

Performing equity investment simulations with the `portfolioSim` package

Kyle Campbell, Jeff Enos, and David Kane

October 2, 2007

Abstract

The `portfolioSim` package provides a flexible system for back-testing the performance of investment strategies using historical data. The package is designed to take into account the limitations of day-to-day trading, so that a simulation's performance will resemble as closely as possible the performance that a given strategy would actually have achieved over a given period of time.

1 Introduction

Note: this document is unfinished and is a work-in-progress.

Whatever investment strategy one uses to manage a portfolio, perhaps the most basic and most important question one can ask of it is: “How well does my strategy work?” Whenever one uses a systematic model to make financial decisions, one would like to have some assurance that the model will yield positive returns on an investment. Unfortunately, due to the unpredictability of the market, no estimates of a model's performance will ever be absolutely certain. The only sure indicator of an investment's value is the returns one can measure after the fact.

Therefore, barring the ability to foresee the future course of the market, the best way to gauge a model's accuracy is to examine its track record. Using historical data, it is possible to calculate how much money one would have made or lost had one invested according to a given strategy over some past period of time. This technique, known as “backtesting,” is widely used by financial professionals to test investment strategies. By making the assumption that there is at least some correlation between a model's past and future performance, investors can use the results of a backtest to make an informed decision on whether to use a strategy in their future investments.

The `portfolioSim` package is intended to give investors the tools to make such a decision. It does so by simulating a changing portfolio of investments over

some period in the past. The simulator makes use of the `portfolio` package¹ to manage the portfolio. The simulator takes historical market data supplied by the user and follows an investment strategy, also provided by the user, to determine which stocks to trade and when to trade them. At the end of the simulation, the user is provided with detailed information on the portfolio's performance over that period. From this information, the user can determine whether or not the investment strategy was effective.

The `backtest` package² can be used to conduct one specific type of simple backtest for a single type of portfolio, specifically, a long-short portfolio formed from the highest and lowest ranked stocks of an input signal. The `portfolioSim` package, in contrast, is much more flexible, allowing the user to specify virtually any criteria for constructing and maintaining a portfolio.

In addition to being far more flexible in both the input it can receive and the output it returns, the `portfolioSim` package has several major advantages over the `backtest` package. One of the most important is its ability to step through time, period by period, making investments based on many market variables as they stood at some point in the past. The `backtest` package reduces the problem of forming a portfolio to a single ranking variable, buying the highest ranked stocks and shorting the lowest. But in reality, there are often many other factors that an investor must take into account. It may take several days to trade to a desired portfolio, and by the time all the selected stocks are actually acquired the prices may have changed and the stocks may no longer be desirable. The cost of making trades must be taken into account as well, along with the turnover in the portfolio.

The `backtest` package ignores all of these complications, and thus the results it reports, while useful for gauging the accuracy of a stock ranking, do not accurately reflect the returns one could expect to gain from actual investments. The `portfolioSim` package, on the other hand, is so named because it tries to simulate, as closely as possible, the day-to-day trading process as it occurs in reality. It allows the user to take into account daily trading volume, stock prices, trade-cost adjustment, portfolio equity, and many other such considerations that the `backtest` package ignores.

2 Running a simulation

At its core, the `portfolioSim` package is a very simple machine for maintaining a representation of a changing portfolio over some span of time. At any moment, this portfolio consists of holdings such as:

```
id shares
1 A      10
```

¹See Enos and Kane, *Analysing equity portfolios in R*, for an introduction to the `portfolio` package.

²See Campbell, Enos, Gerlanc, and Kane, *Conducting Backtests in R*, for more information on the `backtest` package.

```

2 B    20
3 C   -10

```

Changes to a portfolio can be made in the form of “trades”.

trade: A transaction which changes the current holdings of the portfolio by means of buying, selling, shorting, or covering stocks.

Consider the follow list of trades:

```

id side shares
1 B   S    20
2 C   C     5
3 D   B    10
4 E   X    20

```

Applying these four trades to the holdings above would transform the portfolio into a new set of holdings, shown below:

```

id shares
1 A    10
2 C    -5
3 D    10
4 E   -20

```

Essentially, running a simulation consists of exposing a portfolio to different sets of trades at different points in time and observing the results. The duration of time over which the simulation is conducted is divided up by the user into discrete “periods”, at each of which a new set of trades is performed. The manner in which trades are selected for a given period is how we define an investment strategy. This strategy is supplied by the user in the form of an interface passed to the simulator.

Because there are no restrictions on how the user sets up an investment strategy, it is possible that the trades interface will return a set of trades that cannot be performed for one reason or another. For example, if a stock has an average daily trading volume of one thousand shares and one of the trades returned by the trades interface is to buy two thousand shares of that stock, the simulator will have to know that it can only expect to fill part of this trade in a single period.

