Ruohan Wu
4727 Final Project
Trading System & Trading Strategy Development with Back Tester
Dec 19, 2018

"Coworker Memo"

REPORT

Realistic Implementations

I made a few revisions on the skeleton we built in homework 7 to make the new trading system more realistic.

1. Change all pointers to smart pointers and I change them to shared pointers specifically. And I also changed some functions prototypes from pointers involved to references involved. However, for functions *get_venue()* and *get_symbol(),* which return const pointers to the char array of venue and symbol, I did not change the return type to shared pointer because the pointers returned by these two functions will not give us the problem we worry about, such as double deleting.

2. Because the order amount I have in my database are all 100 hundred, I change the rejection threshold amount in the order manager from 1000 to 10, otherwise, all our orders will be rejected.

3. The equities I trade in the project is the IBM equity, so I include "IBM" in the tradable list of the order manager.

4. In order to make the system more realistic, which means the market simulator will not always fill an order whenever we pass it, we will introduce a probability p. Then with probability p, the order I sent to the simulator will be filled and with probability 1-p, the order I sent will be rejected. To realize this, I introduce a binary random variable, which get randomly assigned to the value of 0 or 1 with equal probability of 0.5. If the variable's value is 1, the simulator will fill an order that was passed to it and if the value is 0, the simulator not to fill an order.

Comparison on the result of R and C++

The results given by the R code and our C++ code mutually confirm each other, As can be seen in the following List.

R Output
Go Long
06:01 07:00 08:09 08:51 09:54 10:36 11:03 12:06 13:11 13:35 14:11
14:14 15:10 15:38 16:06 16:44 18:20 19:03 19:34 20:00
Go Short
06:38 07:12 08:31 09:51 10:03 10:42 11:27 12:54 13:21 13:53 14:13
14:40 15:25 15:46 16:18 16:59 18:40 19:33 19:38

C++ Output

Go Long

04:07 06:18 07:00 08:09 08:51 09:54 10:36 11:02 12:06 13:11 13:35 14:11
14:14 15:10 15:38 16:06 16:44 18:20 19:03 19:34 20:00

Go Short

04:11 05:03 06:38 07:12 08:31 09:51 10:03 10:42 11:27 12:54 13:21 13:53 14:13
14:40 15:25 15:46 16:18 16:59 18:40 19:33 19:38

We can see that the time for our trading strategies to push an order are basically the same in the outputs given by these two ways, which means both of these two ways of implementing the moving average strategies work well.

We have to acknowledge, on the other hand, there are some small discrepancies and they exist because of the following reasons.

1. The *rollmean* function I use in R to calculate the moving average is only defined when the data length is greater than the period of the moving average. Therefore, we do not have 5 minutes moving average until the 6-th minutes and do not have 20 minutes moving average until the 21-th minutes. However, in C++, we still define such quantity by taking whatever we have for now to calculate the moving average. This is why we have order placed by C++ at time such as 04:11 but not in R. We can fix this problem by manually calculating the 5-minutes moving average in the first 5 minutes and the 20-minutes moving average in the first 20 minutes, based on the formulas below.

$$MovingAvg_{20}[i] = \frac{sum(AvgPrice[1:i])}{i}, 1 \leq i \leq 20$$
$$MovingAvg_{5}[i] = \frac{sum(AvgPrice[1:i])}{i}, 1 \leq i \leq 5$$

2. The programming language R itself has a pitfall in the comparison of two very close numbers. (Refer to: https://stackoverflow.com/questions/9508518/why-are-these-numbers-not-equal) This general (language agnostic) reason basically says because of IEEE floating point arithmetic, the numbers are not represented exactly as they are in the computer, especially for some simple numbers. Therefore, when it comes to comparison, R will treat two numbers that should be equal as different. For example, at time 05:03, while the value of 5-minutes moving average and the 20-minutes average at its previous time spot (05:02) are both 121.6400, R will treat them as different numbers and R will return FALSE when we test if they are equal to each other using the "==" operator. This is the reason why the orders like the one placed by C++ at time 05:03 was missing for R.

   One possible way to solve this question is to use the *all.equal()* function of R. However, what *all.equal()* does is to introduce a tolerance level when we do comparison and such change in tolerance will influence other comparison operators. For example, when we use the *all.equal()* function, the "<" operator will then crash and will not give us the desired result. In order to make all operators to work properly, we need to check different tolerance levels, using methods such as cross-validation. However, this can be very time

consuming and costly.Based on the above explanation and reasoning, I will leave this discrepancy for now, but it can be fixed.

Results

The pnl we get when we have a 50% of rejection rate when an order is pushed to the market simulator is 121169, which indicates that our trading system and our trading strategy works reasonably well.

