to the `factor`) Neutralization is based on this parameter. If not provided, `neutralize` simply perform `fun` on `factor`. The value of `GICS` must be integers.

fun (function handle) Specifies operation to be neutralized. It should be an ordinary MATLAB function of form:

```
        function out_matrix = fun(input_matrix)
```

where `input_matrix` and `out_matrix` are ordinary matrices with the same dimension. *Whatever it to do, `fun` should ignore `NaN`s.*

level Controls to what level in `GICS` the neutralization is performed.

Standard GICS code consists of 8 digits, a form of like `25203010`. The first 2 (`25` in the previous example) refer to **sector**, the next 2 (`20`) to **industry group**, the 5-6th (`30`) to **industry**, and the last 2 (`10`) to **sub-industry**. Correspondingly, the value of level can be: 0: NO neutralization; 1: sector; 2: industry group; 3: industry; or 4: sub-industry. *Default* is 1. Note that it is only effective when `GICS` being provided.

`level` can also be `'customized'`, used with non-standard `GICS` passed by user. In this case, the neutralization will be performed based on all digits of the non-standard gics code.

Output

res (`myfints`) Result returned.

csregress

Cross-sectional regression of a factor (`yfactor`, regressor) against a group of other factors (`xfactors`, regressands).

Usage

```
[beta, epsilon, std_beta, R2, mse] = csregress(yfactor, xfactors, 'option_name', option_value, ...)
```

Input

yfactor

(`myfints`) $T \times N$, factor used as regressor (dependent variable). T is the number of periods (or time points), N the number of observations at each period.

xfactors

(a cell vector of `myfints`) Contains K `myfints`, each is $T \times N$, factors used as regressands (independent variables)

'option_name', option_value, ...

named arguments providing additional information.

weight

(`myfints` or ordinary matrix compatible to the `yfactor`) $T \times N$. If provided, Weighted Least square will be used to solve the regression problem; otherwise, ordinary least square (weight matrix is identity).

`yfactor` and elements of `xfactors` must be aligned. If `weight` provided, it also must be aligned to `ycode` (in case of `weight` is a `myfints`) or compatible to it (in case of `weight` is a matrix). The regression is performed at each time period as

$$w_i^{\frac{1}{2}} y_i = w_i^{\frac{1}{2}} \sum_{j=1}^{K} \beta_j x_{ij} + \epsilon_i, \quad i = 1, \ldots, N$$

where N is the number of cross-sectional observations (i.e., number of fields in `yfactor`) and each of `xfactors`, K is the number of elements in `xfactors`, $y_i, i = 1, \ldots, N$ is a row from `yfactor`, $w_i, i = 1, \ldots, N$ is a row from `weight`, $x_{i,j}, i = 1, \ldots, N$ is the corresponding row from `xfactors{j}`.

If you want to include a constant term in regression, you should provide it as a `myfints` element in `xfactors`. A simple way to do this is

```
constant_x = yfactor;
constant_x(:,:) = 1;
```

Then you can add this constant `myfints` into `xfactors`.

Output

beta (`myfints`) $T \times K$, corresponding to the estimated coefficients of β.

epsilon

(`myfints`) $T \times N$, regression residual.

std_beta

(`myfints`) $T \times K$, stdandard error of `beta`.

R2 (`myfints`) $T \times 1$, R^2, goodness of fit measure.

mse (`myfints`) $T \times 1$, Mean-Squared-Error of regression, element at time t corresponding to the regression at that time.

A common use of `csregress` is regressing stock returns on a group of factors to get factor returns. In this case, stock return is `yfactor`, every factor is a element (`myfints` object) of `xfactors`.

csqtrtn

Calculate quantile returns based on given signal.

Usage

```
[qtrtn, qtweight] = csqtrtn(signalfts, returnfts, 'option_name', option_value,...)
```

Input

signalfts

(`myfints` object) time series of signal values

reutrnfts

(`myfints` object) time series on which quantile returns are going to be calculated

'option_name', option_value,...

optional arguments. Available options are:

weight

a `myfints` object or an ordinary matrix compatible to `returnfts`, indicates the weights for corresponding elements in `returnfts`, *default* equal weight

univ a `myfints` object or an ordinary matrix compatible to `returnfts`, indicating whether the stock is in the universe at each time point. if specified, the quintile returns will be only calculated for stocks within the universe at each point in time.

GICS

a `myfints` object or an ordinary matrix compatible to `returnfts`, contains GICS code corresponding to elements in `returnfts`. Value of `GICS` must be integer. If provided, the quantile spreads are calculated within each sector identified by `GICS`; if *omitted*, the quantile return are calculated on the whole universe.

qtile Specifies the quantile points. *Default* is quintile (i.e., [0, 0.2, 0.4, 0.6, 0.8, 1])

level Controls to what level in `GICS` the neutralization is performed.

Standard GICS code consists of 8 digits, a form of like `25203010`. The first 2 (`25` in the previous example) refer to **sector**, the next 2 (`20`) to **industry group**, the 5-6th (`30`) to **industry**, and the last 2 (`10`) to **sub-industry**. Correspondingly, the value of level can be: 0: NO neutralization; 1: sector; 2: industry group; 3: industry; or 4: sub-industry. *Default* is 1. Note that it is only effective when `GICS` being provided.

`level` can also be `'customized'`, used with non-standard `GICS` passed by user. In this case, the neutralization will be performed based on all digits of the non-standard gics code.

The quantile returns are calculated as weighted average of part of `returnfts` between certain quantiles.

Output

qtrtn (`myfints` object) time series of quantile returns

qtweight

(cell vector of `myfints` objects) time series of quantile portfolio weights

csqtspread

Calculate quantile spreads based on given signal.

Usage

```
[ofts, weight] = csqtspread(signalfts, returnfts, 'option_name', option_value,...)
```

Input

signalfts

(`myfints` object) time series of signal values

reutrnfts

(`myfints` object) time series on which quantile spreads are going to be calculated

'option_name', option_value,...

optional arguments controlling how the portfolio be constructed. Available options are:

window

an integer, indicates the window size when calculating moving sums, *default* `inf`

weight

a `myfints` object or an ordinary matrix compatible to `returnfts`, indicates the weights for corresponding elements in `returnfts`, *default* equal weight

univ a `myfints` object or an ordinary matrix compatible to `returnfts`, indicating whether the stock is in the universe at each time point. If specified, the quintile spreads will be only calculated for stocks within the universe at each point in time.

GICS

a `myfints` object or an ordinary matrix compatible to `returnfts`, contains GICS code corresponding to elements in `returnfts`. Value of `GICS` must be integer. If provided, the quantile spreads are calculated within each sector identified by GICS; if *omitted*, the quantile spreads are calculated on the whole universe.

level Controls to what level in `GICS` the neutralization is performed.

