

# Strategy Tester

Patrick Costello and Matt Johnson

October 22, 2012

I'll demonstrate how the parts of the Strategy Tester works by apply it to a simple Moving Average momentum example. The code's a bit of overkill for this example, but I'll put comments in to explain the reason for the design along the way. At the end I'll talk about how structure should generalise to more complex examples like strategies on curves.

The R code is broken into a few files:

- *Main.R* executes our strategy. It imports data, constructs required object, and displays results. However, it doesn't have function definitions, it's just a list of commands we want to execute.
- *MarketDataClasses.R*, *TradeClasses.R*, *PortfolioClasses.R*, *TradeStrategyClasses.R* have code that defines the different object we want to create, i.e. classes, and things we want to do to them, i.e. class functions (aka member functions). These files don't execute commands, except for a couple to check the classes are working properly. They just define objects and how they interact.

In the example below I'll just go through the execution of *Main.R* to show how I see the logical parts fitting together.

If you want we can go through class definitions later, but seeing as though we are probably going to do coding in futuer in Python/C++ then probably not worth spending too much time on R implementation as R OO is a bit weird.

I've broken the discussion into a few sections, corresponding a quick description of the MA strategy and then logical sections of the *Main.R* code. The fist part of *Main.R* basically just imports data for the AllOrds index, adds MA's and formats in a way that can be used by the StrategyTester classes. The second part gives a description of how to build Trade objects and combine them into Portfolio objects, then price/value them against the data, with comments on resoning behind definitions. Part 3 describes creating a TradeStrategy object from a MarketData and a Portfolio object, and then operations you can do on the TradeStrategy object like run the strategy and look at the results - this is the main section for generating strategy results, which is done in only a few lines. Part 4 is just goes through the chunk of code to display the results. The last part is discussion of further things we can do.

## 1 Moving Average Momentum Example

The basic idea, illustrated in Figure 1 below, is to take two moving averages of a time series MA1, MA2 where MA1 had less lags than MA2. We think of MA2 as being a more long term average than MA1, and so when MA1 crosses MA2 there is momentum away from the longterm average, and we trade in that direction. I.e. trade when lines cross in the direction of green line crossing blue line: upcross and we put on a buy; downcross we put on a sell; otherwise we hold. Sounds a bit too simple to work - we can test results with StrategyTester.

The code in the "Main.R" file runs through this for the AORD index from Yahoo finance using StrategyTester. The final results of portfolio value of strategy are in Figure 1. Note, I've assumed interest rates are zero, so when only cash position, we have no change in portfolio value. This is easy to change in future.



Figure 1: AORD index with MA's. Generated by quantmod

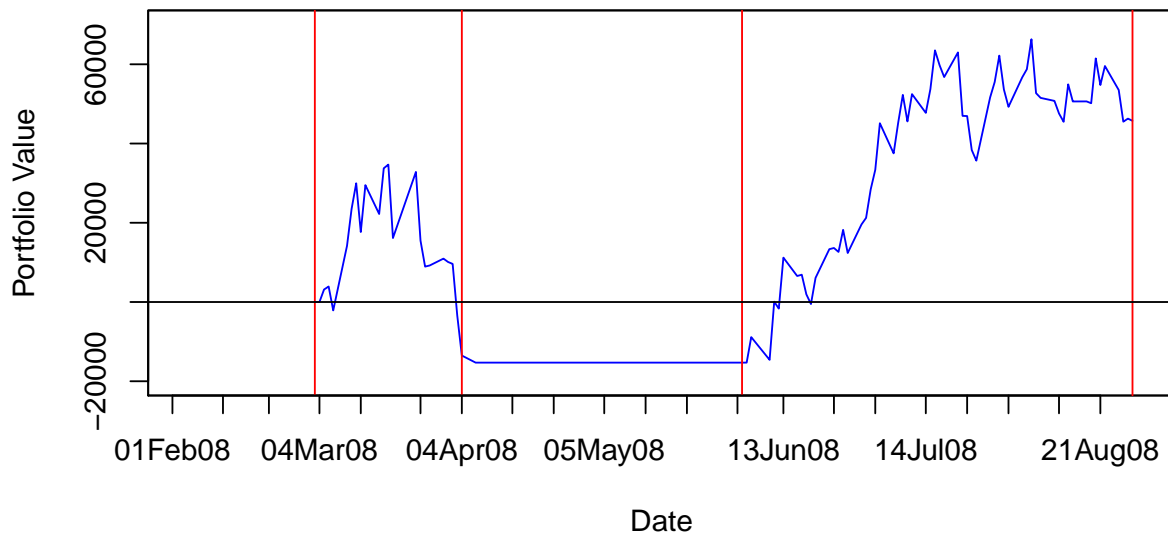


Figure 2: AORD index with MA's. Generated by quantmod

## 1.1 Data Import

The first portion of *Main.R* code doesn't have much to do with the StrategyTester files. It just uses R and quantmod to load in data and construct the moving averages.