In order to check for such limitations, the simulator needs to have access to a fair amount of information on all the securities that might be traded in each period. Specifically, the simulator needs to know the daily trading volume for each security, the price of each security at the start and the close of the trading day, and the returns from each security over the given period. All of this data must be supplied by the user, in the form of another interface passed to the simulator. This interface is queried at the start of every period, and provides the simulator with all the data it will need during that period.

These two interfaces, along with a list of periods, are the primary forms of input the simulator requires from the user. When a simulation is run, the simulator goes period by period, processing the following steps at each period:

1. **Query data interface:** First, the simulator retrieves all the data pertaining to the period. This includes the price, volume, and return information noted above, as well as any other information that might be required by the interfaces. For example, the trades interface may require some additional information to help it determine the set of trades to make. Where this data is stored depends entirely on the data interface. It could be saved in the interface itself, in a local database, or over a network.
2. **Clean up current holdings:** Next, the simulator compares the holdings currently in the portfolio to the new data returned by the data interface. If there are any holdings which are no longer in the investable universe (for example, holdings in delisted securities), then the simulator attempts to remove those holdings from the portfolio.
3. **Query trades interface:** Next, the trades interface is used to obtain a list of desirable trades. All the details of trade selection that affect which trades receive priority, such as trade-cost adjustment, must be handled within the trades interface. The simulator will change the trades returned from the interface only if the number of shares to be traded exceeds some set percentage of the daily trading volume. All other considerations must be dealt with by the interface itself. The `stiFromSignal` interface included in the `portfolioSim` package is a highly flexible interface that takes into account many of considerations one would use to make real trades, including trade-cost adjustment, target equity, and trading volume (see section 4 for more details).
4. **Save start data:** For each period, the simulator saves three types of results: data on the portfolio at the start of the period, data on the trades made during the period, and data on the portfolio at the end of the period. The user has some flexibility about which types of data should be saved, but the most basic type of results for the starting portfolio consists of the long and short equity and the size of the long and short sides of the portfolio.
5. **Save period data:** After the results for the start portfolio are saved, the simulator saves the results for the period itself. This consists of the turnover in the portfolio, the turnover in the universe of investable stocks, and the performance of the portfolio during this period. The performance is calculated based on the returns of the stocks in the portfolio at the *start* of the period. In other words, the performance for the portfolio resulting from trades in one period is calculated during the following period. In addition, the user can select to save details on the performance of individual stocks, and/or the list of trades to be performed during this period.
6. **Perform trades:** Once the start and period results have been saved, the simulator exposes the portfolio to the final list of trades for the period. First, the trades are checked to make sure they do not exceed the fixed fill volume percentage; any trades that do are adjusted to match the maximum

allowed percentage of the daily trading volume. Then the holdings in the portfolio are transformed according to the list of trades.

7. **Save end data:** Finally, the simulator saves the results for the portfolio at the end of the period. Again, this consists primarily of saving the size and equity of both sides of the portfolio.

These steps are repeated for each period in the simulation, with the portfolio carrying over from one period to the next. After all the periods have been processed, the simulator is left with summary information on each step of the portfolio's history, from which it is easy to calculate the overall performance of the portfolio.

2.1 A simple example

Consider a basic simulation conducted over four periods in a market with only four stocks: "A", "B", "C", and "D". The prices of these stocks remain constant over the course of the simulation, and the daily trading volume is 100 shares for each stock. For the purposes of this example, we ignore the implementation of the data and trades interfaces. We begin the simulation with no holdings.

First, the simulator gets the data for the first period:

	period	id	ret	start.price	end.price	volume	universe
1	1	A	0	10	10	100	TRUE
2	1	B	0	10	10	100	TRUE
3	1	C	0	10	10	100	TRUE
4	1	D	0	10	10	100	TRUE

Since the current portfolio is empty, we have no holdings to compare to the data. Next, the simulator gets a list of trades from the trades interface:

	id	side	shares
1	A	B	10
2	B	X	10

For the first period, we want to perform two very simple trades: buying 10 shares of stock "A" and shorting 10 shares of stock "B". Both of these amounts are lower than the maximum fill volume percentage (15 percent), so the simulator can carry out the complete list of trades. Our new holdings after the first period are thus:

	id	shares
1	A	10
2	B	-10

The simulator then proceeds to the next period after saving out the summary data for the first period. New data and a new set of trades are retrieved from the interfaces. We now have a slightly more complex set of trades to be performed:

```

id side shares
1 B    C      5
2 C    B      5
3 A    X     10
4 A    S     10