Standard GICS code consists of 8 digits, a form of like `25203010`. The first 2 (`25` in the previous example) refer to **sector**, the next 2 (`20`) to **industry group**, the 5-6th (`30`) to **industry**, and the last 2 (`10`) to **sub-industry**. Correspondingly, the value of level can be: 0: NO neutralization; 1: sector; 2: industry group; 3: industry; or 4: sub-industry. *Default* is 1. Note that it is only effective when `GICS` being provided.

`level` can also be `'customized'`, used with non-standard `GICS` passed by user. In this case, the neutralization will be performed based on all digits of the non-standard gics code.

long a number between [0,1], indicating long stocks with returns greater than 80% of quantile returns among all stocks. Default is 0.8.

short

a number between [0,1], indicating short stocks with returns less than 20% quantile returns among all stocks. Default is 0.2

The quantile spreads are calculated as weighted average of part of `returnfts` greater than `long` quantile, minus weighted average of part of `returnfts` less than `short` quantile.

Clearly, *by default*, it's quintile spread (Q5−Q1). For *long-only* portfolio, just set 'long' 0 and short 1.

Output

ofts (`myfints` object) time series of quantile spreads and its moving sum

weight

(`myfints` object) time series of weight associated with `returnfts`

tsplot

Visulizes a `myfints` object.

Usage

```
h = tsplot(fts, 'parameter',value,...)
```

Input

fts (`myfints`) Object to be plotted.

'parameter',value,...

pairs acting as named arguments providing additional information for plotting the time series.

group

a matrix, where

- number of columns is equal to the number of time series in `fts` (i.e., number of fields (columns) of `fts` when it's a matrix);
- number of rows indicates how many subplots be drawn in the plot;
- its elements can be 0 (not draw the corresponding series in the subplot), 1 (draw the corresponding series in the corresponding subplot with y-axis on the left), or -1 (draw the corresponding series in the corresponding subplot with y-axis on the right).

For example, for a 3-column `fts` which contains 3 time series, `group` can be

```
[1 0 0; 0 1 0; 0 0 1]
```

which means the plot will have 3 subplots and every plot for a series. Also `group` can be `[1 1 1]` which means drawing all 3 series in one subplot.

Other examples of valid values for `group`:

```
[1 0 1; 0 1 1]                % 2 sub plots where the 3rd series used in both
[1 0 0; 1 0 0; 1 0 0; 1 0 0]  % The same 4 subplots where only 1st series used
```

Default value is row vector with all elements being one and length equal to number of column of `fts`; i.e., all series will be drawn in one plot.

title A string or a cell string vector used as the caption of plots.

- If it is a string, all subplots will use it as caption;
- if it is a vector, the length of it should be equal to the number of groups (implicitly specified by `group`) or 1 (all subplots share the same title as caption.)
- If not provided, no caption will be shown in the plot or subplots.

ylabel

A string or a cell string vector used as the label for y-axis of plots. The number of rows of the cell should be 1 (will be expanded to the number of subplots) or equal to number of subplots implied by `group`; the number of columns of the cell can be 1 (only for left y-axis) or 2 (for left- and right-y axes). Right-y axes only exist when -1 exists in `group`; otherwise, the second column is useless.

ycolor

A cell of 3-element (representing RGB color) vectors like

```
    {[1 0 0] [0 1 0]; [1 0.4 0] [0 0 1]}
```

used as the color for y-axis of plots. The number of rows of the cell should be 1 (will be expanded to the number of subplots) or equal to number of subplots implied by `group`; the number of columns of the cell can be 1 (only for left y-axis) or 2 (for left- and right-y axes). Right-y axes only exist when -1 exists in `group`; otherwise, the second column is useless.

notes

A string or a cell string vector that will be shown just below every subplot as notes. Its format is similar with `title`.

dateformat

A string specifying date format shown along the x-axis. Default is `'yy-mm'`.

drawfun

A cell vector of function handles specifying how data be plotted (such as `@bar`, `@plot`, etc.) It can be any length since it will be used rotatively. *Default* is `{@plot}`, i.e., every series will be drawn as a curve.

style A cell string vector specifying line style to be drawn in plots. Its can be of any length since it will be used rotatively, but need to be properly corresponding to `drawfun`; every element of the vector can be any form the corresponding `drawfun` allowed.

Default value is `{'-b', '-g', '-r', '-k', '-c', '-m', '-y'}`, meaning solid line with different colors and line width is MATLAB's default value. This is appropriate since default `drawfun` is `@plot`.

More sophisticated example is

`{{'-b', 'linewidth', 2}, {'-.', 'Color', [1 0.4 0], 'Marker', '+'}}`.

Note that `[1 0.4 0]` is the ING **orange**. See MATLAB help on specific draw functions about valid properties.

ymax

a scalar or vector specifying maximum value of y-axis. If it is a scalar, every subplot will have the same `ymax`; otherwise its length must be the same as number of subplots implicitly specified by `group`, in this case every element corresponds to a subplot.

If not provided (*default*) or is `NaN`, the function will automatically use maximum value of series used in a subplot as `ymax` for that subplot.

ymin Similar as `ymax`, except it is minimum value of y-axis.

hornlineposn

a vector specifying horizontal lines shown in *every* subplot. So the value of its elements are treated as y-coordinates. The length of the vector is the number of lines to be drawn. These line will shown in red color with width of 1.

vertlineposn

Similar with `hornlineposn` except it is for vertical line. The value of its elements are treated as x-coordinates.

legend

A cell used to control how legends be plotted.

- If it only has one element (i.e., one string), all the subplots will share the same legends.
- Otherwise, its length should be equal to the number of subplots implicitly specified by `group`.

Examples:

`{{'Universe', 'Sector'}}`, will be used in every subplot, or `{'Universe', 'Sector'}`, where `'Universe'` will be used in first subplot, and `'Sector'` in second subplot, suppose only 2 suplots be specified by `group` (otherwise an error occurs).

layout

A 1×2 vector specifying the subplot layout. For example, `[3,2]` means the there are total of 6 (3×2) sub-plots, arranged in 3 rows - 2 columns manner and counted from left to right, then top to bottom. The number of subplot determined by `layout` must be great than or equal to the number of groups implied by `group`. Default is `[<number of group>,1]`, meaning number of subplots equal to number of groups, and every subplot holds a whole line.

range

A cell vector of length equal to number of subplots (implicitly specified by `group`), with each element is a vector or matrix, specifying the position and area the subplot corresponding to group (corresponding to) will hold. Default is `1,2,3,...<number of group>`, meaning the ith group occupies the ith subplot.

For better understanding of `layout` and `range`, refer MATLAB's help on `subplot`.

figure

Specifies a figure handle on which the plots will be ploted. If not specified (which is default), a new figure will be created and used. Specifying an exist figure handle, so we can draw on the same figure with different `tsplot` calls (by specifying the same `layout` and different `range`).

orientation

Can be `portrait` or `landscape`. Default is `portrait`.

