Final Project

Objective

Our objective is to build an algorithm that can predict the movement of 1 month WTI Oil future contracts by using the spot price as a predictor. We will be using particle filters, a form of Sequential Monte Carlo, as well as a stochastic Poisson jump model that implements Bayesian components, to predict the price of oil. To test the effectiveness of our model, we have implemented an algorithm that simulates trading based on our predictions. Our goal is to develop a strategy that maximizes our profits over an approximately ten year time horizon. In doing so, we will incorporate aspects of: Sequential Monte Carlo, Particle Filters, Kalman Filters, Poisson jump processes, and Brownian motion.

Terminology

A future contract grants the holder the right to X units of some commodity for $\$Y$ in Z months. If held until expiry, the commodity will be delivered physically. Future contracts give individuals the ability to hedge their commodity position by locking in a future price, or speculate on the directional movement of the commodity.

The spot price of the commodity refers to the price a commodity can be bought and delivered at in the present.

Dataset

We took out data from Investing.com. We used historical data for Spot prices and 1 month future contract prices April 15th, 2010 to April 15th, 2020 to create our data set.

Initially, we noticed that the data did not align perfectly, with there being missing days between our spot price data and future price data. We removed all data points that were not shared between the two data sets.

https://www.investing.com/commodities/crude-oil-historical-data https://www.investing.com/currencies/wti-usd-commentary

Particle Filtering

Particle filtering is a sequential Monte-Carlo (MC) method that seeks to predict a hidden state variable ($\mathbf{x}$) from a series of observations ($\mathbf{y}$). $p(\mathbf{x}_0)$ is the initial state of the distribution, the transition equation is $p(\mathbf{x}_t|\mathbf{x}_{t-1})$, and $p(\mathbf{y}_t|\mathbf{x}_t)$ is the marginal distribution of the observation. Using Bayes' theorem, we derive an expression for $p(\mathbf{x}_{0:t}|\mathbf{y}_{1:t})$, the marginal distribution of the hidden state variable from the observations:

$$p(\mathbf{x}_{0:t}|\mathbf{y}_{1:t}) = \frac{p(\mathbf{y}_t|\mathbf{x}_t)p(\mathbf{x}_t|\mathbf{y}_{1:t-1})}{\int p(\mathbf{y}_t|\mathbf{x}_t)p(\mathbf{x}_t|\mathbf{y}_{1:t-1})d\mathbf{x}_t}$$

$$p(\mathbf{x}_{0:t}|\mathbf{y}_{1:t}) \propto p(\mathbf{y}_t|\mathbf{x}_t)p(\mathbf{x}_t|\mathbf{y}_{1:t-1})$$

We can also compute $p(\mathbf{x}_t|\mathbf{y}_{1:t})$ recursively via the marginal distribution:

$$p(\mathbf{x}_t|\mathbf{y}_{1:t}) = \int p(\mathbf{x}_t|\mathbf{x}_{t-1})p(\mathbf{x}_{t-1}|\mathbf{y}_{1:t-1})d\mathbf{x}_{t-1}$$

To find the expected value of $E[f(x_t)]$:

$$E[f(\mathbf{x}_t)] = \int f(\mathbf{x}_{0:t})p(\mathbf{x}_{0:t}|\mathbf{y}_{1:t})d\mathbf{x}_{0:t}$$

Do we need intermediate steps here?

$$E[f(\mathbf{x}_t)] = \frac{\int f(\mathbf{x}_{0:t})p(\mathbf{x}_{0:t}|\mathbf{y}_{1:t})d\mathbf{x}_{0:t}}{\int p(\mathbf{x}_{0:t}|\mathbf{y}_{1:t})d\mathbf{x}_{0:t}}$$

To evalute this integral, we introduce $w(x_{0:t})$, the importance weight. The importance weight is equal to:

$$w(x_{0:t}) = \frac{p(x_{0:t}|y_{1:t})}{\pi(x_{0:t}|y_{1:t})}$$

the importance sampling factor. The weight is very important in a particle filter algorithm as it allows us to pick which states are more likely than others and reduce down potential states before resampling. This weight, and subsequently, the importance sampling factor relies on (in our project with crude oil prices) the probability of a tomorrow's future price, given today's spot price divided by π, which is denoted by a factor that assesses different variables that influence the movement of tomorrow's future price.

$$E[f(\mathbf{x}_t)] = \frac{\int f(x_{0:t})w(x_{0:t})\pi(x_{0:t}|y_{1:t})dx_{0:t}}{\int w(x_{0:t})\pi(x_{0:t}|y_{1:t})dx_{0:t}}$$

Because we are operating under a MC framework, we can create an approximation for this integral:

$$E[f(\mathbf{x}_t)] \approx \frac{\sum_{i=1}^{i=N} f(x_{0:t}^{(i)})w(x_{0:t}^{(i)})}{\sum_{j=1}^{j=N} w(x_{0:t}^{(j)})} = \sum_{i=1}^{i=N} f(x_{0:t}^{(i)})w_t^{*(i)}$$

wIs there a reason the indices differ between the sums?

$$p(x_{0:t}|y_{1:t}) \propto p(y_t|x_{0:t}, y_{1:t-1})p(x_{0:t}|y_{1:t-1})$$

$$= p(y_t|x_t)p(x_t|x_{0:t-1}, y_{1:t-1})p(x_{0:t-1}|y_{1:t-1})$$
$$= p(y_t|x_t)p(x_t|x_{t-1})p(x_{0:t-1}|y_{1:t-1})$$