Analysis

PART I: Implementation in R

```
Data = read.csv(file="/Users/ruohanwu/Desktop/IBM.1min.TradesAndQuotes.20160128.csv",
                sep=",")
data = data.frame("Time" = Data$TimeBarStart)
data$BidPrice = Data$OpenBidPrice
data$AskPrice = Data$OpenAskPrice
data$Avg = (data$BidPrice + data$AskPrice) / 2
length = length(data$Avg)
data$MovingAvg_5[6:length] = rollmean(data$Avg, 6)
data$MovingAvg_20[21:length] = rollmean(data$Avg, 21)
```

```
# Signal of 1 means go long and Signal of 0 means go short
signal = array(dim = c(length,1))
signal[,1] = 0
for (i in 21:length){
  ind1 = (data$MovingAvg_5[i] > data$MovingAvg_20[i]
          & data$MovingAvg_5[i-1] <= data$MovingAvg_20[i-1])
  ind2 = (data$MovingAvg_5[i] < data$MovingAvg_20[i]
          & data$MovingAvg_5[i-1] >= data$MovingAvg_20[i-1])
  if (!is.na(ind1) & ind1){
    signal[i] = 1
  }
  if (!is.na(ind2) & ind2){
    signal[i] = -1
  }
}
data$signal = signal
```

```
#Time to go long
print(data$Time[which(data$signal == 1)])
```

```
##  [1] 06:01 07:00 08:09 08:51 09:54 10:36 11:03 12:06 13:11 13:35 14:11
## [12] 14:14 15:10 15:38 16:06 16:44 18:20 19:03 19:34 20:00
## 960 Levels: 04:01 04:02 04:03 04:04 04:05 04:06 04:07 04:08 04:09 ... 20:00
```

```
#Time to go short
print(data$Time[which(data$signal == -1)])
```

```
##  [1] 06:38 07:12 08:31 09:51 10:03 10:42 11:27 12:54 13:21 13:53 14:13
## [12] 14:40 15:25 15:46 16:18 16:59 18:40 19:33 19:38
## 960 Levels: 04:01 04:02 04:03 04:04 04:05 04:06 04:07 04:08 04:09 ... 20:00
```

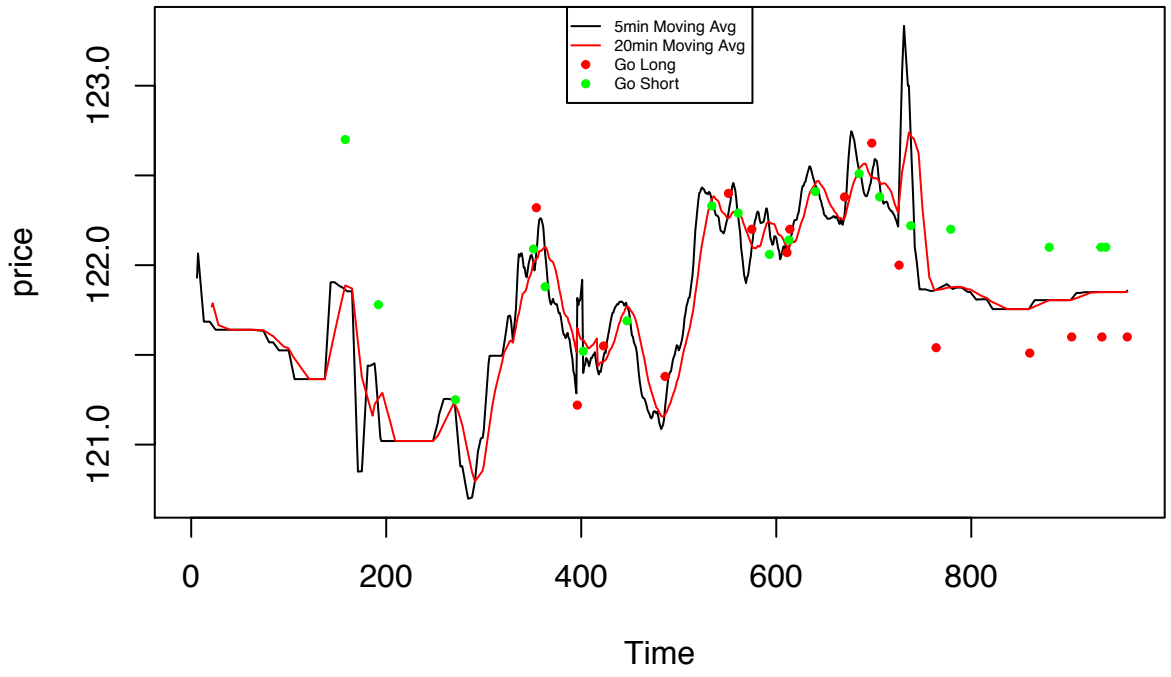
```
#Time to pass an order (go long or go short)
print(data$Time[which(data$signal == 1 | data$signal == -1)])
```

```
##  [1] 06:01 06:38 07:00 07:12 08:09 08:31 08:51 09:51 09:54 10:03 10:36
## [12] 10:42 11:03 11:27 12:06 12:54 13:11 13:21 13:35 13:53 14:11 14:13
## [23] 14:14 14:40 15:10 15:25 15:38 15:46 16:06 16:18 16:44 16:59 18:20
## [34] 18:40 19:03 19:33 19:34 19:38 20:00
## 960 Levels: 04:01 04:02 04:03 04:04 04:05 04:06 04:07 04:08 04:09 ... 20:00
```