Output

Return the figure handle it plotted.

Examples:

In its simplest form, this function can be called as

```
tsplot(fts)
```

Every field in `fts` will be treated as a series and drawn together in one plot using default line style, without caption and y-axis label, and x-axis shown with dates in format of `'yy-mm'`.

```
tsplot(fts, 'group', [1 0 0; 0 1 0, 0 0 1])
```

Suppose `fts` has 3 field (3 series), then this call will create a figure with 3 subplots, every subplot draws a series with default specifications.

```
tsplot(fts, 'group', [1 0 0; 0 1 1], 'style', {'-g', {'.r', 'linewidth', 1}})
```

Suppose `fts` has 3 field (3 series), then this call will create a figure with 2 subplots: the first draws the first column in `fts`, the second draws two line corresponding to the second and third column in `fts`, and in second subplot, the line corresponding to the third column of `fts` will be draw in dotted red line of width 1.

The *font shape*, *font size* and *text color* (and more, such as Greek letters, formulas typesetting) in any text-type options (title, labels, notes, legends, etc.) of plots (or subplots) can also be changed by mixing TEX directives in the string. Suppose you want add some notes in a figure, set note string as

```
note = '\fontsize{12}\bf\color[rgb]{1 .4 0}This is a note example.';
```

This set the note contents `'This is a note example.'`, font size 12, font shape **bold face**, font color orange. For more TEX directives, see MATLAB help on `text properties`. Then you can call plot as normal:

```
tsplot(fts, 'group', [1 0 0; 0 1 0, 0 0 1], 'notes', note);
```

In this example, there will be 3 subplots; given only one note string, the same note will be displayed at the bottom of each subplot. If you want to different notes for different subplots, just pass 3 notes as a cell vector such as

```
tsplot(fts, 'group', [1 0 0; 0 1 0, 0 0 1], 'notes', {'note1', 'note2', 'note3'});
```

mat2fts

Convert data from MATLAB's vectors into `myfints` objects.

Usage

```
fts = mat2fts(date, val, sid)
fts = mat2fts(date, val, sid, fid)
```

Input

date, val
vectors represent dates and corresponding values, receptively.

sid vector represents individual ids (e.g., securities ids).

fid (optional) vector represents group ids (e.g., factor ids). If omitted, all data in `val` belong to the same group.

Note that the sizes of `date`, `val`, `sid` and `fid` must be agree with each other. The combination of `date`, `sid`, `fid` must be unique. The orders of these vectors are irrelevant (ordering will be done inside `mat2fts`).

Output

fts a cell vector of `myfints` object corresponding to each `fid`. The number of `myfints` in `fts` is determined by number of unique values in `fid`, the number of fields in each `myfints` is determined by number of unique values in `sid`, and number of periods (time points) in each myfints is determined by number of unique values in date. You can access the fid for each returned myfints in fts by fts{i}.desc.

xls2fts

Convert data from Microsoft Excel sheet into myfints objects.

Usage

```
[fts,...] = xls2fts(fileName, sheetName, dateColRange, sidColRange, valColsRange)
```

Input

fileName

Name of an Excel document file from which data is read.

sheetName

Sheet name inside the fileName excel file.

dateColRange

a string, specify a range where data will be used as date. Range should be in vecor form; i.e., either one row or one column, like B2:B10000.

sidColRange

a string, range where data be used as security id. Range should be in vecor form; i.e., either one row or one column, like C2:C10000.

valColsRange

a string or cell vector of string, specifies data used as values. Every string corresponds to a final myfints returned.

Note all the range spcified should be agree with each other in dimension. The combination of content from dateColRange and sidColRange must be unique. The occurance order is irrelevant.

Output

fts, ...

one or more myfints objects corresponding to each element in valColsRanges. The number of fields in each myfints is determined by number of unique values in sidColRange, and number of periods (time points) in each myfints is determined by number of unique values in dateColRange. You can get the corresponding valColsRange string for each returned myfints in fts by fts{i}.desc.

4.4 Comparison Operators: >=, >, ==, ∼=, <=, <

These operators perform element-by-element comparison between two `myfints` objects or one `myfints` object and an ordinary MATLAB matrix (or scalar). You can compare two `myfints` objects like: [10]

```
        result= obja > objb;
```

The `result` is of same type as of `obja`. You cOan also compare a `myfints` object with an ordinary matrix or scalar, as long as they are compatible in term of matrix dimension, like this

```
        result = 5 > obja;
```

or

```
        result = obja < 5;
```

In this case, `result` is of the same type as `obja`. The general rule for type of returned result is the `result` has the same type as of the most left of `myfints` objects.

For compare if two myfints object equal as a whole (return a single true or false), you can use `isequal(ftsa, ftsb)` function, just as for normal MATLAB matrix.

4.5 Ordinary Matrix Algebra Operators: +, -, *, /, .*, ./

For `myfints`, `*` and `/` are identical to `.*` and `./`, respectively. All other operators have the same meanings as those for MATLAB matrices. All these operators can be used with mixing a `myfints` object and ordinary matrix in any order.

4.6 Indexing

Not only does `myfints` allow the same indexing methods supported by MATLAB's `fints`, it also support MATLAB's matrix indexing methods. The following examples summarize various indexing forms possible.

- index rows (values of all fieldnames on the same date):

```
    a = fts('01-Apr-2010');
    a = fts('01-Apr-2010',:);  % equivalent to the line above
    a = fts('01-Apr-2010::01-Apr-2011');
    a = fts({'01-Apr-2010', '01-May-2010', '01-Oct-2010'},:); % must be with ,:
    % Below union of dates specified used as index (sequence irrelevent)
    a = fts({'2011-01-01::2011-05-31', '2011-03-21', '2009-12:21-2011-01-02'}, :);
```

[10]Note that all the factors are derived from `myfints`, so these operators also apply to factors. For a complete methods of a class, type `methods(obj)` or `methods(class name)` from MATLAB command window.

```
a = fts({734100 734150 734200}, :); % numeric in cell used as dates
% Below combination of string & numeric type indexing
a = fts({'2011-01-01::2011-05-31', num2cell(now-5:now), '2011-03-21'}, :);
a = fts(5);          % numeric in ordinary vector used as positioning index
a = fts(5,:);        % equivalent to the line above
a = fts(5:7);
a = fts(5:7,:);      % equivalent to the line above
a = fts([5 7]);
a = fts([5 7],:);    % equivalent to the line above
a = fts(logical([1 0 0 1]),:);  % equivalent to fts([1 4],:); logical vector used as TF index
```

- index columns (values of fields on all dates):

```
a = fts.fieldname;
a = fts(:,'fieldname');
a = fts(:,{'fieldname1', 'fieldname2'});
a = fts(:,2);
a = fts(:,[2:10]);
```