Recalling $w(x_{0:t}) = \frac{p(x_{0:t}|y_{1:t})}{\pi(x_{0:t}|y_{1:t})}$, we can plug in our value for $p(x_{0:t}|y_{1:t})$:

$$w_t^{*(i)} = \frac{p(y_t|x_t^{(i)})p(x_t^{(i)}|x_{t-1}^{(i)})p(x_{0:t-1}^{(i)}|y_{1:t-1})}{\pi(x_{0:t}^{(i)}|y_{1:t})}$$

However, we want the weights to update recursively. Defining $\pi(\cdot)$ recursively will help us redefine $w_t^{*(i)}$:

$$\pi(x_{0:t}|y_{1:t}) = \pi(x_t|x_{0:t-1}, y_{1:t})\pi(x_{0:t-1}|y_{1:t-1})$$

$$w_t^{*(i)} = \frac{p(y_t|x_t^{(i)})p(x_t^{(i)}|x_{t-1}^{(i)})}{\pi(x_t|x_{0:t-1}, y_{1:t})} \frac{p(x_{0:t-1}^{(i)}|y_{1:t-1})}{\pi(x_{0:t-1}|y_{1:t-1})}$$

$$w_t^{*(i)} = \frac{p(y_t|x_t^{(i)})p(x_t^{(i)}|x_{t-1}^{(i)})}{\pi(x_t|x_{0:t-1}, y_{1:t})} w_{t-1}^{*(i)}$$

Naive Alogrithm

We began our model by implementing a naive particle filter. Instead of predicting the price of the future contract, we predicted the volatility of the future contract. We did this because volatility is non-linear, for example, a 1 dollar price swing at 10 dollars is significantly larger than a 1 dollar price swing at 100 dollars.

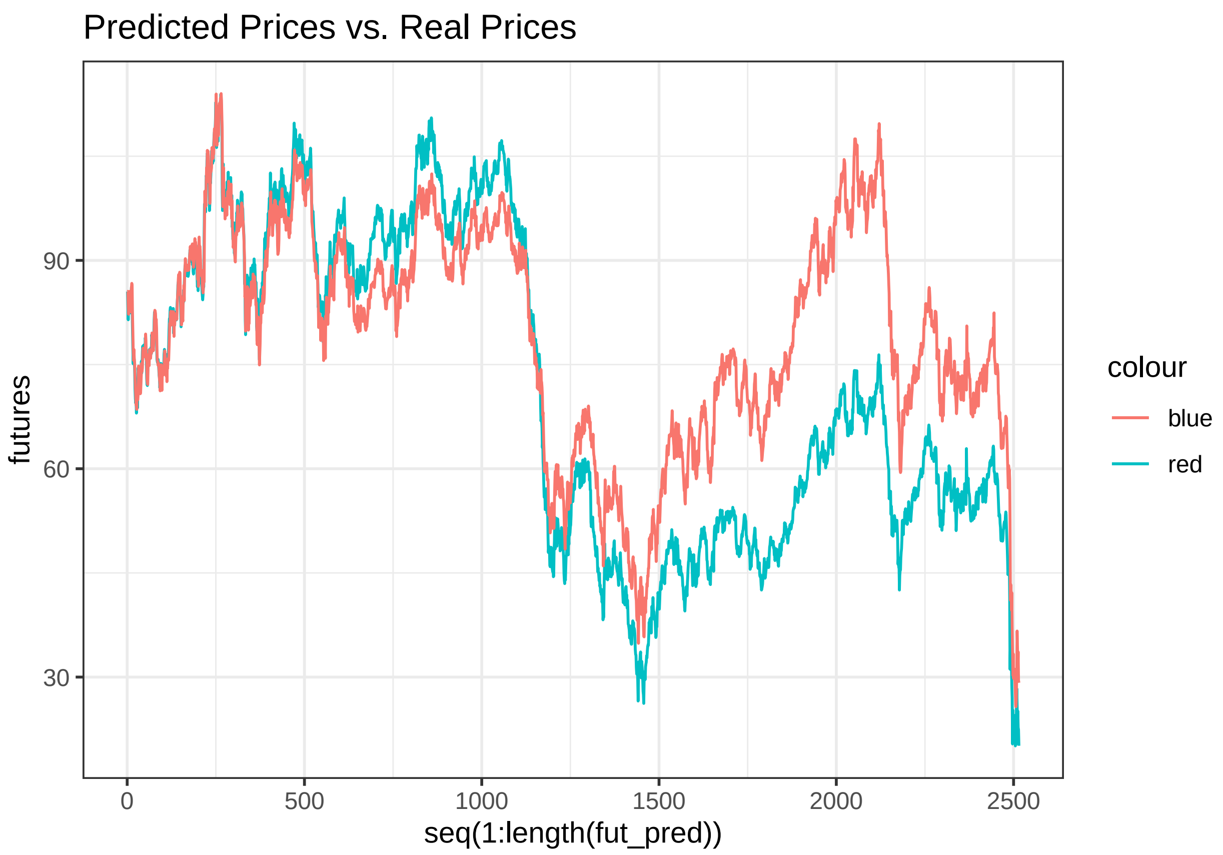
```r
set.seed(000)

library(readxl)
Oil_Data <- read_excel("oil_complete.xls")
oil <- na.omit(Oil_Data)

T <- nrow(oil)-1
x_true <- rep(NA, T)
obs <- rep(NA, T)

a <- oil$`Spot Vol`[1:T]
b <- oil$`Future Vol`[1:T]

sx <- sqrt(var(b))
sy <- sqrt(var(a))

for (t in seq(1, T)) {
  x_true[t] <- b[t]
  obs[t] <- a[t]
}


T <- length(obs)
N <- 5000
```

In the naive model, we assume a normal prior distribution.