First we load the libraries we need

```
library(xts)
library(zoo)
library(quantmod)
```

Then we set the working directory so that it will include the StrategyTester class files correctly,

```
setwd("~/Documents/R/StrategyTester")
```

Next we import data.

```
AORD = as.xts(as.zoo(read.table("Data/AORD.csv", header = T, sep = ",")))
```

The csv loaded here was one I prepared earlier using quantmod's canned functions for downloading yahoo data.

We then use quantmod to create the moving averages series by charting AORD then adding MAs, and extracting series values. I've suppressed chart as it just Figure 1 above.

```
chartData = AORD["2008-02::2008-08"]
chartSeries(chartData, TA = NULL, theme = "white")
EMA1 = addEMA(n = 10, col = "green") #Adding Moving Average to chart
EMA2 = addEMA(n = 20, col = "blue") #Adding Moving Average to chart
EMA1Vals = EMA1@TA.values #Extracting Vals
EMA2Vals = EMA2@TA.values #Extracting Vals
```

**Remark 1** *Quantmod, has a whole bunch of functions which add all different kinds of technical indicators (TA) to charts.*

Then we convert MA's to xts and combine with the original AORD data = chartData.

```
Series1 = as.xts(zoo(EMA1Vals, order.by = index(chartData)))
colnames(Series1) = "MAs"
Series2 = as.xts(zoo(EMA2Vals, order.by = index(chartData)))
colnames(Series2) = "MA1"
chartData = cbind.xts(chartData, Series1, Series2)
head(chartData, 2)
```

| ##            | AORD.Open | AORD.High | AORD.Low | AORD.Close | AORD.Volume |
|---------------|-----------|-----------|----------|------------|-------------|
| ## 2008-02-01 | 5717      | 5888      | 5714     | 5882       | 890030700   |
| ## 2008-02-04 | 5894      | 6058      | 5886     | 5922       | 865578100   |

| ##            | AORD.Adjusted | MAs | MA1 |
|---------------|---------------|-----|-----|
| ## 2008-02-01 | 5882          | NA  | NA  |
| ## 2008-02-04 | 5922          | NA  | NA  |

Note that the beginning MA values don't exist as the first value in the series can only be caculated when there are enough lags available, e.g. MAs will only begin after 10 periods, as it has 10 lags.

## 1.2 Market Data, Trade and Portfolio Classes

Following other finance libraries, like quantlib, I have broken down pricing trades into the following logical pieces, which are implemented as classes:

- Market Data. This contains data inputs, and interface to other classes.
- Trades. These objects contain info about trades that is not dynamic market data. Notionals, Trade-Type. Also need functions to Price, Value given market data.
- Portfolios. This just collections of trades.

There's a couple of other classes I've added whose purpose I'll describe as we go.

Continuing with *Main.R* we load the code that defines the classes

```
source("R/PortfolioClasses.R")
```

**Remark 2** • *Note, I have setup so that loading this files loads other classe files it needs.*

- *Make sure working directory is set correctly or won't load.*

Now let's look at how to build objects of these classes and what they do. We build an object **MD** of the **MarketData** class,

```
MD = MarketData(chartData) #Create Market Data
```

This is just a class that contains our market data. It is a bit redundant as all our code is in R, but will provide the bridge when we implement in other languages. I.e. we'll construct data in R because it's easy, then we can use the MarketData class to convert to form usable by C++/Python internal code we develop.

Next we build **T1**, **T2** objects of the Trade class I've defined.

```
T1 = Trade(Name = "Cash", Type = "Cash", Notional = 100)
T2 = Trade(Name = "AORD", Type = "Eq", Notional = 100)
```

So far I've only defined simple "Cash" and "Eq" types, and forgotten to include an argument for trade date info.

- "Cash" always has price = 1
- "Eq" has takes price from 'Close' of a MarketData object.

In the future we can make more complicated trade types like: futures, swaps, options, etc. These will have a lot more parameters, and much more complicated pricing formulas.