- index columns and rows

```
a = fts.fieldname('01-Apr-2010');
a = fts.fieldname(3:5);
a = fts.fieldname(4);
a = fts('01-Apr-2010','fieldname3');
a = fts(3, 'fieldname2');
a = fts({'01-Apr-2010', '01-Apr-2011'}, {'IBM', 'C', 'APPL'});
a = fts({'01-Apr-2010', '01-Apr-2011'}, 1:30);
a = fts('01-Apr-2010::01-Apr-2011', 1:30);
a = fts(2:5, 3:8);
a = fts(fts > 0.5);          % return a vector
a = fts(fts == anotherfts); % must be aligned
a = fts(matrix);            % must compatible in size and contents
a = fts(anotherfts);        % must be aligned and compatible in size and contents
```

- index the whole thing

```
fts(:,:);
fts('::');
```

- modify `myfints` object by indexing

```
% Remove some dates
fts(3:5,:) = [];
fts({'01-Apr-2010', '01-Apr-2011'},:) = [];
% Remove some fields
fts.IBM = [];
fts(:, 'IBM') = [];
fts(:, {'MSFT', 'GOOD'}) = [];
fts(:, '0059AAPL') = [];
fts.('0059AAPL') = [];    % NOTICE this usage when field is not valid matlab identifier
fts(:, 3:5) = [];
```

Removing this way must be about entire rows or columns, otherwise MATLAB will complain. Assigning to irregular part of a `myfints` object with compatible data is possible, like

```
    fts(fts > 0.5) = 0;
    fts(fts > anotherfts) = -fts(fts > anotherfts); % flip signs
```

The returned from indexing a `myfints` object usually still is a `myfints` object with appropriate dates and fieldnames labeled. However, for indexing like `fts>0.5` the returned is ordinary MATLAB vector since it is irregular. In fact, this kinds of index mostly used in this way: `fts(fts>0.5)=100;`

`end` can be appeared whenever it is appropriate; i.e., when idexing with numeric values, like

```
    a = obja(3:end);
```

return the 3rd to last rows of `obja` and return a object (`a`) of the same type as of `obja`.

The principle is: when the first dimension is a cell vector or char-string, it is treated as dates, no matter what's inside (i.e., date strings or numeric dates). If the first dimension index is a numeric/logical vector, it is treated as position/true-false index.

A key difference between first (date)-dimension and second (field)-dimension index: no matter what order you specified for dates, dates will always be ordered ascendingly; while filed sequence will always be the same as you specified in index.

4.7 Cross-Sectional Statistics

cscorr, csrankcorr, cscorrwt

Calculate the cross sectional correlation time series of two aligned `myfints`.

Usage

```
  ofts = cscorr(iftsA, iftsB, 'argname', value,...)
  ofts = csrankcorr(iftsA, iftsB)
  ofts = cscorrwt(iftsA, iftsB, iftsWt)
```

Input

iftsA (`myfints` object) One inputs those cross sectional correlation is to be found

iftsB (`myfints` object) One inputs those cross sectional correlation is to be found

'argname', value,...

specifies one or more optional name/value pairs. Specify name inside single quotes. The following table lists valid parameters and their values.

Parameter	Values
type	• `'Pearson'`, (the default) computes Pearson's linear correlation coefficient • `'Kendall'`, computes Kendall's τ • `'Spearman'`, computes Spearman's ρ • a matrix or `myfints` object serveing as weights, calculating weighted correlation. if matrix, must be compatible with size; if `myfints`, must be aligned with `iftsA` and `iftsB`. Parameter `rows` is not applicable for weighted correlation (if provided, ignored.)
rows	• `'all'` (the default) uses all rows regardless of missing values (`NaN`s) • `'complete'` uses only rows with no missing values • `'pairwise'` computes $\rho(i, j)$ using rows with no missing values in column i or j

`csrankcorr` actually is equivelent to

```
ofts = cscorr(iftsA, iftsB, 'type','spearman','rows','complete');
```

and `cscorrwt` is equivelent to [11]

```
ofts = cscorr(iftsA, iftsB, 'type', iftsWt);
```

Both of them are shortcut forms of `cscorr`.

Output

ofts a `myfints` object of cross section correlation.

cscov

Corss-sectional covarance.

Usage

```
ofts=cscov(iftsA, iftsB)               % NaN will be counted
ofts=cscov(iftsA, iftsB, 'ignorenan')  % NaN will be excluded
```

csmax, csmean, csmedian, csstd, cssum, csnorm

Cross-sectional statistics for a `myfints` object.

[11] Weighted mean is $m[x; w] = (\sum_i w_i * x_i)/(\sum_i w_i)$, weighted covariance $c[x, y; w] = (\sum_i w_i * (x_i - m[x; w]) * (y_i - m[y; w]))/(\sum_i w_i)$, and then weighted correlation $r[x, y; w] = c[x, y; w]/\sqrt{c[x, x; w] * c[y, y; w]}$.

Usage

```
ofts = csmax(fts)
ofts = csmean(fts)
ofts = cssum(fts)
ofts = csmedian(fts)
ofts = csstd(fts)
ofts = csnorm(fts)
```

Note all these functions ignore `NaN` values.

What if you want to a statistics not listed above? Using `uniftsfun`. For example, suppose you want to calculate cross-sectional kurtosis for a `myfints`, simply do this

```
ofts = uniftsfun(fts, @(x) kurtosis(x,1,2))
```

See also MATLAB's help on `kurtosis` for meanings of other two parameters.

ftsmovavg, ftsmovstd, ftsmovsum

Calculate the moving average/standard deviation/summation of a time series.

Usage

```
newFts = ftsmovavg(oldFts, window, ignoreNaN)
newFts = ftsmovstd(oldFts, window, ignoreNaN)
newFts = ftsmovsum(oldFts, window, ignoreNaN)
```

Where

`window` is the number of rows included in every calculation from current time point backward to previous time point (including current time point). `NaN`s will be ignored in calculation if `ignoreNaN = 1`.

If you want to perform functions other than average/standard deviation/summation, use `ftsmovfun`.

4.8 Third-Dimensional Statistics

ftsnanmean, ftsnanstd, ftsnansum

Usage

```
ofts = ftsnanmean(ifts1, ifts2, ..., iftsN)
ofts = ftsnanstd(ifts1, ifts2, ..., iftsN)
ofts = ftsnansum(ifts1, ifts2, ..., iftsN)
```

Input

ifts1, ifts2, ..., iftsN

myfints objects on which specific statistics will be performed. They must be aligned.

Output

ofts the result of type myfints, compatible with ifts1,. . . ,iftsN, etc.

4.9 Performing Specified Functions on myfints

uniftsfun

Core function performing actions when only one myfints object involved. Acutally the csmean, csmax, etc. just a direct call to this function.