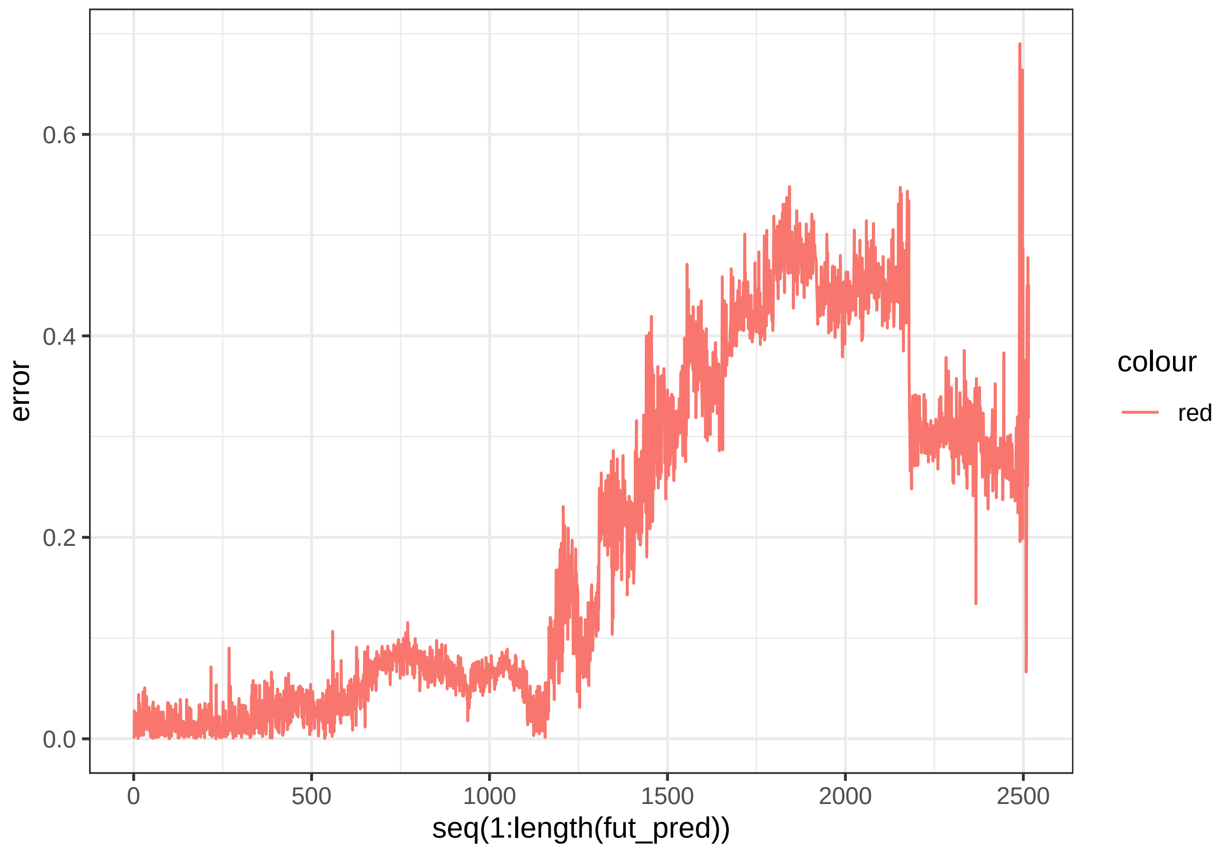
```r
x <- matrix(nrow =  N, ncol = T)
weights <- matrix(nrow =  N, ncol = T)

x[, 1] <- rt(N, 300) #Assume a normal prior

weights[, 1] <- dt((obs[1] - x[, 1])/sx, 300) # initial weight set
weights[, 1] <- weights[, 1] / sum(weights[, 1]) # initial weight normalized

x[, 1] <-
  sample(x[, 1],
         replace = TRUE,
         size = N,
         prob = weights[, 1])

for (t in seq(2, T)) {

  x[, t] <- rt(N - x[, t - 1], 300) #sample x from previous x

  weights[, t] <- dt((obs[t] - x[, t])/sx, 300) #get weights from obsveration
  weights[, t] <- weights[, t] / sum(weights[, t]) #normalize weights

  x[, t] <- #resample x using the weights
```

```
    sample(x[, t],
           replace = TRUE,
           size = N,
           prob = weights[, t])
}

fut_pred <- oil$`Future Price`[1]
x_mean <- apply(x, 2, mean) #mean of predictions found

for (i in 1:ncol(x)) { #volatility converted to prices

  new_pred <- fut_pred[length(fut_pred)] *(1+x_mean[i])
  fut_pred <- c(fut_pred, new_pred)

}

futures <- oil$`Future Price`[1:length(fut_pred)]

fut_naive <- NULL

fut_naive <- cbind(fut_naive, Real = futures)
fut_naive <- cbind(fut_naive, Predicted = fut_pred)
```

```
# Real prices vs. Predictions

p1 <- ggplot() + geom_line(aes(y = futures, x = seq(1:length(fut_pred)), colour = "red")) + geom_line(a

error <- abs(futures - fut_pred) / futures

# Absolute error as percentage of price

p2 <-
  ggplot() + geom_line(aes(
    y = error,
    x = seq(1:length(fut_pred)),
    colour = "red"
  ))


p1 + ggtitle("Predicted Prices vs. Real Prices")
```


Predicted Prices vs. Real Prices
futures
90
60
30
0
500
1000
1500
2000
2500
seq(1:length(fut_pred))
colour
blue
red

```
p2
```



Our naive model, as seen in the graph above, is mediocre at predicting oil futures prices, and the predictions seem to be getting worse as time passes. This makes sense, considering that there's a big oil sell-off in the back end of our data. We think there are a couple reasons why this model might not be very successful: (1) our prior is normally drawn into the matrix X, and when looking at the distribution of the data, it does not seem to be normally distributed; and (2) the weights assigned to predict the future price is based off of volatility, which can boost the direction of prediction when looking at larger shifts in the market. To fix these, we will first draw our prior from a log-normal distribution and assign weightages in a different way.