```

During this period, we want to cover some of the shares of stock “B” which we had shorted in the previous period; we want to buy into stock “C”, and we want to switch our holdings of stock “A” from the long side to the short side of the portfolio. Again, all of these trades are less than 15 shares, so the simulator can fill them all. Our holdings after the second period are:

```

id shares
1 A    -10
2 B     -5
3 C      5

```

In the trades for the third period, however, not all of the trades can be filled during a single period:

```

id side shares
1 B    C      5
2 D    B     20

```

The trades interface has returned a list of trades that includes buying 20 shares of stock “D”. However, the daily trading volume for this stock is 100 shares, of which the simulator will fill a maximum of 15%. Therefore, only 15 shares of stock “D” will be bought during this period.

```

id shares
1 A    -10
3 C      5
4 D     15

```

Note also that we have covered the rest of our shares in stock “B”, so it no longer appears in our holdings.

Finally, in the last period, the trades interface returns only one trade: the remaining 5 shares we had intended to buy of stock “D”. Note that the simulator does not automatically process left over trades during the following period, but this is something that a good trades interface will take into account.

```

id side shares
1 D    B      5

```

Looking at the data for period 4, we see that stock “C” is no longer in the universe of investable stocks:

```

period id ret start.price end.price volume universe
13      4 A  0          10         10    100    TRUE
14      4 B  0          10         10    100    TRUE
15      4 C  0          10         10    100    FALSE
16      4 D  0          10         10    100    TRUE

```

Because our current holdings include shares of stock “C”, the simulator will attempt to remove these shares from the portfolio before processing the trades for this period. Therefore, our updated portfolio before we perform the final set of trades contains these holdings:

```
id shares
1  A    -10
4  D     15
```

Finally, we buy the remaining 5 shares of stock “D”. There are no more periods, so the simulation is finished. Our final holdings are:

```
id shares
1  A    -10
2  D     20
```

This example covers only the most basic functionality of the simulator. There are many other features built into the `portfolioSim` package, including the ability to calculate exposures and contributions across different variables. Most importantly, the implementation of the different interfaces allows the user include build many custom features into the simulator.

3 Overview of the portfolioSim package

Of the classes contained in the `portfolioSim` package, the user interacts primarily with objects of the `portfolioSim` class, used to conduct the simulation, and objects of the `simResult` class, used to store and analyze the results of the simulation. In addition, there are three interface classes which the user can choose to implement in order to customize the simulation to meet specific needs.

3.1 The portfolioSim class

When beginning a new simulation, the first step is to construct an object of class `portfolioSim` which will contain all the information required by the simulator. An instance of class `portfolioSim` represents a unique simulation, which can then be run at any time by calling the `runSim` method. This allows the user to make changes to the simulator after a run and, over repeated simulations, to see how those changes affect the results.

A `portfolioSim` object contains the following slots:

- **periods**: A data frame listing the periods to be used in the simulation. Each period represents a single iteration of the simulator, in which a new set of trades is calculated and carried out. The periods data frame must have columns `period`, `start`, and `end`. The `period` column contains labels which are used throughout the simulator to represent each period. The `start` and `end` columns are used to differentiate between saved data from before and after the trades are performed in each period. Generally, these columns should contain the actual dates corresponding to each period.

	period		start		end
1	1	2006-01-01	09:30:00	2006-03-31	16:00:00
2	2	2006-04-01	09:30:00	2006-06-30	16:00:00
3	3	2006-07-01	09:30:00	2006-09-30	16:00:00
4	4	2006-10-01	09:30:00	2006-12-31	16:00:00

- **freq**: The annual frequency of the periods listed in the periods slot. For example, the frequency corresponding to the periods data frame shown above is 4. When running a simulation with monthly periods, the frequency should be 12. With daily periods, it should be 252, the total number of trading days in a year.
- **data.interface**: A data interface object of some class containing the virtual class **simDataInterface**. The data interface serves to transform the raw data used in the simulation into an object of class **simData**, containing information on a single period.
- **trades.interface**: A trades interface object of some class containing the virtual class **simTradesInterface**. The trades interface represents the implementation of the trading strategy to be tested in the simulation. Based on the current portfolio and the data available for a given period, the trades interface contains some mechanism for determining a set of trades to make. These trades are encapsulated in a **simTrades** object which the interface returns to the simulator. The default trades interface is an object of class **stiFromSignal**, which uses some signal to rank stocks and form a portfolio based on those rankings.
- **summary.interface**: An optional summary interface object of a class containing the virtual class **simSummaryInterface**. The summary interface allows the user to specify information to be saved out during the simulation beyond that supported by the result classes **instantData** and **periodData**.
- **start.holdings**: A portfolio object representing the portfolio at the start of the simulation. If this slot is not specified, the simulator starts with an empty portfolio. See the documentation in the **portfolio** package for information on constructing a portfolio.
- **fill.volume.pct**: The maximum percentage of the daily trading volume of a stock that the simulator is allowed to trade in a single period. The default is 15.
- **exp.var**: A character vector of additional variables to be used when analyzing the exposures for each period. See section 7 for more information.
- **contrib.var**: A character vector of additional variables to be used when analyzing the contributions for each period. See section 7 for more information.