Usage

```
ofts = uniftsfun(ifts, fun_handle)
ofts = uniftsfun(ifts, fun_handle, fieldname)
```

Where

ifts myfints object to be operated on

fun_handle

operation to be performed on ifts. fun_handle should be a function of

```
function out_matrix = fun(in_matrix)
```

where the dimension of out_matrix can be different from that of in_matrix; see explanation for fieldname below in this case.

fieldname

Specifies what field names should be in the returned myfints object ofts. fieldname should be a char vector (representing a string) or a cell vector of strings. The number of field names provided should be 1 (a char vector or a one-element cell) or matching the result returned by fun_handle.

In case of one field name provided and returning multiple fields (columns) by fun_handle, the field name will be appended by a counter. If number of fields names beyond 1 and not matching the result of fun_handle (not equal the number of columns returned by fun_handle), an error will occur and break the program execution.

By *DEFAULT*, the fieldname in ifts will be used. This is fine if input and output have the same columns. Otherwise, user should provide an appropriate fieldname.

For most cross-sectional operation, user should provide a fieldname since the returned usually has less than the original number of columns (mostly only one column, such as mean, sum, etc. on column dimension) For most time-series operation, user don't need to provide `fieldname` since most of these operation returned something having the same number of columns as operand (still, consider `mean`, `sum` on row dimension).

Generally speaking, operation along a specified dimension usually removes that dimension from the result. For instance, `mean(A,1)` will remove row(1st) dimension, and `mean(A,2)` remove the column(2nd) dimension.

ofts result returned. Usually, it's still a `myfints` object having the similar structure as `ifts` (see explanation about `fieldname` above). HOWEVER, in case of *the result returned by `fun_handle` has different number of rows as that of inputted `ifts`*,

- if the returned row number is 1, `ofts` uses last date in `ifts` as its date;
- otherwise, return directly the result (type of matrix) instead of `myfints` object.

Examples:

```
    function ofts = csmedian(fts)
        ofts = uniftsfun(fts, @(x)nanmedian(x,2), 'csmedian');
    end

    function ofts = cssum(fts)
        ofts = uniftsfun(fts, @(x)nansum(x,2), 'cssum');
    end
```

biftsfun

Core function performing actions on at least two `myfints` objects. [12]

Usage

```
ofts = biftsfun(lhs, rhs, fun_handle)
ofts = biftsfun(lhs, rhs, fun_handle, fieldname)
```

Where

lhs, rhs

operands (left and right in case of `fun_handle` being bi-operand operator) on which `fun_handle` to be performed. One of them surely is `myfints` object (otherwise MATLAB WON'T call this function). The other can be anything `fun_handle` allowed (e.g., a scalar, matrix, or `myfints` object, etc.) `lhs` and `rhs` must be compatible in the sense of:

- if both are `myfints` objects, `iscompatible` returns `true`
- if only one of them is of type `myfints`, the matrix form of them are allowed by `fun_handle`

[12] More `myfints` objects possible, all depends on `fun_handle`. `csqtspread` is such an example where at most 4 `myfints` object involved: one for signal, one for return, one for GICS, one for weights.

fun_handle

operation to be performed on `lhs` and `rhs`. The function pointed to by `fun_handle` should be of form

```
function out_matrix = fun(in_matrixA, in_matrixB)
```

where `in_matrixA` and `in_matrixB` have the same dimension, and `out_matrix` can be different of dimension from them. See explanation for `fieldname` below.

fieldname

Specifies what field names should be in the returned `myfints` object `ofts`. `fieldname` should be a char vector (representing a string) or a cell vector of strings. The number of field names provided should be 1 (a char vector or a one-element cell) or matching the result returned by `fun_handle`.

In case of one field name provided and returning multiple fields (columns) by `fun_handle`, the field name will be appended by a counter. If number of fields names beyond 1 and not matching the result of `fun_handle` (not equal the number of columns returned by `fun_handle`), an error will occur and break the program execution.

By *DEFAULT*, the `fieldname` in `ifts` will be used. This is fine if input and output have the same columns. Otherwise, user should provide an appropriate `fieldname`.

ofts result returned. Usually, it's still a `myfints` object having the similar structure as `ifts` (see explanation about `fieldname` above). HOWEVER, in case of *the result returned by `fun_handle` has different number of rows as that of inputted `ifts`*,

- if the returned row number is 1, `ofts` uses last date in `ifts` as its date;
- otherwise, return directly the result (type of matrix) instead of `myfints` object.

multiftsfun

Perform an element-by-element operation on multiple `myfints` objects.

Usage

```
ofts = multiftsfun(ifts1, ifts2, ..., iftsN, fun_handle)
```

Input

ifts1, ifts2, ..., iftsN

`myfints` objects on which operation specified by `fun_handle` will be performed. They must be aligned. In the following, we suppose each of them is of size $T \times N$.

fun_handle

actions to be performed on these `myfints`. The function pointed by `fun_handle` should be of from

```
function out_2D_matrix = fun(in_3D_array)
```

where `in_3D_array` is of size $T \times N \times K$ and K corresponds to the number of `myfints` inputted as parameters of `multiftsfun`, and `out_2D_matrix` is of $T \times N$. For example, if you want to perform a element-by-element summation among a group of `myfints`, you may set `fun_handle` as

```
fun_handle = @(x) sum(x,3)
```

and keep in mind that `x` is a 3D matrix.

Output

ofts the result of type `myfints`, compatible with `ifts1`,...,`iftsN`, etc.

Example:

```
function sfts = ftsnansum(varargin)
    sfts = multiftsfun(varargin{:}, @(x)nansum(x,3));
end
```

ftsmovfun

Calculate the moving statistics (e.g., moving average) of a time series.

Usage

```
ofts = ftsmovfun(ifts, window, fun_handle)
ofts = ftsmovfun(ifts, window, fun_handle, fieldname)
```

Input

ifts (`myfints` object)

window
size of moving window. If windows is `inf`, then it's the expanding window.

fun_handle
operation to be performed in moving manner. It should be like

```
function out_matrix = fun(in_matrix)
```

and keeping in mind that `in_matrix` is the current moving part matrix. The number of columns of returned `out_matrix` can be different from that of `in_matrix`. The row numbers, however, must be 1 (otherwise, it'll be confused where's the moving comes from). See explanation for `fieldname` below.

fieldname
Specifies what field names should be in the returned `myfints` object `ofts`. `fieldname` should be a char vector (representing a string) or a cell vector of strings. The number of field names provided should be 1 (a char vector or a one-element cell) or matching the result returned by `fun_handle`.

In case of one field name provided and returning multiple fields (columns) by `fun_handle`, the field name will be appended by a counter. If number of fields names beyond 1 and not matching the result of `fun_handle` (not equal the number of columns returned by `fun_handle`), an error will occur and break the program execution.