Poisson Jump Model

```
oil_data <- read_xls("oil_complete.xls") #2589 rows

names(oil_data)[2] <- "fut_price"
names(oil_data)[3] <- "spot_price"
names(oil_data)[4] <- "fut_vol"
names(oil_data)[5] <- "spot_vol"
```

```
genNoise <- function(data, size) {

  i <- 1
  hurst <- NULL

  while (i < length(data) - size - 1) {
    j <- i + size
    hurst <- c(hurst, (2 - fd.estim.hallwood(data[i:j])$fd))
    i <- i + 1

  }

  noise <- NULL

  for (i in 1:length(hurst)) {
    if (hurst[i] < 0)
      noise <-
        c(noise, fbm(0, 1)[1])
    else
      noise <- c(noise, fbm(hurst[i], 1)[1])

  }

  return (noise)

}
```

```
genJumps <- function(T, la, j_var) {

  jump_count <- rpois(1, T * la) # in T units, how many jumps will there be

  x <- runif(jump_count, 0, 1)
  case_when(x < 0.5 ~ -1, x > 0.5 ~ 1)

  jump_vals <- x*rnorm(jump_count, 0, j_var) # what will the value of the jumps be

  # determine time of poisson jumps

  units <- runif(jump_count, 1, 100)
  units <- round (T * units / sum(units), 0)
  if (sum(units) != T) units[length(units)] <- (units[length(units)] + T - sum(units))
  jump_times <- sample(units)

  final_data <- c()
  pjumps <- c()

  for (i in 1:length(jump_times)) {

    final_data <- c(final_data, rep(jump_vals[i], jump_times[i]))
    pjumps <- c(pjumps, rep(jump_vals[i], jump_times[i]))

  }

  final_data <- final_data

  return (final_data)

}
```

To improve our model, we looked into implementing Poisson jump shifts. The algorithm for simulating Poisson jumps is simple. First, to determine how many jumps there will be, N_T is found:

$$N_T \sim Po(T\lambda)$$

Then, simulate N_T uniforms to determine the time between jumps:

$$p_{1\ldots N_T} \sim U(0, a)$$

Where a is some arbitrary value. However, the total of the lengths between the jumps must be equal to T. Therefore, we normalize the data to sum to T.

$$U_i = \frac{Tp_i}{\sum_{i=1}^{N_T} p_i}$$

This ensures the following sum holds:

$$\sum_{i=1}^{N_T} U_i = T$$

Addionallity, the total time elapsed before the next jump is:

