There's a really good video by the interminably talented Jon Bois about Barry Bonds' 2004 season, where Bonds achieved the highest On Base Percentage (0.609) in baseball history. It's called "What if Barry Bonds had played without a baseball bat?", a premise inspired by how unusually often Barry Bonds was walked that season. It unfolds as follows:

- Jon introduces his premise: Take every 2004 at-bat for Bonds, and "take away his bat" - meaning the same pitches get thrown at him, Barry just can't hit any of them. Walks are still walks, strikes are still strikes, if he gets hit by a pitch he still gets beaned, but hits (including fouls) become either walks or strikes. If the pitch is in the strike zone, Bonds retroactively gets a strike, if it's out he gets a walk.
- Jon introduces his methodology: He initially planned to use PitchFX, which tracks the exact position of the ball on every major league pitch, but this was not introduced until 2006. Instead, he uses data from Retrosheet, which tracks the outcome of each pitch. In cases where not having a bat leads to more pitches than there were in real life, Jon simulates the pitch by drawing from the average of pitches that season. For example, 19.1% of the fouls Bonds hit were within the strike zone, so if the foul would've affected the outcome, Jon generates a random number between 1 and 1000, and anything above 191 counts the at-bat as a strike.
- Jon reveals the results: Even without a bat, Bonds would still have had an OBP of 0.608 in 2004, still the best in baseball history.

I've been a massive fan of Jon Bois for years, and this video has always irked me, because he didn't have to simulate the pitches! In the example above, it would've been much easier and more accurate to give Bonds 0.191 "bases" instead of doing an all or nothing simulation. He mentions at the end of the video that re-simulating may have given him a different result. I don't think he needs to simulate this at all. Maybe I'm wrong - Bois is a pretty smart guy - but I at least want to find out the hard way.

My goals are as follows:

1. Develop better bonds-without-a-bat methodology
2. Apply it to the retrosheet data
3. Recalculate Bonds' batless OBS

PART 1: Methodology

I have immediately run into a snag: a hit is a hit, but what about multiple fouls?

Let's think it through:

1. Take the hitting line CCBBBFB (strike strike ball ball ball foul ball, originally a walk). That's a critical foul, and our result is obvious for reasons outline above: 0.191 bases for that at-bat.

2. Take the hitting line CCBBFFBB, also originally a walk. This is harder. There's an 80.9% chance that that's a strikeout at pitch 5, and a 19.1% chance that it continues. The same is true for pitch 6. That means that we'll be crediting bonds with $0.191^2 = 0.036481$ bases for this at bat.
3. We can apply the same logic to multiple fouls in similar situations.

Some observations:

1. We only need to count how often he gets on base, and Barry only gets on base if he gets walked or hit by the pitch. We'll never add a HBP, only take them away. Really this is an exercise in counting walks.
2. We could use a weighting system to do these calculations, perhaps. The problem is that order matters. The first step here is to develop a formula to calculate, based on any hitting line, what the hitting outcome is.

Perhaps we're getting ahead of ourselves. Let's get the data first.

Part 2: Getting The Data

Retrosheet is a pain in the ass.

Retrosheet presents their data in a very compact format, which I'm sure is great for their purposes, but makes it a pain to work with. Instead of a standard csv, they present their data in custom files with custom meanings. They call them Event files. Each event file only contains each team's home games, meaning I will have to compile 30 datasets, with irregular data patterns, into one. I dislike this immensely.

Stackoverflow says that I can get around the jaggedness of the .csvs by naming the columns in the import. That makes it easier. The problem now will be getting the filenames.

I started by listing all the filenames out, but I figure there has to be a way to do it in code. Looking into it further, the setwd() and list.files() functions seem to be my best bet. For list files, I have to use the correct regular expression.

Except I don't actually lmao - I managed to do it with this:

```
1  library(readr)
2  library(tidyverse)
3
4  setwd("BarryBonds2004/")
5  retrofile <- list.files(pattern="")
6
7  testsheet <- retrofile %>% map_df(~read_csv(.,
8                        col_names = c("type","a","b","player","c","hitting")))
9
```

One day I will learn regular expressions. Not today. Also this read more than 350k lines into a single object.