By *DEFAULT*, the `fieldname` in `ifts` will be used. This is fine if input and output have the same columns. Otherwise, user should provide an appropriate `fieldname`.

Output

ofts the resulting `myfints` object.

Examples:

```
function newFts = ftsmovsum(oldFts, window, ignoreNaN)
if nargin < 3
    ignoreNaN = 1;
end

if ignoreNaN
    fun = @(x) nansum(x,1);
else
    fun = @(x) sum(x,1);
end

newFts = ftsmovfun(oldFts, window, fun);
```

In `ftsmovsum`, if `window == inf`, then it actually calculates cumulated sum.

bsxfun

Apply element-by-element binary operation to two `myfints` or a `myfints` and an array with singleton expansion enabled.

Usage

```
  C = bsxfun(fun,A,B)
```

Where

A,B operands. at the least one of them is a `myfints` object (otherwise MATLAB won't be here); the other can be an ordinary MATLAB matrix or `myfints` object.

fun a function handle, and can either be an M-file function or one of the following built-in functions:

Function	**Operation**
`@plus`	Plus
`@minus`	Minus
`@times`	Array multiply
`@rdivide`	Right array divide
`@ldivide`	Left array divide
`@power`	Array power
`@max`	Binary maximum
`@min`	Binary minimum
`@rem`	Remainder after division
`@mod`	Modulus after division
`@atan2`	Four quadrant inverse tangent
`@hypot`	Square root of sum of squares
`@eq`	Equal
`@ne`	Not equal
`@lt`	Less than
`@le`	Less than or equal to
`@gt`	Greater than
`@ge`	Greater than or equal to
`@and`	Element-wise logical AND
`@or`	Element-wise logical OR
`@xor`	Logical exclusive OR

If an `M`-file function is specified, it must be able to accept either two column vectors of the same size, or one column vector and one scalar, and return as output a column vector of the size as the input values.

`bsxfun` applies an element-by-element binary operation to `A` and `B`, with singleton expansion enabled, The operation performed *irrespective of the compatibility of `A` and `B`* when both of them are `myfints` objects, so long as their dimensions agree. By "agree", we mean each dimension of `A` and `B` must either be equal to each other, or equal to 1. Whenever a dimension of `A` or `B` is singleton (equal to 1), the array is virtually replicated along the dimension to match the other array. The array may be diminished if the corresponding dimension of the other array is 0.

The result returned usually still is a `myfints` object, with dates and field names match that of first suitable in `A` and `B` (in order of occurrence in argument list). If no matched dates found with the result, the result returned in matrix form. It will be an error no matched field names found with the result. (However, if the number of field names found is 1, the name will be appended with a counter up to the number of columns of result as the result's field name).

4.10 Field Operations

This section lists function related to manipulating `myfints`' fields.

```
fnames = fieldnames(fts, 1)          % return field names
fnames = fieldnames(fts, 0)          % return field names, incl. freq, dates, desc

ret = getfield(fts, fids)            % => fts(:,flds)
ret = getfield(fts, fids, dates)     % => fts(dates,flds)

fts = setfield(fts,flds,val)         % => fts(:,flds) = val
fts = setfield(fts,flds,dates,val)   % => fts(dates, flds) = val

fts = extfield(fts, flds)            % => fts = fts(:,flds);
fts = rmfield(fts, rmflds)           % => fts = fts(:, setdiff(fieldnames(fts,1),rmflds))

fts = chfield(fts,oldnames,newnames) % change field names; newnames 1-to-1 oldnames

isfld = isfield(fts, fldname)        % check if fldname is a field of fts

fts = padfield(fts, flds)            % NOT => fts = fts(:,flds), padding new fields with NaNs
fts = padfield(fts, flds, padStuff)  % NOT => fts = fts(:,flds), padding new fields with padStuff
```

Note that `fts = padfield(fts, flds, padStuff)` is not equivalent to `fts = fts(:,flds)`; it allows flds not be existing fields of `fts`, in which case new fields will be added to fts with values being `padStuff` which by default is `NaN`.

4.11 Resampling

convertto, toannual, tosemi, toquarterly, tomonthly, toweekly, todaily

```
ofts = convertto(ifts, freq, 'argname', argvalue, ...)
 ofts = toannual(ifts, 'argname', argvalue, ...)
 ofts = tosemi(ifts, 'argname', argvalue, ...)
 ofts = toquarterly(ifts, 'argname', argvalue, ...)
 ofts = tomonthly(ifts, 'argname', argvalue, ...)
 ofts = toweekly(ifts, 'argname', argvalue, ...)
 ofts = todaily(ifts, 'argname', argvalue, ...)
```

Where

`argname` and `argvalue` can be:

Parameter	Default	Meaning	Freqs Appl.
`'EOW'`	0	0 - 6, Specifies the end-of-week day: 0–Friday, 1–Saturday, 2–Sunday, 3–Monday, 4–Tuesday, 5–Wednesday, 6–Thursday	weekly(2)
`'ED'`	0	The end of date (of month) in a period. 0 is the last day (or last business day), 1-31 is the specific day (if beyond end of month, adjust to end of month)	monthly(3), quarterly(4), semiannual(5), annual(6)
`'EM'`	3	Last month of the first period (e.g., quarter, year). All subsequent quarterly dates are based on this month	quarterly(4), semiannual(5), annual(6)
`'Busdays'`	0	Either 0 or 1, indicating whether business days (non-weekend and non-holidays) are counted. If a sampling day is fallen in a non-business day, it will be adjusted to the next business day, unless there's no business day in the current frequency period (e.g., current month), in which case prior business day will be used.	All frequencies
`'Holidays'`	NYSE	A vector specifying holidays	All frequencies
`'Weekend'`	[1 0 0 0 0 0 1]	Specify which day is weekend in order of [Sun Mon Tue Wed Thu Fri Sat]	All frequencies

There is an additional named argument. Following a char-type string indicator parameter `'CalcMethod'` (case-insensitive), specifies which method is used to calculate resampling sample data. Methods could be one of

- `'CumSum'` Returns the cumulative sum of the values between each sampling period (e.g., week, month, etc.) Data for missing dates are given the value 0.

- `'Exact'` Returns the exact value at the end-of-sampling-period date. No data manipulation occurs.

- `'Nearest'` (*default*) Returns the values located at the end-of-sampling-period date. If there is missing data, Nearest returns the nearest data point preceding the end-of-sampling-period date.

- `'SimpAvg'` Returns an averaged value over each sampling period that only takes into account dates with data (non NaN) within each quarter.

4.12 List of Other Functions

Some are new, most have the same signatures as those of MATLAB's financial toolbox.