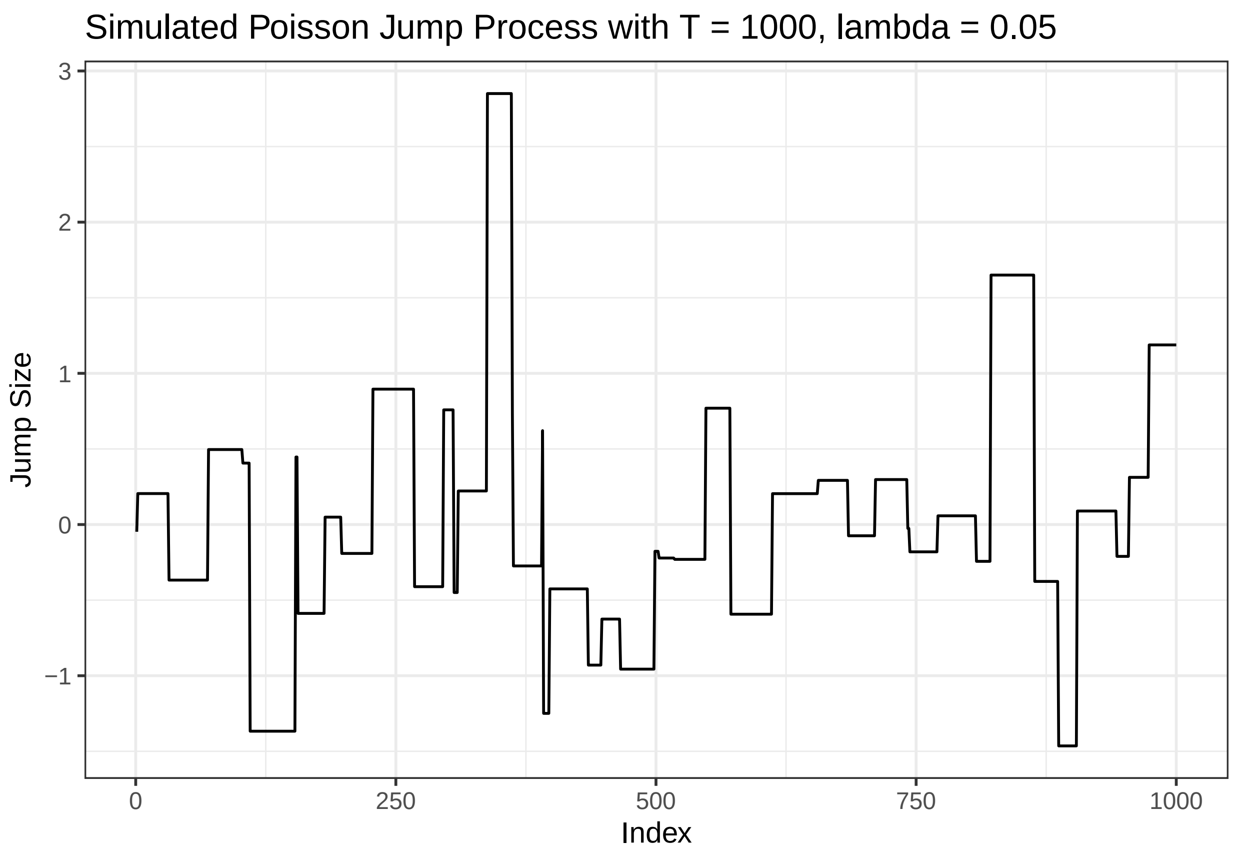
$$F(U_i) = \sum_{j=1}^{i} U_j$$

To actually compute this, whole integers are needed. Because rounding may make $\sum_{i=0}^{N_t} P_i \neq T$, a buffer is added to the last value equal to $B = T - \sum_{i=0}^{N_t} Y_i$ to ensure the equivalence holds. We then sampled all the values to randomize the order, so the buffer wasn't always at the end. Finally we simulate the size of the jumps:

$$Y_{1\ldots T} \sim N(\mu, \sigma^2)$$

This is what a example of a sample may look like for $T = 1000$, $\lambda = 0.05$

```
example <- genJumps(1000, 0.05, 1)
ggplot() + geom_line(aes(x = seq(1:1000), y = example)) + ggtitle("Simulated Poisson Jump Process with '
```



Because the price of oil is subject to random political events that may quickly change the price, adding a poisson jump process to our simulation will help us to form confidence intervals of how we expect the price to perform over a period of time. After we implement the jumps, we must account for the intraday noise, or Brownian motion. For simple Brownian Motion, $B_i \sim N(0, 1)$, with

$$S_n = \sum_{i=1}^{n} B_i$$

$$E(S_n) = \sqrt{n}$$

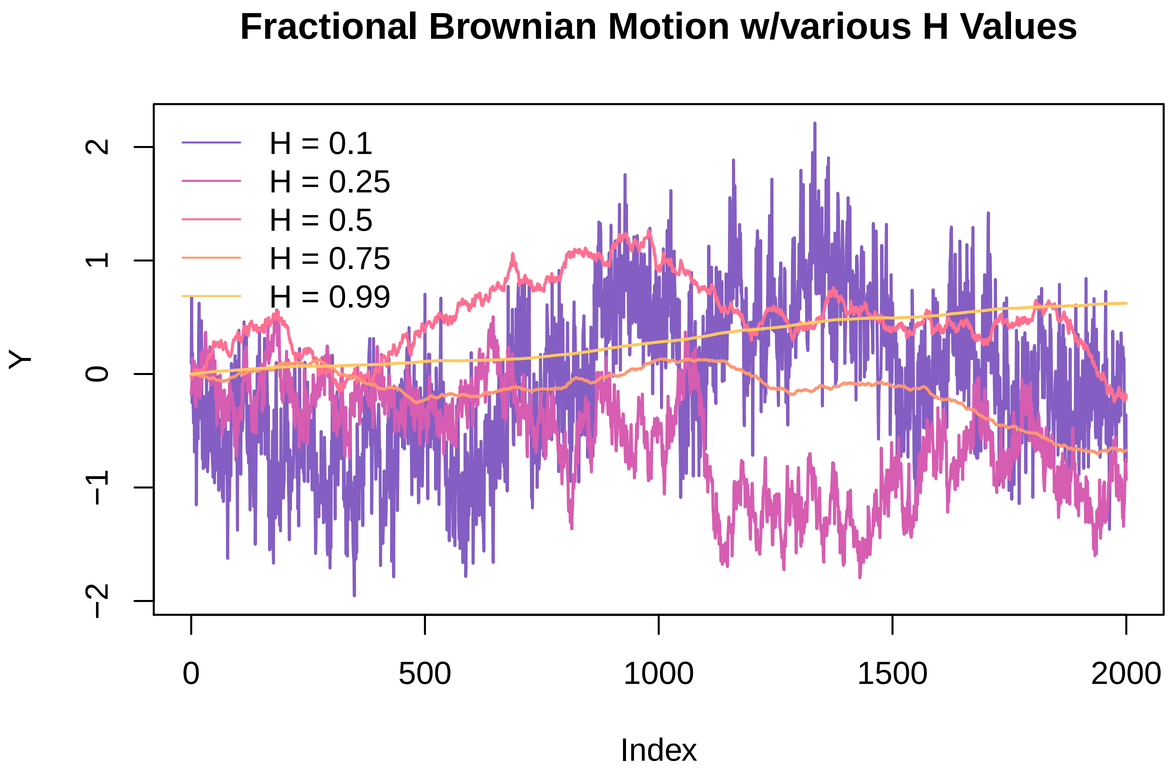
Therefore, the interpretation is that over n samples, the expected distance from the origin is $\sqrt{n}$. However, for oil, we believed this was an oversimplification of the price behavior. Oil is very connected to macroeconomic forces, creating cyclicality in supply and demand. A company like Apple may be expected to grow on average for 40 years, but this assumption does not hold for oil, thus we believe the Brownian Motion used to simulate our volatility should reflect that. In our research, we came across Hurst Exponents. For some sum S_n of arbitrary random variables Y_i