- **out.loc**: The location of a directory, relative to the current working directory, to which the simulator will save out the results. The **runSim** method will also return the full **simResult** object, but for large simulations it is more efficient to let the simulator save out to a directory and then load in the desired results after the simulation has finished.
- **out.type**: A character vector specifying what kind of information the simulator should remember from each period. There are five basic types, any combination of which can be specified:
 - **basic**: Saves information on the equity and size of the portfolio, the turnover in both the portfolio and the universe, and summary information on the portfolio’s performance for each period.
 - **detail**: Saves performance details for all stocks in the portfolio at each period.
 - **exposures**: Saves exposures for each period.
 - **contributions**: Saves contributions for each period.
 - **trades**: Saves the list of trades performed for each period.

In addition, there are three preset configurations of these types that can be specified in the **out.type** slot:

- **all**: Saves all five types of data.
- **default**: Saves the basic result information and the trades for each period. This mode also saves exposures if an **exp.var** has been specified and contributions if a **contrib.var** has been specified. This is the default behavior for the simulator.
- **lean**: Same as the “default” type above, except trades are not saved.

3.2 Interfaces

The flexibility of the **portfolioSim** package stems from three virtual classes which serve as interfaces that the user can implement to create a customized simulation. Each of these interfaces deals with a different part of the simulator, but multiple interfaces can be designed to work together. The first two interfaces deal with the two primary types of input required by the simulator.

- **simDataInterface**: All the raw data used in the simulation is accessed through the data interface. This interface is supplied to the simulator by the user, and must be an object containing the virtual class **simDataInterface**. The actual data can be stored in as an R object in the interface itself, or the interface can query some outside database to obtain information on a given period. An implementation of the **simDataInterface** class must contain the method **getSimData**, whose purpose it is to retrieve the raw data and transform it into a **simData** object.

A `simData` object contains all the information the simulator will need during a single period. This information is stored as a data frame, with each row representing a unique security. The simulator itself requires every `simData` object to include the following columns:

- `period`: The period to which the data in this `simData` object corresponds. Since a `simData` object contains data from only one period, all the values for `period` should be identical. This value should also match one of the periods in the `periods` slot of the `portfolioSim` object.
- `id`: A unique identifier for each security.
- `start.price`: The price of this security at the beginning of the period.
- `end.price`: The price of this security at the end of the period.
- `ret`: Total return for this period.
- `universe`: A logical vector indicating whether securities are in the universe of investable stocks for this period.
- `volume`: The daily trading volume of this security.

Since the simulator requires all this data, it must either be included in the raw data retrieved by the `simDataInterface`, or the interface must have some means of calculating it. The other interfaces used in a simulation might require additional columns to appear in the `simData` object. For example, the trades interface included with the `portfolioSim` package, `stiFromSignal`, requires some “signal” column in the data (see section 4). When designing multiple interfaces together, it can be useful to require additional columns in `simData` objects.

A very simple data interface, the `sdiDf` class, is included with the `portfolioSim` package. This interface stores all the raw data in a data frame, and the `getSimData` method simply subsets the data frame by period. Because the interface performs no additional work to format the data, all the columns required throughout the simulation must be included in the raw data when using the `sdiDf` interface.

- **`simTradesInterface`**: The trades interface represents the investment strategy the simulator uses to select trades. This interface, stored in the `trades.interface` slot of the simulator, determines the trades to perform at each period and returns them as an object of class `simTrades`. The process by which these trades are determined takes place entirely within the `getSimTrades` method of the interface, so the possible trading strategies are limitless. The only requirement is that the method return a `simTrades` object. This class is basically a wrapper for an object of the `trades` class contained in the `portfolio` project. A `trades` object is simply a data frame with columns `id`, `side`, and `shares` indicating which

stock to trade, how to trade it (buy, sell, short, or cover), and how much of it to trade.

The trades interface `stiFromSignal` is included in the `portfolioSim` package. This class is designed to take full advantage of the `tradelist` features found in the `portfolio` package. It can be used to test the effectiveness of any kind of signal one might use to make investments, anything from professional stock analyses to one-day returns. See section 4 for a detailed introduction to using the `stiFromSignal` interface.