- `not(~)` and `power(.^)`
- `isnan, isinf`
- `log, log10, log2`
- `exp`
- `abs, diff, cumsum`
  All function above (from `not`) return another `myfints` object.
- `max, min, mean, std`
  Functions below return a structure, one evidence of inconsistence in MATLAB's fints.
- `tsmovavg, macd`
  These two functions also implemented on ordinary matrix.
- `var, cov, corrcoef`
  These three functions return a ordinary matrix.
- `nanmax, nanmin, nanmean, nanmedian, nansum, nanstd, nanvar, nancov`
  All these nan-functions return a normal matrix, inconsistent with their none-nan counterparts which many return a weird structure. Note their matrix version implemented in MATLAB's stat toolbox.
- `plot` Now directly call `tsplot`.
- `iscompatible, isempty, isequal, issorted, size, length, fts2mat`
- Lag and Lead Operators the same as before:

```
fts = lagts(fts, nperiod, padmode)
fts = leadts(fts, nperiod, padmode)
```

- Concatenating Operators:

```
fts = horzcat(fts1, fts2, ..., ftsN)   % => [fts1, fts2, ..., ftsN]
fts = vertcat(fts1, fts2, ..., ftsN)   % => [fts1; fts2; ...; ftsN]
```

  `vertcat` has been enhanced to allow date overlapping in `myfints` objects to be mergerd, in which case the first occurance data used in the result.

5 Mutliple Dimensional Time Series: xts

Matrix of dimensions beyond 2 in Matlab is a natural extension of 2-D matrix. So is out multiple dimension financial times series class: `xts`. The usage is consistent with Matlab's matrix operations and common sense.

5.1 Common Functions with slightly different usage

For easy comparation, we list them in tabel 6.

5.2 Common Functions

Function which have the same usage as myfints are listed in table 7.

Table 6: Comparison between `xts` and `myfints`

method name	**myfints**	**xts**	**notes**

`constructor`	fts = myfints
	fts = myfints(dates, data)
	fts = myfints(dates, data, fieldnames)
	fts = myfints(dates, data, fieldnames, freq)
	fts = myfints(dates, data, fieldnames, freq, desc)
	fts = myfints(dates, data, fieldnames, freq, desc, unit)
		xts = xts
		xts = xts(dates, data)
		xts = xts(dates, data, fieldnames1, ..., fieldnamesN)
		xts = xts(dates, data, fieldnames1, ..., fieldnamesN, freq)
		xts = xts(dates, data, fieldnames1, ..., fieldnamesN, freq, desc)
		xts = xts(dates, data, fieldnames1, ..., fieldnamesN, freq, desc, unit)
			N is called field dimension. `xts` is a $N+1$ dimension `xts`.

`alignfields`	[ofts1, ofts2, ...] = aligndata(ifts1, ifts2, ...) [ofts1, ofts2, ...] = aligndata(ifts1, ifts2, ..., mode)	[oxts1, oxts2, ...] = aligndata(ixts1, ixts2, ...) [oxts1, oxts2, ...] = aligndata(ixts1, ixts2, ..., mode) [oxts1, oxts2, ...] = aligndata(ixts1, ixts2, ..., flddim) [oxts1, oxts2, ...] = aligndata(ixts1, ixts2, ..., mode, flddim) [oxts1, oxts2, ...] = aligndata(ixts1, ixts2, ..., flddim, mode)	Aligning specified field dimension. if `flddim` == 0 (default), all field dimensions will be aligned. Input arg `xts` objects can be different dimensional so long as they all have the specified field dimension.
`aligndata`	[ofts1, ofts2, ...] = aligndata(ifts1, ifts2, ...) [ofts1, ofts2, ...] = aligndata(ifts1, ifts2, ..., mode) [ofts1, ofts2, ...] = aligndata(ifts1, ifts2, ..., aligningdates) [ofts1, ofts2, ...] = aligndata(ifts1, ifts2, ..., mode, aligningdates) [ofts1, ofts2, ...] = aligndata(ifts1, ifts2, ..., aligningdates, mode) [ofts1, ofts2, ...] = aligndata(..., 'CalcMethod', calcMethod)		Align data along all dimensions (time and all field dimensions). So all `xts` objects must have the same dimension.

`isaligneddata`	tf = isaligneddata(fts1, ..., ftsN)	tf = isaligneddata(xts1, ..., xtsN)
		tf = isaligneddata(xts1, ..., xtsN, flddim)
			`flddim` by default is 0, means check all field dimensions as well as time dim.

`isalignedfields`	tf = isalignedfields(fts1, ..., ftsN)	tf = isalignedfields(xts1, ..., xtsN)
		tf = isalignedfields(xts1, ..., xtsN, flddim)
			`flddim` is 0 by default.

`fieldnames`	fn = fieldnames(fts)
	fn = fieldnames(fts, 1)
		fn = fieldnames(xts, 0, flddim)
		fn = fieldnames(xts, 1, flddim)
			`flddim` is 1 by default.

method name	**myfints**	**xts**	**notes**
`getfield`	fval = getfield(fts, fields) (fields can be multiple) fval = getfield(fts, fields, dates) (fields can be multiple)	fval = getfield(xts, field) (field must be char) fval = getfield(xts, 'dates', dateidx) (equiv. xts.dates(idx))	Unless accessing `xts.freq`, `xts.desc`, `xts.unit`, or `xts.dates`, you are recommended to use `xts(timeidx, fidx1, ..., fidxN)` to get part of `xts`.

`setfield`	fts = setfield(fts, fields, value)
	fts = setfield(fts, fields, dates, value)
		xts = setfield(xts, field, value)
		xts = setfield(xts, 'dates', dateidx, value)
			Similiar to `getfield` except a value need to be provided.

`extfield`	fts = extfield(fts, fldnames)	xts = extfield(xts, fldnames, flddim = 1)	
`rmfield`	fts = rmfield(fts, fldnames)	xts = rmfield(xts, fldnames, flddim = 1)	
`padfield`	fts = padfield(fts, fldnames, padstuff = NaN)	xts = rmfield(xts, fldnames, padstudd = NaN, flddim = 1)	
`chfield`	fts = chfield(fts, oldfldnames, newfldnames)	xts = chfield(xts, oldfldnames, newfldnames, flddim = 1)	
`isfield`	tf = isfield(fts, fldname)	tf = isfield(xts, fldname, flddim = 1)	

`fts2mat`	mat = fts2mat(fts)
	mat = fts2mat(fts, datesflag)
	mat = fts2mat(fts, fldnames)
	mat = fts2mat(fts, datesflag, fldnames)
		mat = fts2mat(xts)	`xts` only allow obtaining whole data, excluding dates.

`cat`	fts = cat(1, fts1, ..., ftsN) or fts = [fts1, ..., ftsN]
	fts = cat(2, fts1, ..., ftsN) or fts = [fts1; ...; ftsN]
		xts = cat(1, xts1, ..., xtsN) or xts = [xts1, ..., xtsN]
		xts = cat(2, xts1, ..., xtsN) or xts = [xts1; ...; xtsN]
		xts = cat(dim, xts1, ..., xtsN)
			Vertical cat (dim==1, or in form of [fts1;...; ftsN]) will order data by dates in final result.