$$S_n = \sum_{i=1}^{n} B_i$$

$$E(S_n) = n^H$$

The Hurst Exponent describes the degree of dispersion a time series exercises, with the domain of $H : [0, 1]$. $H = 0.5$ would be standard Brownian Motion. $H = 1$ would dsuggest the series will move far from the origin, meaning the series is trend following. $H = 0$ would suggest the series is mean reverting or returns to the starting point over time.

```r
set.seed(10)

df <- NULL

n <- 2000

h1 <- fbm(0.1, n)
h2 <- fbm(0.25, n)
h3 <- fbm(0.5, n)
h4 <- fbm(0.75, n)
h5 <- fbm(0.99, n)


ts.plot(main = "Fractional Brownian Motion w/various H Values",ts(h1, start = c(0,1)),
        ts(h2, start = c(0,1)),
        ts(h3, start = c(0,1)),
        ts(h4, start = c(0,1)),
        ts(h5, start = c(0,1)),
    col = c('#845EC2', '#D65DB1', '#FF6F91', '#FF9671', '#FFC75F'),
    xlab = "Index",
    ylab = "Y",
    lwd = 1.5)

legend("topleft", bty="n", lty=c(1,1), col=c('#845EC2', '#D65DB1', '#FF6F91', '#FF9671', '#FFC75F'),
       legend=c("H = 0.1", "H = 0.25", "H = 0.5", "H = 0.75 ", "H = 0.99"))
```



The Hurst exponent is found empirically by looking at how self-similar the data is. First, we found the Hurst Exponent of the last 100 trading days, using the library fractaldim. Then, we generated a sample of random data with a Hurst exponent equal to the one found empirically. To do this, we used the somebm library. In conclusion, our simulated price at time T is equal to the sum of all previous noise plus all previous poisson jumps plus the initial starting value:

$$X_T = X_1 + \sum_{t=1}^{T} B_H(t) + \sum_{t=1}^{T} \left[\sum_{i=0}^{N_T} Y_i \, \mathbf{1}_{F(U_i) \leq t} \right]$$

However, the definition of fractional Browian motion is:

$$B_H(t) = \int_0^t \frac{1}{\Gamma(H + \frac{1}{2})} (t - s)^{H - \frac{1}{2}} dB(s)$$

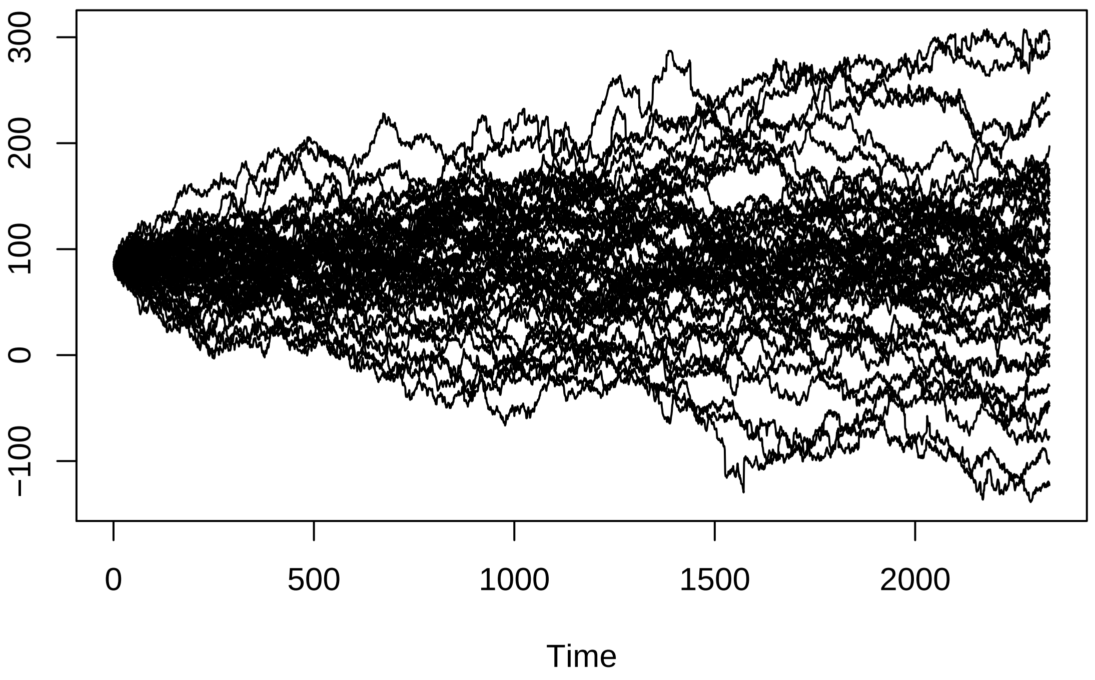
$B_H(t)$ sums up a given process with given H up to time t. Because we H changes at each t value, we correct the sum to:

$$X_T = X_1 + \sum_{t=1}^{T} B_H(\Delta t) + \sum_{t=1}^{T} \left[\sum_{i=0}^{N_T} Y_i \, \mathbf{1}_{F(U_i) \leq t} \right]$$