The third interface allows the user to customize the output of the simulator. The `out.type` slot of the `portfolioSim` class allows for a considerable amount of flexibility in the type of data the simulator can return. However, in some cases the user might want to save out some information beyond what the five basic result types allow. The summary interface gives the user this ability. Because the output types built into the result classes are sufficient for the most common types of simulations, this interface is optional.

- **simSummaryInterface:** The summary interface is the most open-ended of the interfaces. At each period, the simulator passes to the summary interface a snapshot of the current portfolio, along with the `simData` and `simTrades` objects for the period. What the summary interface does with this data is entirely up to the user. Generally, the interface will save some sort of statistical data on the portfolio at the current period. This data can either be stored in the interface itself, or it can be saved out like the other simulation results. The `updateSummary` method is responsible for computing whatever summary data the user wants to keep and storing it to be accessed later.

The summary interface requires a second method, called the `summary` method. This method is called from within the `summary` method of the `simResult` class, and is responsible for accessing and displaying whatever data the interface is storing. Other accessor or helper methods for the interface can be provided at the user's discretion.

3.3 Classes for storing simulator results

The `runSim` method of class `portfolioSim` stores the results of the simulation in a hierarchy of results classes, outlined below. The user most often interacts with objects of the `simResult` class, which provides methods for displaying statistical and summary information on the simulation.

- **instantData:** Stores information about the portfolio at a single point in time. Specifically, this class stores information on the current holdings in the portfolio, the size and equity of the long and short sides of the portfolio, and any exposures if the simulator has some `exp.var` specified.
- **periodData:** Stores information on a single period in the simulation. This includes turnover in the portfolio and in the universe, all the trades done

in a given period, the performance of the portfolio during that period, and contributions to that performance if a `contrib.var` is specified.

- **simResultSinglePeriod**: For each period in the `periods` slot of the simulator, one object of class `simResultSinglePeriod` is created. This class contains one slot for an object of class `periodData`, and two slots for objects of class `instantData`, one representing the portfolio at the start of the period and one at the end.
- **simResult**: Stores information on the simulation as a whole, including annual frequency of the periods, the type of results stored, and any error messages that occurred during the simulation. Most importantly, `simResult` objects contain a list of the `simResultSinglePeriod` objects for each period in the simulation. In addition, it stores the final version of the summary interface, if one is specified.

For more information on accessing the results of a simulation, see section 6.

4 Using the `stiFromSignal` interface

The `stiFromSignal` trades interface included in the `portfolioSim` package requires some numeric ranking of stocks. It then generates trades intended to maintain a portfolio with the highest ranked stocks on the long side and the lowest ranked stocks on the short side. This is only one trading strategy, but it is very useful for testing the accuracy of any model that makes quantifiable predictions of a stock's future performance. Testing such models is one of the most common reasons for conducting a backtest.

The core of the `stiFromSignal` interface is in the generation of a `tradelist` object, part of the `portfolio` package. The mechanism by which `tradelist` calculates trades lies beyond the scope of this article.³ This section is intended to allow a user unfamiliar with `tradelists` to use the `stiFromSignal` interface to conduct simulations.

An object of class `stiFromSignal` has the following slots:

- **in.var**: A variable to be used as the “signal” by which stocks are ranked. The `in.var` must be a column of the data frame stored in the `simData` objects that the data interface returns.
- **type**: The type of weight formation to be used when forming the portfolio (see the documentation for the `portfolio` package for details). The default is “equal”, which results in an equal-weighted portfolio.
- **size**: The size of the portfolio to be formed; can either specify the number of securities per side, or can be relative to the total number of securities. The default is “quintile”, meaning each side of the portfolio will consist of one fifth of the stocks for which the `in.var` provides rankings.

³For a detailed introduction to `tradelist`, see Enos, Gerlanc, and Kane, Trading with the portfolio package.