Plot of the price change and the orders

The black curve is the 5-minutes moving average and the red curve is the 20-minutes moving average. The red dots denote the orders corresponding to our go-long signal. And the green dots are the orders to go short. All of their values are the prices.

```
plot(data$MovingAvg_5, type = 'l',xlab = "Time", ylab = "price")
lines(data$MovingAvg_20, col = 'red', type = 'l')
points(which(data$signal == 1),data$BidPrice[which(data$signal == 1)],
       pch = 19, cex = 0.5, col = 'red')
points(which(data$signal == -1),data$AskPrice[which(data$signal == -1)],
       pch = 19, cex = 0.5, col = 'green')
legend("top",cex=.5,legend=c("5min Moving Avg", "20min Moving Avg", "Go Long", "Go Short"),
       col=c("black", "red","red","green"), lty=c(1,1,0,0), pch = c(NA,NA,19,19))
```



PART II: Comparision with C++ Output

```
MyDataAvg <- read.csv(file="/Users/ruohanwu/Desktop/ieorhomework7/cmake-build-debug/averagePrice.csv",
                      header=FALSE, sep=",")
nRow = nrow(MyDataAvg)
MyDataOrder <- read.csv(file="/Users/ruohanwu/Desktop/ieorhomework7/cmake-build-debug/output.csv",
                        header=FALSE, sep=",")
BuyRow = (MyDataOrder[,4] == 1)
SellRow = (MyDataOrder[,4] == 0)
BuyOrder = MyDataOrder[BuyRow,]
SellOrder = MyDataOrder[SellRow,]
#Time to go long
print(BuyOrder$V1)
```

```
##  [1] 20160128 04:07:00 20160128 06:18:00 20160128 07:00:00
```

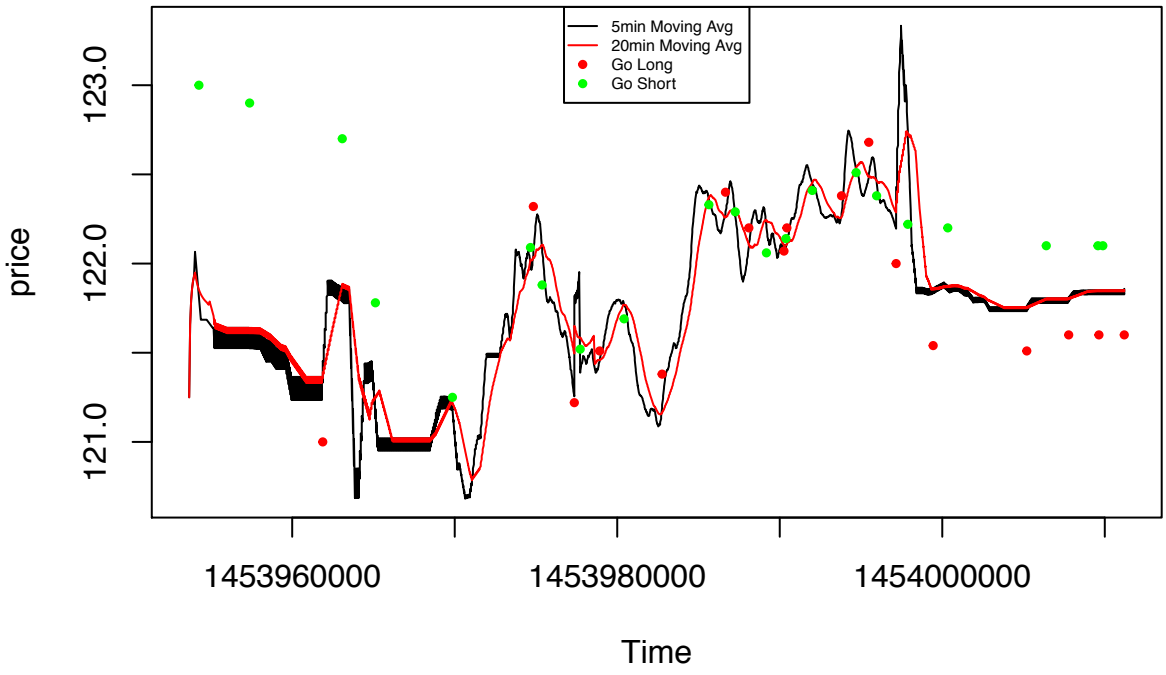
```
##  [4] 20160128 08:09:00 20160128 08:51:00 20160128 09:54:00
##  [7] 20160128 10:36:00 20160128 11:02:00 20160128 12:06:00
## [10] 20160128 13:11:00 20160128 13:35:00 20160128 14:11:00
## [13] 20160128 14:14:00 20160128 15:10:00 20160128 15:38:00
## [16] 20160128 16:06:00 20160128 16:44:00 20160128 18:20:00
## [19] 20160128 19:03:00 20160128 19:34:00 20160128 20:00:00
## 42 Levels: 20160128 04:07:00 20160128 04:11:00 ... 20160128 20:00:00
```

```
#Time to go short
print(SellOrder$V1)
```

```
##  [1] 20160128 04:11:00 20160128 05:03:00 20160128 06:38:00
##  [4] 20160128 07:12:00 20160128 08:31:00 20160128 09:51:00
##  [7] 20160128 10:03:00 20160128 10:42:00 20160128 11:27:00
## [10] 20160128 12:54:00 20160128 13:21:00 20160128 13:53:00
## [13] 20160128 14:13:00 20160128 14:40:00 20160128 15:25:00
## [16] 20160128 15:46:00 20160128 16:18:00 20160128 16:59:00
## [19] 20160128 18:40:00 20160128 19:33:00 20160128 19:38:00
## 42 Levels: 20160128 04:07:00 20160128 04:11:00 ... 20160128 20:00:00
```