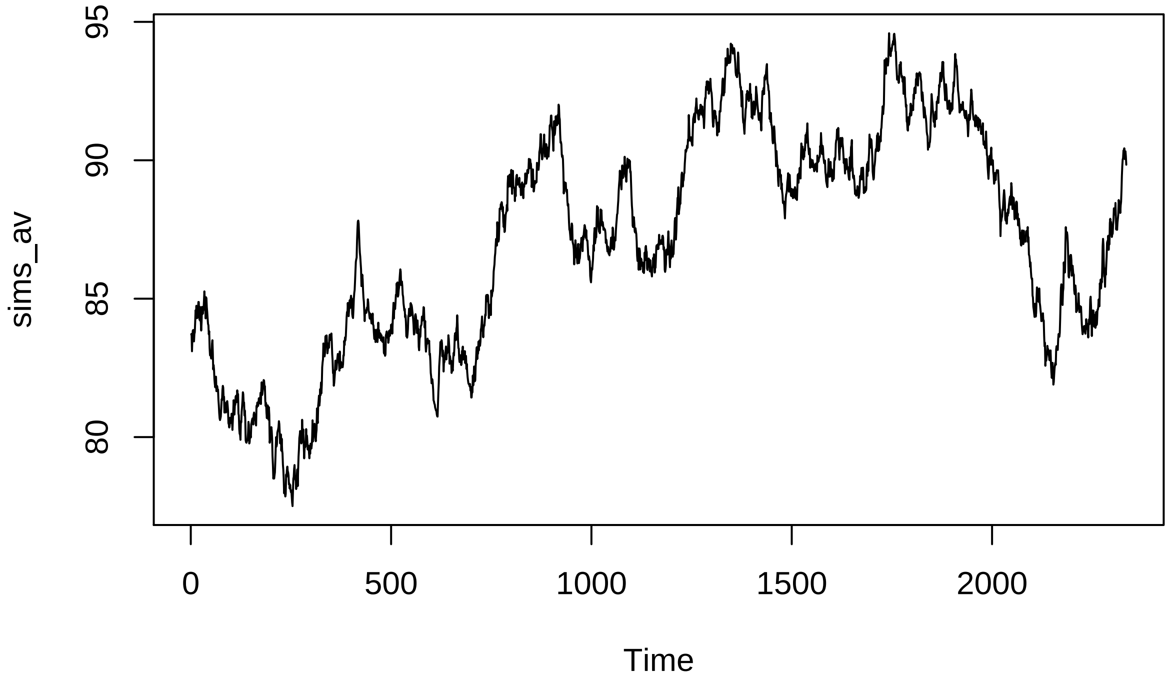
We then ran 50 simulations and averaged the results to create our predictions.

```r
la_real <- 0.03592
T <- nrow(oil_data)
init <- 85.5

fut_jumps <- filter(oil_data, abs(fut_vol) > 0.05)
fj_vol <- abs(fut_jumps$fut_vol)
j_mu <- mean(fj_vol)
j_var <- var(fj_vol)

run <- function() {

  jumps <- exp(genJumps(T, la_real, 0.05))
  noise <- cumsum(genNoise(oil_data$fut_price, 252))

  b <- length(jumps) - length(noise) + 1
  jumps <- jumps[b:length(jumps)]

  sim <- init + jumps * noise

  sims <- cbind(sims, sim)

}
```

```r
sims <- read_xls("sims.xls")[-1]
sims_av <- NULL

for (i in 1:nrow(sims)) {

  sims_av <- c(sims_av, sum(sims[i,][2:ncol(sims)])/ncol(sims))

}

ts.plot(sims)
```



```r
ts.plot(sims_av)
```



Trading Algorithm

To test our particle filter algorithm, we built a simple trading agorithm to see how much money we would make in the markets. We start with some initial amount of cash. Each day, we make our prediction for the next day. If our predicted value if greater than the current price of the future contact, we buy a contract at today's price. The next day, we sell if the price matches our prediction. If there is an 8% drop, we decide to buy more.

```
prof_ret <- function(range) {
init = oil[1,2]
profit = 0
sellpt = 0
current = init
hold = TRUE
for (i in 1:nrow(oil)){
  if (current < fut_pred[i+1] && hold && abs(x_mean[i+1]) <= range){
    profit = profit + (oil[i+1,2] - current)
    sellpt = oil[i+1, 2]
    hold = FALSE
  }
  #after repeated trials, 8% drop seems to be the best point to rebuy
  if (!hold && oil[i, 2] <= 0.92*sellpt){
    current = oil[i, 2]
    hold = TRUE
  }
}
tot = profit*1000 #minimum order of 1000 barrels to trade futures
investment = init*1000
roi = tot/investment
roi
tot

return(roi)
}
```

We want to sample through potential ranges to stop trading. This ensures that the model primarily focuses on trading when volatility is small and makes safer trades.

top = 0 range = 0 for (i in seq(from = 0.02, to = 0.20, by = 0.005)){ if (prof_ret(i) > top){ top = prof_ret(i) range = i } } range # This code above took forever to run each time, but it yielded 0.035 as the desired volatility to stop trading

Shorting Algorithm

```
profit <- NULL
hold = FALSE
for (i in 1:nrow(oil)){
  if (x_mean[i+1] < 0 && i <= 2414){
    shortval = oil[i,2]
    hold = FALSE
  }
  if (!hold && x_mean[i+1] > 0){
    profit = rbind(profit, (shortval - oil[i,2]))
    hold = TRUE
  }
}
tot = sum(profit)
tot*1000
```

```
## [1] 610700
```

```
count = 0
for (i in 1:635){
  if (profit[i,] < 0){
    count = count +1
  }
}
count
```

```
## [1] 61
```

```
((tot*1000 + 85510)/85510)^(1/10)-1
```

```
## [1] 0.2333102
```