**Remark 3** *Pretty much all of the high tech pricing libraries, like quantlib, fit in here. This is a good example of how OO separates logical parts. If we want to swap in a different pricing library, then it will be only at this point that we need to redesign the interface.*

Now to Price/Value trades, we need to do it against a time slice of market data. For this reason I created the **MarketDataSlice** class, which is combined with Trade objects to value and price.

```
MDSlice = MarketDataSlice(MarketData = MD, TimeIndex = 1) #Create MarketData time slice
Value(T1, MDSlice)

## [1] 100

Value(T2, MDSlice)

## [1] 588230
```

Again, having a whole class just to slice the `MarketData` objects is a bit over the top for this simple example, but for more complicated examples, we may need to use the market data to create complicated objects with which to price, which we might want to defer till we actually want to compute a price.

For example, with swaps, the `MarketData` object will contain timeseries of all the swap rates etc. To price we'll need to create curve objects. If we have a large timeseries, then if we created a curve for each time point in the market data we might blow up the memory (stackoverflow!). However, if we leave curve construction to the `MarketDataSlice` class, then we can run algo's over timesteps and chuck out each curve when we go to the next step in the algo loop.

The above code chunk demonstrates how the **MarketData** and **Trade** objects interact. We can then combine trades into a **Portfolio** object.

```
TradeList = list(T1, T2)
Port1 = Portfolio(Name = "Port1", Trades = TradeList)
print(Port1) #Overloaded print for Portfolio

## [1] "Portfolio Name: Port1"

##   TradeName Type Notional
## 1      Cash Cash      100
## 2      AORD  Eq      100
```

Note in the above

- I've put the trades into a R list object. The way I've defined the Portfolio constructor, you have to do this, even for only a single Trade object. In robust production level code you would have type checking when constructing objects that would throw an error when wrong data types are given to constructor. In C++, the code won't even compile if data types don't match.
- I've overloaded 'print' for the portfolio class, to give a nice summary table. Should probably have a 'TradeDate' field.

Then similar to creating a `MarketDataSlice` and valuing a trade, we make a **PortfolioSlice** object and define functions **Price**, **Value** that apply to it.

```
time = 1 #Price time
PortfolioSlice = PortfolioSlice(MD, Port1, time) #Create Portfolio slice
Price(PortfolioSlice) #Query slice for Price

## Cash AORD
##    1 5882

Value(PortfolioSlice) #Query slice for Value

## Cash AORD
##    100 588230
```

I have slightly different interface for valuing Portfolio's and Trades. Have to think a bit about this in a final version.

### 1.3 TradeStrategy Classes and Running Strategy

The above section discusses the objects the TradeStrategy class will use, but so far we haven't actually defined anything that will actually run our strategy. The above classes are not specialised to running strategies and

can be used for other tasks, such as evaluating risk profiles for hedging, etc. This is what you want from OO. Reusability of code for logical chunks in different contexts.

To implement strategy we load the the TradeStrategy classes.

```
source("R/TradeStrategyClasses.R")
```

This file implements a class **TradeStrategy** with properties:

- You construct **TS**, a **TradeStrategy** object, from a MarketData object, an initial portfolio Portfolio object, and an initial time.
- There is a class (aka member) function "updSig" which looks at the data and Portfolio at a given time step and generates a trade signal for the period. **This function defines the trade strategy**. E.g. for MA momentum, you buy or sell depending on if there was a crossover at that point in time.
- There is an "updPortfolio" class function that adds trades to the portfolio depending on what the "updSig" function says. For MA momentum this is adding a buy/sell trade of a given notional at that date. For a delta hedging strategy it would be adding trades so that delta of portfolio = 0.
- There is a "runStrategy" class function, which loops the "updPortfolio" function of the MarketData object and collects the results.
- There is a "Results" data member for recording output.

Continuing with "Main.R", we see how the above class is used.

We initialise an empty portfolio with an initial cash trade of Notional = 0.

```
Cash = Trade(Name = "Cash", Type = "Cash", 0)
Port_MAtest = Portfolio("Port_MAtest", list(Cash))
```

Initialise TradeStrategy object, **TS1**, with market data object **MD** as above, and set the initial time=21 such that there are enough lags for both the MA series exist.

```
TS1 = TradeStrategy(Port_MAtest, MD, 21)
Time = TS1$CurrentTime
# Check MA series exist
TS1$MarketData$Data[(Time - 1):Time, c(4, 7, 8)]

##           AORD.Close  MAs  MA1
## 2008-02-29       5675 5728 5720
## 2008-03-03       5511 5712 5713
```

I've also picked an initial time so that there is a crossover right at the start. We can check this by running the "updSig" class function on our created **TS1** TradeStrategy object.

```
updSig(TS1)

## [1] "sell"
```