A couple more lines of code and I've gotten every 2004 at-bat from Barry Bonds in one sheet.

```
testsheet <- retrofile %>% map_df(~read_csv(.,
                      col_names = c("type","a","b","player","c","hitting")))
bondsheet <- filter(testsheet,player == "bondb001")
bondsheet <- filter(bondsheet,is.na(hitting) == FALSE)

```

Cool! That was easy.

Retrosheet is great. I've never said anything bad about it, ever.

Part 3: Methodology again

After seeing the full list of plate appearances, we begin to see another reason Bois simulated the plate appearances: Bonds hit a LOT. When he hit, he usually cut the plate appearance short. He had 59 plate appearances where he walked to the plate and hit on the first pitch. There has to be a way to calculate Bonds' expected base value from that plate appearance.

We have some percentages from Bois' video:

- Pitches Bonds swung at (for our purposes, F fouls, X hits, and L foul bunts) were in the strike zone 80.9% of the time and outside 19.1% of the time
- All pitches thrown at bonds were in the strike zone 41.3% of the time and out 58.7%.

So we're going to want a script that does a couple things. First:

- We're going to want to clean the data so that every pitch is either a ball, strike, HBP, or a swung-at pitch.
- We're going to want to analyze the plate appearance based on normal baseball rules. We'll count the strikes and balls in order. If we get 3 strikes, we assign what I'm calling an earned-base-value (EBV) of 0 since he didn't get on base, and 4 balls get an EBV of 1 since he did get on base, one time.
- We can do this via multiplication. If the first terminal result is three strikes, we multiply the EBV by 0. If we get three balls, we multiply it by 1. That'll make it easier to deal with partial EBVs.
- If he got hit by a pitch, he gets an EBV of 1. None of his 9 HBP outcomes would've been changed if he didn't have a bat.
- If he gets to a hit, we start with an EBV of 0.191 (to represent the 19.1% of swung-at pitches outside of the strike zone) and add a ball. If we're not at 4 balls, we then multiply that by 0.587 and add a ball until we get 4 balls.

So good so far. What happens when we get to a foul or foul bunt? Take, for example, this fucking nightmare of a plate appearance from the real data: BFFFBBFFFFX. We don't have to deal with all of this - we're going to get a result of 3 balls or 4 strikes in 7 pitches. So, BFFFBBF.

My thought is to replace the original row with two rows:

- BCFFBBF, and we multiply the EBV by 0.809
- BBFFBBF, and we multiply the EBV by 0.191

Then, each row would itself be replaced by two rows, each EBV would be multiplied by the right percentage, so on and so forth. This way I will never have to learn how to do permutations.

Part 4: Cleaning the data

There are some things in a Retrosheet hitting line we don't care about. Namely, anything that isn't a batting interaction. Retrosheet helpfully explains what we don't want here: https://www.retrosheet.org/eventfile.htm

We also only want the first 7 pitches, and we want an EBV column. We can start each EBV at 1, since we'll be multiplying by values smaller than 1 to calculate the correct EBV. We also want exactly 5 outcomes of each pitch:

- Strike (we'll note this as C)
- Ball (B)
- Hit (X), which will trigger the hit procedure
- Foul (F), which will trigger the foul procedure
- HBP (H), which will immediately award Bonds an EBV of 1

Let's clean the data into having only these 5 values. Note that Bonds has no other pitch values other than the 5 above and T, S, L, and I.

```
same_pitch_types <- function(hitting){
  for (i in 1:length(hitting)){
    for (j in 1:nrow(hitting)){
      if(hitting[[j,i]] %>% is.na() == FALSE){
        if (hitting[[j,i]] == "T"|hitting[[j,i]] == "L"){
          hitting[[j,i]] <- "F"
        } else if (hitting[[j,i]]=="S"){
          hitting[[j,i]] <- "C"
        } else if (hitting[[j,i]]=="I"){
          hitting[[j,i]] <- "B"
        }
      }
    }
  }
  return(hitting)
}

bondhit <- same_pitch_types(bondhit)
```

Hacky, but it works. I also renamed the first column of 1s EBV for ease of comprehension.