- **sides**: The sides to be contained in the portfolio. Can be “long”, “short”, or c(“long”, “short”).
- **equity**: The total equity of the portfolio.
- **target**: An environment in which the interface stores its target portfolio between periods. Because of the restrictions placed on trades by **tradelist**, very often it is not possible to attain the portfolio specified by the signal during a single trading period. Therefore, the target portfolio is saved so that the interface will continue trading towards that target in subsequent periods.
- **rebal.on**: A vector of periods during which the target portfolio is to be rebalanced. These periods must correspond to the periods in the **periods** slot of the **portfolioSim** object. Rebalancing is simply the process of forming a target portfolio from the **in.var**. The interface rebalances automatically if there is no saved target. Otherwise, it will continue trading to the target portfolio until it reaches a **rebal.on** period.
- **trading.style**: The style of trading to use. Possible styles are:
 - **immediate**: The default trading style, this style returns trades that immediately transform the current portfolio into the target portfolio. This style is the simplest, but also the most unrealistic since it overrides many of the **tradelist** features intended to simulate actual trading.
 - **percent.volume**: Returns trades for a maximum of 15 percent of a stock’s daily trading volume; otherwise the same as “immediate”.
 - **15.pct.vol.to.equity**: The most realistic trading style, making use of all the features in **tradelist**, such as trade-cost adjustment. This style requires the column **md.volume.120.d** in the **simData** object, and allows for sorts if the column **alpha.6** appears.
- **chunk.usd**: The maximum chunk size, in U.S. dollars, into which **tradelist** breaks up possible trades.
- **turnover**: The maximum turnover allowed in the portfolio per period. Defaults to infinity, placing no restriction on turnover.

The **getSimTrades** method of **stiFromSignal** consists of two basic steps. First, it generates a target portfolio. If the portfolio is not being rebalanced, this means simply retrieving the saved target. If the portfolio needs to be rebalanced, the **portfolio** package is used to generate a new portfolio from the **in.var**. The kind of portfolio created depends on the **type**, **size**, **sides**, and **equity** slots. Second, the current portfolio and the target portfolio are used to create a new **tradelist** object. What kinds of trades are generated depends on the **trading.style** specified, along with the **chunk.usd** and **turnover** slots.

The interface then returns the `trades` object contained in the `tradelist`, and saves the target portfolio to the `target` environment to be used in the next period.

5 A multi-period example

Starmine is a San Francisco based research company that creates rankings of stocks based on predicted future earnings. One such ranking is the StarMine Indicator, which ranks stocks on a scale of 1 to 100, with 100 being the highest.⁴ We can use the `stiFromSignal` interface to test the accuracy of the StarMine Indicator. If there is indeed a positive correlation between StarMine rankings and returns, we should expect a long-short portfolio formed by buying the highest ranked stocks and shorting the lowest ranked stocks to yield high returns. How well our portfolio performs should give us some idea of whether we wish to use the StarMine Indicator to make investment decisions.

The data set `starmine.sim` included in the `portfolioSim` package contains StarMine Indicator rankings for stocks from January 31, 1995 to November 30, 1995. In this data set, the rankings are updated monthly. All the other data we need to run a simulation are also included in the data set.

```
> data(starmine.sim)
> names(starmine.sim)

[1] "id"           "date"           "name"
[4] "country"      "sector"         "cap.usd"
[7] "size"         "smi"            "fwd.ret.1m"
[10] "fwd.ret.6m"   "price.usd"      "prior.close.usd"
[13] "volume"       "ret.1m"
```

The first step in constructing our `portfolioSim` object is to create the periods data frame. Looking at the `date` column, we can easily construct a data frame that looks like this:

```
> periods <- data.frame(period = sort(starmine.sim$date[!duplicated(starmine.sim$date)]))
> periods$start <- periods$period
> periods$end <- c(periods$start[-1], as.Date("1995-12-31"))
> periods

   period      start      end
1 1995-01-31 1995-01-31 1995-02-28
2 1995-02-28 1995-02-28 1995-03-31
3 1995-03-31 1995-03-31 1995-04-30
4 1995-04-30 1995-04-30 1995-05-31
5 1995-05-31 1995-05-31 1995-06-30
6 1995-06-30 1995-06-30 1995-07-31
7 1995-07-31 1995-07-31 1995-08-31
```

⁴See www.starmine.com

```

8 1995-08-31 1995-08-31 1995-09-30
9 1995-09-30 1995-09-30 1995-10-31
10 1995-10-31 1995-10-31 1995-11-30
11 1995-11-30 1995-11-30 1995-12-31

```

Special attention must be paid to the **period** column. These are the labels used at every level of the simulator to identify the current period. Any time the data or trades interfaces need to refer to a period, it must be identical to one of the periods in this column. On one other hand, these labels can also be more abstract than the **start** and **end** columns; it is perfectly acceptable to simply number the periods, so long as the numbering system is consistent throughout the simulator.

The next step is to set up the data interface. Note that of the information we need for our **simData** objects (**id**, **period**, **start price**, **end price**, **volume**, **return**, and **universe**), all except for “universe” are contained in the data set. However, the column names do not match those required by the simulator. If we expected to run many simulations with data formatted in this same way, it would probably be worth our time to write a new **simDataInterface** which, as part of its **getSimData** method, would change all the names for us and add a **universe** column. However, for the purposes of this example, it is easier to simply make these changes manually and use the **sdIdf** interface included with the package.