When the simulator fill all the data we pass to the simulator, the plot will look like the following, where the red dots are orders to go long and the green dots are orders to go short.

```
plot(MyDataAvg[,1],MyDataAvg[,2],type = 'l',xlab = "Time", ylab = "price")
lines(MyDataAvg[,1],MyDataAvg[,3],type = 'l', col = 'red')
points(BuyOrder[,2],BuyOrder[,3], col = 'red',pch = 19, cex = 0.5)
points(SellOrder[,2],SellOrder[,3], col = 'green',pch = 19, cex = 0.5)
legend("top",cex=.5,legend=c("5min Moving Avg", "20min Moving Avg", "Go Long", "Go Short"),
       col=c("black", "red","red","green"), lty=c(1,1,0,0), pch = c(NA,NA,19,19))
```



We can see from the plot that the time given by the two different implementations to pass an order are the same.

If we want to make the simulator more realistic, which means the simulator will not always fill an order

whenever we pass it, we will introduce a probability p. So I introduce a binary random variable, which can give us the random probability p = 0.5 to fill an order that was passed to the simulator and 0.5 probability not to fill an order. We will plot such result in the following figure.

```
BuyRow_fill = (MyDataOrder[,4] == 1 & MyDataOrder[,5] == "FILLED")
BuyRow_rjct = (MyDataOrder[,4] == 1 & MyDataOrder[,5] == "REJECTED")
SellRow_fill = (MyDataOrder[,4] == 0 & MyDataOrder[,5] == "FILLED")
SellRow_rjct = (MyDataOrder[,4] == 0 & MyDataOrder[,5] == "REJECTED")
BuyOrder_fill = MyDataOrder[BuyRow_fill,]
BuyOrder_rjct = MyDataOrder[BuyRow_rjct,]
SellOrder_fill = MyDataOrder[SellRow_fill,]
SellOrder_rjct = MyDataOrder[SellRow_rjct,]
plot(MyDataAvg[,1],MyDataAvg[,2],type = 'l',xlab = "Time", ylab = "price")
lines(MyDataAvg[,1],MyDataAvg[,3],type = 'l', col = 'red')
points(BuyOrder_fill[,2],BuyOrder_fill[,3], col = 'red',pch = 19, cex = 0.5)
points(BuyOrder_rjct[,2],BuyOrder_rjct[,3], col = 'red',pch = 4, cex = 0.5)
points(SellOrder_fill[,2],SellOrder_fill[,3], col = 'green',pch = 19, cex = 0.5)
points(SellOrder_rjct[,2],SellOrder_rjct[,3], col = 'red',pch = 4, cex = 0.5)
legend("top",cex=.5,legend=c("5min Moving Avg", "20min Moving Avg",
                             "Filled Go Long","Rejected Go Long",
                             "Filled Go Short", "Rejected Go Short"),
       col=c("black", "red","red","red","green","green"),
       lty=c(1,1,0,0,0,0), pch = c(NA,NA,19,4,19,4))
```