We see that by just shorting, we can make a lot more money. This algorithm relies on microtransactions: each time the predicted future volatility is negative, we short and hold that position until we predict the volatility to go up again. Overall, we would make $610,700 over a period of 10 years using shorts and a particle filter. This yields a 23.33% return annually, which is somewhat competitive among financial institutions.

```
init = oil[1,2]
profit = 0
sellpt = 0
current = init
hold = TRUE
for (i in 1:nrow(oil)){
  if (current < fut_pred[i+1] && hold && abs(x_mean[i+1]) <= 0.035){
    profit = profit + (oil[i+1,2] - current)
    sellpt = oil[i+1, 2]
    hold = FALSE
  }
  #after repeated trials, 8% drop seems to be the best point to rebuy
  if (!hold && oil[i, 2] <= 0.92*sellpt){
    current = oil[i, 2]
    hold = TRUE
  }
}
tot = profit*1000 #minimum order of 1000 barrels to trade futures
tot
```

```
##   Future Price
## 1         9850
```

```
profit
```

```
##   Future Price
## 1         9.85
```

```
((tot + 85510)/85510)^(1/10)-1
```

```
##   Future Price
## 1   0.01096224
```

Using a buying strategy would yield $9850 over the past 10 years of oil futures trading. With an initial investment of $85,510 on the first day of trading, our model yields an 11.52% overall return on investment and an annual return of 1.09%, which is pretty bad.

Discussion of Stochastic Poisson Jumps Method

Unlike the particle filter method, the Poisson Jump method predicted the price of oil as opposed to the volatility of oil. This model used an empirical Bayes to estimate how often jumps occured in the price of oil, and to calculate the Hurst exponent and simulate the fractional Brownian motion. However, the model is not a pure Bayesian model because we are not updating a posterior and finding an analogous distribution. Because of this, we had hypothesized that this model would perform worse than the particle filter in calculating the volatility or price on a day by day basis.

In order to test our model, we simulated 50 10 year periods. In our generated data, we found the model's price estimates to the actual prices to be ineffective, with a coefficient of correlation of -61%. Furthermore, the correlation of the volatility was 2.3%. Because of this, we decided not to build a trading algorithm that would trade off of these predictions and instead focus on the particle filter model. We hypothesize that this model may work better for other types of financial assets. Oil is very volatile, and a lot of the volatility is determined by short term trends of imbalances in global supply and demand. Because our Brownian motion looked at the previous 252 days to simulate its noise, the data in the immediate past was not very influential in forming our predictions. Because of this, we think assets that may exhibit longer term trends, or be exposed to less political volatility, may be more accurately predicted using this model.

Additionally, we see two other problems with this model. First, because the previous data points have little weight in the next day's prediction, very strong momentum develops in our predictions, and we ended up having some simulations with extremely negative prices or extremely high prices. Another drawback to this model was how computationally expensive it was. Simulating 50 iterations took close to 25 minutes to run, and it made it difficult to run with larger sample sizes or optimize parameters in the model.

Ideally, if we had more time, we would like to integrate Poisson jumps into the particle filter model. We believe this would better help to predict big jumps in the price of oil. We found it difficult to integrate this into our filter because the jumps dealt with price and the filter predicted volatility, and think greater exploration would lead to a more robust model.

Trading Algorithm and Comparison of Profits for Both Methods

In order to compare the profits gained following both methods, we designed two different trading algorithms that either followed a buying (long position) or selling (short position) strategy. These were primarily built out using for-loops that iterated through and used the predicted volatility the next day to make a decision in the current day, whether that was buying or shorting a future. In terms of buying, we encountered a couple optimization quirks, such as only trading when volatility was below 3.5% to avoid those sudden jumps that could lead to losses. We also discovered that after selling, the optimal time to re-buy the future was an 8% drop from the previous sell point. These are interesting financial quirks that we could do some future analysis on. This algorithm yielded a 1.09% annual return, which is pretty low, but showed that our algorithm was effective at predicting volatility. Knowing this fact, we pivoted to another trading algorithm that exclusively shorted the futures. In this algorithm, once sold, the future was not rebought until the predicted volatility the next day was positive. Our algorithm was incredibly successful at predicting these volatility switches, so we were able to turn a profit 90.4% of the time the algorithm decided to short. Overall, we found a 23.33% annual return over the period of 10 years, which is a pretty decent number. Although our SMC model was not the most successful at predicting jumps or the magnitude of movement, it was still very effective at predicting volatility switches and on a very simple level, understanding if the market was going to move up or down the next day.

What We Learned

We enjoyed challenging ourselves by learning material similar, but not taught to in class. While the Poisson jump model was ultimately ineffective, by learning two different modeling techniques, we were able to see the pros and cons of each model, and better understand what would make a more effective model to predict oil prices. Furthermore, the Poisson model is a great example of what is more complicated is not necessarily better. We took some creative risk in trying to integrate the Hurst exponent into our Brownian motion, and while frustrating that the model performed so poorly, we were able to conclude with some degree of confidence that the particle filter method works better than the Poisson jump model, and use those results to build a trading strategy with strong performance.

Bibliography

https://users.aalto.fi/~ssarkka/course_k2012/handout6.pdf

https://rpubs.com/awellis/180442 (naive code)