We already have **id** and **volume** columns in the data set. Our **period** column is clearly **date**, so we can rename this first.

```
> starmine.sim$period <- starmine.sim$date
```

It is important to note that while the periods are spaced at monthly intervals, most of the columns in the data set refer to single days, not months. We must therefore consider carefully how we would go about trading in the real world, if we had only the information contained in this data set. Essentially, we are only allowed to trade on one day of every period, specifically the last trading day of the month. We therefore take the closing price of the day before the last trading day of the month as the **start.price**, and the closing price for the last trading day of the month as the **end.price**.

```

> starmine.sim$start.price <- starmine.sim$prior.close.usd
> starmine.sim$end.price <- starmine.sim$price.usd

```

We now need to select a column for the returns the simulator will use to calculate the portfolio’s performance. The simulator uses the portfolio at the start of each period, before any trades are made, to calculate the performance for that period. In other words, we assume that it takes the entire span of a period for us to trade to a new portfolio. So even though we are trading on January 31, 1995, the simulator assumes that we do not get the portfolio resulting from those trades until the end of the period, on February 28, 1995. This is somewhat counter-intuitive, because we want to use returns from the

month of February to calculate the performance of the portfolio that we traded to at the end of January (remember that we actually do all our trading on January 31, and then hold that portfolio throughout the rest of the period). To correct this, we use one month backward returns so that the performance for the period beginning on February 28 is calculated using the portfolio we formed on January 31 and the returns from the month of February.

```
> starmine.sim$ret <- starmine.sim$ret.1m
```

Finally, we need to add a `universe` column to the data set. The `universe` column is a logical indicating which of the stocks in the `simData` object the simulator will be allowed to trade. The data interface can define this flag in any number of ways. For the purposes of this example, we will assume that all the stocks for which we have data exist in the investable universe for that period.

```
> starmine.sim$universe <- TRUE
```

Now that we have our data correctly formatted, we can create a new object of class `sdiDf` to be used as the data interface in our simulation.

```
> data.interface <- new("sdiDf", data = starmine.sim)
```

Again, the process of reformatting our input data would quickly become tedious if we were running multiple simulations like this. It would be quite easy to create a modified version of `sdiDf` to do all this work for us.

After the data interface has been created, we move on to the trades interface. In this example we will use the `stiFromSignal` interface, which is specifically designed for testing the accuracy of a numeric ranking such as the StarMine Indicator.

We set the `in.var` to be `smi`, the ranking that the simulator will use to form the target portfolio. For the sake of simplicity, and because we have a relatively small number of periods, we want the interface to rebalance the target portfolio at every period, so we simply pass the `period` column from the `periods` data frame to `rebal.on` slot. Because of the large number of stocks in our data set, we set the size of the portfolio to “decile” and the total equity to one million U.S. dollars. All the other slots keep their default settings. We do not worry about setting a target environment; the initialize method of `stiFromSignal` will do this for us.

```
> trades.interface <- new("stiFromSignal", in.var = "smi",
+   size = "decile", equity = 1000000, rebal.on = periods$period)
```

This interface will return trades for creating and maintaining a long-short, equal-weighted portfolio, with the long side formed from the ten percent of stocks receiving the highest StarMine Indicator rankings, and the short side formed from the ten percent receiving the lowest rankings. The total equity of the portfolio will be kept at \$1,000,000. Trades will be selected based on the “immediate” trading style, meaning that at each period, the list of trades

returned will shift all of our holdings to match the target portfolio. This target will be rebalanced at each period to match the StarMine Indicator rankings for that month.

Finally, we are ready to construct a `portfolioSim` object to contain all these elements of the simulator:

```
> ps <- new("portfolioSim", periods = periods, freq = 12, data.interface = data.interface,
+   trades.interface = trades.interface, fill.volume.pct = Inf,
+   out.loc = "out_dir_2", out.type = "lean")
```

Because we are using monthly periods, we set `freq` to be 12. Because we do not specify any `start.holdings`, the simulator will begin with an empty portfolio. The `fill.volume.pct` is set to infinity so that all the trades returned from the trades interface will be filled. The `out.loc` can be an existing directory, or a non-existent directory in which case the simulator will create it. Our `out.type` is “lean”, so only the “basic” type of results will be saved out, since we have no `exp.var` or `contrib.var`. We also have no summary interface to specify, so the simulator will save no additional information.