Now we check if **TS1** updates itself for the "sell" signal.

```
print(TS1$Portfolio) #Initial state: only cash in Portfolio
```

```
## [1] "Portfolio Name: Port_MAtest"

##   TradeName Type Notional
## 1      Cash Cash        0

TS1$CurrentTime #Current time index

## [1] 21
```

Run update and we should see a "sell" trade has occurred

```
TS1 = updatePortfolio(TS1) #Run portfolio update
print(TS1$Portfolio)

## [1] "Portfolio Name: Port_MAtest"

##   TradeName Type Notional
## 1      Cash Cash   551070
## 2    Trade2  Eq     -100

TS1$CurrentTime #Current time index has been advanced

## [1] 22
```

and we see that a new trade with with Notional = -100 has been added, ie a "sell", and that we've added the value of the 100 Notional of AORD to the cash Notional.

After the trade the total Value of the portfolio hasn't changed ( = 0), but when the "price" of AORD changes, with the price of Cash staying constant (=1), we will get PL for the portfolio.

So we see how **TS1** updates itself for one period. Now we can loop this proces over all the time periods and collect the results to see how the MA strategy performs. This is implemented with **runStrategy** class member.

```
TS1 = TradeStrategy(Port_MAtest, MD, 21) #Re-initialise back to start time
TS1 = runStrategy(TS1) #Run the strategy and collect results
```

The results get collected into a "Results" data member of the TS1 object.

```
head(TS1$Results, 5)

##   Time Value Signal
## 1 13941     0   sell
## 2 13942     0   hold
## 3 13943  3150   hold
## 4 13944  3910   hold
## 5 13945 -2120   hold
```

**Remark 4** Note that to run our strategy after creation of **MD** data object all we did was run the following four lines of code:

```
# Create Empty Portfolio
Cash = Trade(Name = "Cash", Type = "Cash", 0)
Port_MAtest = Portfolio("Port_MAtest", list(Cash))
```

```
# Initialise Strategy
TS1 = TradeStrategy(Port_MAtest, MD, 21)
# Run Strategy
TS1 = runStrategy(TS1)
```

Then we can look at the results.

The implementation details of the strategy are in the definitions of the functions "updSig" and "updPortfolio" in the StrategyTester class. If we want to add a strategy, we just add a new type of StrategyTester class by modifying the functions "updSig" and "updPortfolio" and run the same way.

If we had Strategy not implemented as OO, there's a good chance that code for the data import, strategy running, results analysis would get tangled up, and could be very confusing to add new strategies.

## 1.4 Plotting Results

The last bit of code is just displaying results. Just looks up results for timeseries of Value of portfolio and for when trades occur and does plots.

```
# Find where Trade Events occurred
TradeEvents = which(TS1$Results[, "Signal"] != "hold") #What indexes were trades at
TD = as.Date(TS1$Results$Time)[TradeEvents] #Convert to date stamp
TradeDate = index(TS1$MarketData$Data)[TradeEvents]
TradeSigs = TS1$Results[TradeEvents, "Signal"] #Get TradeSigs
```

merge Portfolio Value ts with price for plotting

```
ResZoo = zoo(TS1$Results$Value, order.by = as.Date(TS1$Results$Time))
ResZoo = merge(ResZoo, chartData)
```

Calculate where to draw trade lines, and format labeling of charts

```
TradeDatesZooPlot = domain[which(domain %in% TradeDate)]
numberTix = 20
spacing = floor(length(domain)/numberTix)
marks = domain[seq(1, length(domain), spacing)]
labs = format(marks, "%d/%b/%y")
```

Plot results in Figure 1.4 below (wherever LaTeX ends up putting it)

```
x11() #Open Graphic device on Linux
split.screen(figs = c(2, 1))

## [1] 1 2

screen(1)
plot.zoo(ResZoo[-1, c(5, 8, 9)], ylim = c(4500, 6200), screens = c(1, 1, 1),
  col = c("black", "green", "blue"), xaxt = "n", xlab = "Date", ylab = "Price And TA")
axis(side = 1, at = marks, labels = labs)
abline(v = TD, col = "red")
screen(2)
plot.zoo(ResZoo[-1, 1], ylim = c(-20000, 70000), screens = c(1), col = c("blue"),
  xaxt = "n", xlab = "Date", ylab = "Portfolio Value")
abline(v = TD, col = "red")
abline(h = 0, col = "black")
axis(side = 1, at = marks, labels = labs)
```



```
## Warning: calling par(new=TRUE) with no plot
## [1] 3 4
```