`uniftsfunbiftsfun`	xts = uniftsfun(xts, fun, fields, dates) xts = biftsfun(lxts, rxts, fields, dates)	lxts and rxts must be aligned in all dimensions. Now fields can be a cell vector with each element is a cell of strings representing corresponding field dimension's fieldnames. If you want use the original xts' fieldnames, leave corresponding posn an empty thing (of course, this dimension in result should be matched to that in original xts arguments.	

Table 7: Same usage functions

method name	usage	notes
aligndates	[ofts1, ofts2, ...] = aligndates(ifts1, ifts2, ...) [ofts1, ofts2, ...] = aligndates(ifts1, ifts2, ..., aligningdates) [ofts1, ofts2, ...] = aligndates(..., 'CalcMethod', calcMethod)	Resampling along the time dimension. `'CalcMethod'` can be `'cumsum'`, `'exact'`, `'nearest'`(default), and `'simavg'`. Input arguments of `xts` can be different in number of dimensions.
isaligneddates	tf = isaligneddates(fts1, ..., ftsN)	
isempty	tf = isempty(xts)	
isequal	tf = isequal(fts1, ..., ftsN)	

size	sz = size(xts)
	sz = size(xts, dim)
	[sz1, sz2,...,szN] = size(xts)	

length	len = length(xts)	Equivalent to size(xts,1)

lagtsleadts	xts = lagts(xts, nperiod, padmode = 0)
	xts = leadts(fts, nperiod, padmode = 0)
		padmode default value takes 0 for compatible with MATLAB fints.

Relation Ops	`>=, >, <, <=, ==,` ∼=, *sim*	One operand should be xts, the other either be a scalar, dimensional-match matrix, or aligned xts
uniary Ops	-xts, +xts	−: flip sign of xts, + has no effect
Binary Ops	`+, -, *, /, .*, ./, ^`	Note that `*` and `/` are equivalent to `.*` and `./`
nanmax, nanmin, nanmean, nanmedian, nansum, nanstd, nanvar, var		Same as MATLAB's corresponding functions; all return ordinary matrix. For Compatible with MATLAB's fints
isnan, isinf, exp, log, log10, log2, abs		Element-level ops; all return another xts
cumsum, diff	oxts = cumsum(ixts), oxts = diff(ixts)	Ops along time dimension. Return another xts. Returned from diff has one less row
convertto, toannual, todaily, tomonthly, toquarterly, tosemi, toweekly		Resampling along time dimension

backfillforwardfill	backfill(o, lookbackperiod, mode)
	forwardfill(o, lookfwdperiod, mode)
		Now when mode is 'entry', it fill along time dimension for all other dimensions. If it is 'row', filling will use data without NaNs in all dimensions.

multiftsfun	oxts = multiftsfun(xts1, ..., xtsN, fun)	Perform fun(xts1,...,xtsN); all xts must be aligned in all dimensions

method name	usage	notes
bsxfun	oxts = bxsfun(fun, lxts, rxts)	Same usage as for 2-D. Note now singleton extension expanded to all dimensions. First fit (in terms of dimension) field used in result. See 4.9.

5.3 Functions only applicable for myfints

- `nancov corrcoef cov`
- `max min mean std`: It's wield that these 4 return a structure in MATLAB's fints
- All `csXXXX` functions
- All `ftsmovXXX` functions
- `ftsnanmean, ftsnanstd, ftsnansum, ftswgtmean`
- All `XXXXif` function which we don't recommend to use even for `myfints`
- Backtest functions: `neutralize, normalize, optimize, tsplot, plot`
- *Depreciated* functions: `datemismatch, fieldmismatch, isidenticaldates, isidenticalfields`
- Functions for compatibility with MATLAB's fints: `tsmovavg, macd`

5.4 Indexing

Suppose we have a 4-D xts: first dimension is time, second for security ids, third for factors, and forth for long or short. Further suppose size of time dim is 10, size of security dim is 10, size of factor dimension is 5, for factors 'fac1', 'fac2', 'fac3', 'fac4', 'fac5', and size of forth dimension is 2 for 'short', 'long'. Our 4-D xts variable is x with size of $10 \times 10 \times 5 \times 2$. Following examples show you how to indexing an xts.

```
a = x(:,:,:, 'short');              % return a 3-D xts since last dimension is singleton.
a = x(:,:,{'fac1','fac5'}, :);      % return a 4-D xts: 10x10x2x2
a = x(:,:,'fac2', {'long'});        % return a myfints (10x10) since last two dimension are singletons.
a = x(:,:,'fac2', {'long', shor'}); % return 4-D xts: 10x10x1x2

a = squeeze(x(:,:,'fac2', {'long', shor'}));
% return a 3-D xts: 10x10x2, the factor dim disappeared since it's singleton

a = x(:,5,:,'short');
% return a 3-D xts: 10x1x5, last dim is singleton and removed automatically

a = squeeze(x(:,5,:,'short'));
% return a myfints: 10x5, last dim removed automatically and security dim squeezed since it's singleton

a = x(1,:,:,:);                    % return 4-D xts: 1x10x5x2
a = x(1,:,'fac1','long');          % return myfints: 1x10, traling singleton dims removed
a = squeeze(x(1,:,'fac1','long'))  % same as above since myfints don't have squeeze ops
a = squeeze(x(1,:,:,:));           % return a 10x5x2 matrix, since squeeze ops on an xts

a = squeeze(x(1,:,'fac1',:))
% return a 3-D xts: 1x10x2, since squeeze ops on 1x10x1x2 xts which keeps time dim even it singleton
```

The rules are

1. Traling singleon dimensions beyond 2 are always removed automatically.

2. 2-D xts will automatically casted into a myfints.
3. Squeeze() has no effect on a myfints.
4. If some dimensions but time are singleton in an xts, squeeze() will remove them.
5. If the only singleton dimension is time in an xts, squeeze() will remove time dimension and return a plain matrix.

5.5 Packing Matrix into `xts`

mat2xts

Convert data from MATLAB's vectors into an `xts` object.

Usage

```
xts = mat2xts(date, val, fields1, ..., fieldN)
```

Convert vectors to an (N+1) xts.

Input

date, val

vectors represent dates and corresponding values, receptively.

field1,...,fieldN

vector represents individual field dimensions (e.g., securities ids).

Note that the sizes of `date`, `val`, `fields1`,..., and `fieldsN` must be agree with each other. The orders of these vectors are irrelevant (ordering will be done inside `mat2xts`).

Output

xts an $N+1$ dimentional `xts`. The number of periods (time points) is determined by number of unique values in `date`; other dimension's size is determined by unique values in corresponding `fields1`.