Once our `portfolioSim` object is created, we simply call the `runSim` method to conduct the simulation.

```
> result <- runSim(ps, verbose = FALSE)
```

The `runSim` method steps through each period of simulation. It first uses the data interface to retrieve a `simData` object containing the data for that period. Second, the trades interface is called to generate the trades to be conducted for that period. Finally, the `performance` and `expose` methods of the `portfolio` class are used to update the holdings based on the trades returned from the trades interface. A `simResultSinglePeriod` object is created for each period, and stored in a master `simResult` object. The results are also saved out after each period to the `out.loc` directory specified in the `portfolioSim` object.

6 Analyzing the simulation results

The `summary` method of a `simResult` object can be used to obtain a quick summary of the simulation results.

```
> summary(result)
```

Simulation summary:

```
Start: 1995-01-31
End:   1995-11-30
```

	profit	return
Total:	228,313	22.5 %

Sharpe:	3.3		

Mean return:	1.9 %		
Mean return (ann):	22.5 %		
Volatility:	1.9 %		
Volatility (ann):	6.7 %		

Best period:	55,031	6.3 %	(1995-10-31)
Worst period:	-259	-5.7 bps	(1995-05-31)
Worst drawdown:	12,386	1.1 %	(1995-04-30 to 1995-05-31)

Mean size:	494		
Mean equity:	831,496		
Mean gross equity:	1,662,992		
Universe turnover:	0		
Turnover:	26,258,923		
Mean turnover:	2,188,244		
Mean turnover (ann):	26,258,923		
Holding period (mth):	1.5		

Mean NA weights:	10		
Mean NA returns:	8		

This display summarizes the data saved by the simulator when using the “basic” `out.type`. From this summary, we can quickly observe that our investment strategy based on the StarMine Indicator yielded a profit of \$228,313, or 22.5% of our original \$1,000,000 investment. Across our monthly periods, the mean return was 1.9%. The best month for our portfolio was October, while our worst returns came in May. Overall, it seems like our strategy would have worked very well during 1995. While such strong returns are encouraging, we would have to compare these figures to the behavior of the market as a whole during that year, in order to gauge how well the model performed relative to the rest of the universe.

If the simulation is conducted with the “detail” `out.type`, the summary method prints two lists of the best and worst performing stocks over the entire simulation. The first list is based on profit, while the second list is based on contribution.

Simulation summary:

Top/bottom performers by profit:

	id	profit
799	45031210	-5973
357	16961710	-5127
284	11157210	-3930
1001	52465R10	-3292
1338	64108110	-2618
2414	G2847110	-2557

```

317 12738710 4032
372 17275510 4227
68 01877H10 4393
1742 680492 4574
2417 N5424G10 4720
2000 72003530 5322

```

Top/bottom performers by contribution (%):

```

      id contrib
799 45031210 -0.60
357 16961710 -0.50
284 11157210 -0.40
1001 52465R10 -0.34
2056 75886F10 -0.26
1338 64108110 -0.25
317 12738710 0.40
372 17275510 0.40
68 01877H10 0.43
1742 680492 0.46
2417 N5424G10 0.46
2000 72003530 0.53

```

The saved results from a previous simulation can be loaded into a new R session by using the `loadIn` method of class `simResult`:

```
> result <- loadIn(new("simResult"), in.loc = "out_dir_2")
```

All the analysis methods previously discussed can also be called on a `simResult` object loaded from previously saved results.

7 Running portfolioSim with contributions and exposures

Note: this section of the document is unfinished.

The `portfolioSim` object has two slots, `contrib.var` and `exp.var`, for specifying contribution and exposure variables. These are variables found in the data slot of `simData` objects that the interface returns. The simulator can thus analyze the exposures and contributions to the portfolio across other variables.

For example, suppose we want to look at the exposures to our portfolio from price and country. We add these two variables, both included in our data set, to the `exp.var` slot of the simulator, and change the `out.type` to “exposures”. (The “default” or “lean” types will automatically save exposures data if an `exp.var` is specified, but in this case we are only interested in looking at exposures.)

Likewise, to see the contributions to our portfolio from different sectors, we simply specify “sector” as a `contrib.var` and set `out.type` to “contributions”.

For an introduction to contributions and exposures, see the documentation in the `portfolio` package.

8 Conclusion

Simulating investments over some period in the past is often a time consuming and data-intensive process. The `portfolioSim` package automates the process of running such a simulation, by allowing the user to specify in advance all the details of the portfolio to be maintained, the investment strategy by which to maintain it, and the results from the simulation to be saved. The flexibility of the `portfolioSim` package stems from its interfaces, which can be customized to work with virtually any type of input data and any method of investment. Together with the `portfolio` package, `portfolioSim` provides investors with a powerful and adaptable set of tools for managing their investments.