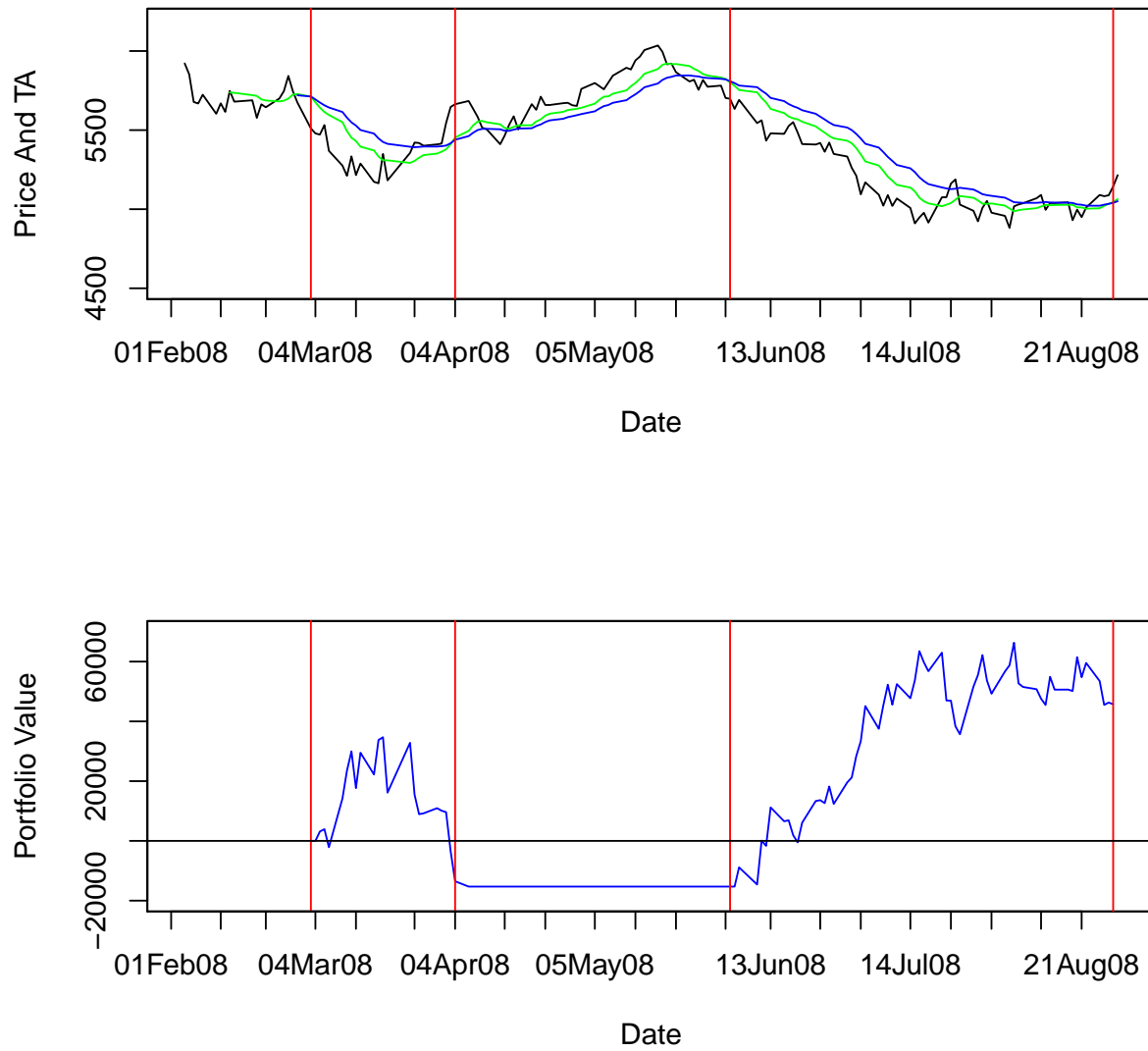


Figure 3: Top:AORD Timeseries and MA's, Bottom: Portfolio Values

## 2 Thoughts and Directions

Under construction. Some first thoughts:

- Does this look good to you as basic structure?
- Add realism. Bid/Offer, interest rates.
- Should be able add MA mean reversion strategy very easily
- Think doing an alpha PCA curve reversion would be pretty easy to do with RQuantlib package. Realistic would be harder on data front, as would need to include collateral data.
- Redo in Python.
- More post analysis of results. Add standard drawdown, Sharpe ratio calcs, etc. Can then compare easily across strats.
- What directions do we want to take this?
  - Something to research if/where different trade strategies are effective
  - Suggest medium term trade strats?
  - Suggest intraday trading - looking at tick data?
  - Use for automated trading?
- Not that we have to nail down, kinda cool creating and seeing what happens, but being on same page with a few big picture goals will keep momentum going.