Part 5: Calculate the EBV

I think I'm going to do this in two parts. First, I'm going to split off the fouls.

I'm using the test line CBCFFX. I've written a function:

```
foul_splitter <- function(hitline){
  hitline_dupe <- hitline
  for (i in 2:length(hitline)){
    hitline_dupe <- hitline
    if (is.na(hitline[, i]) == FALSE && hitline[,i] == "F"){
      hitline[,i] <- "C";
      hitline$EBV <- hitline$EBV*0.809;
      hitline_dupe[,i] <- "B";
      hitline_dupe$EBV <- hitline_dupe$EBV*0.191
      hitline <- rbind(hitline, hitline_dupe)
    }
  }
  return(hitline)
}
```

When it takes the test line as input, it appears to do exactly what we want it to do:

```
> foul_splitter(test_line)
# A tibble: 4 x 8
     EBV `1`   `2`   `3`   `4`   `5`   `6`   `7`
   <dbl> <chr> <chr> <chr> <chr> <chr> <chr> <chr>
1 0.654  C     B     C     C     C     X     NA
2 0.155  C     B     C     B     C     X     NA
3 0.155  C     B     C     C     B     X     NA
4 0.0365 C     B     C     B     B     X     NA
```

Beautiful! That does exactly what I want it to do.

Next, we need to account for the hits. We can essentially use the same function, but with 0.413 for C's and 0.587 for B's for every pitch after the hit. The major change here will be replacing the slots after the hit with split predicted pitches.

I need to ask myself the question at this point: should I be splitting every pitch? Even with fouls, I really only want to split until I know whether or not Barry gets on base in any given split.

I spent about five minutes editing the function to only split in some cases, and then I thought to myself: No I don't! I'm going to take this column by column. If I overgenerate data it doesn't matter. I'm going to write a function that splits every hit no matter what.

```r
hit_splitter <- function(hitline){
  hitline_dupe <- hitline
  for (i in 2:length(hitline)){
    hitline_dupe <- hitline
    if (is.na(hitline[, i]) == FALSE && hitline[,i] == "X"){
      hitline[,i] <- "C";
      hitline$EBV <- hitline$EBV*0.809;
      hitline_dupe[,i] <- "B";
      hitline_dupe$EBV <- hitline_dupe$EBV*0.191
      for (j in i:length(hitline)){
        hitline_trip <- hitline
        hitline[,j] <- "C";
        hitline$EBV <- hitline$EBV*0.413;
        hitline_trip[,j] <- "B";
        hitline_trip$EBV <- hitline_trip$EBV*0.587
      }
      hitline <- rbind(hitline, hitline_dupe, hitline_trip)
    }
  }
  return(hitline)
}
```

```
> test_line %>% foul_splitter() %>% hit_splitter()
# A tibble: 16 x 10
        EBV `1`   `2`   `3`   `4`   `5`   `6`   `7`   strike_count ball_count
      <dbl> <chr> <chr> <chr> <chr> <chr> <chr> <chr> <chr>        <chr>
 1 0.0125   C     B     C     C     C     C     C     C            C
 2 0.00294  B     B     C     C     C     C     C     C            C
 3 0.00294  C     B     C     B     C     C     C     C            C
 4 0.000695 B     B     C     B     C     C     C     C            C
 5 0.00294  C     B     C     C     B     C     C     C            C
 6 0.000695 B     B     C     C     B     C     C     C            C
 7 0.000695 C     B     C     B     B     C     C     C            C
 8 0.000164 B     B     C     B     B     C     C     C            C
 9 0.0120   C     B     C     C     C     B     B     B            B
10 0.00283  B     B     C     C     C     B     B     B            B
11 0.00283  C     B     C     B     C     B     B     B            B
12 0.000669 B     B     C     B     C     B     B     B            B
13 0.00283  C     B     C     C     B     B     B     B            B
14 0.000669 B     B     C     C     B     B     B     B            B
15 0.000669 C     B     C     B     B     B     B     B            B
16 0.000158 B     B     C     B     B     B     B     B            B
```

That did not do what I wanted it to do.

I think my move here is to make it even simpler. I'm just going to add X to the splitter function, since we're calculating the correct values for a hit. Then we can split all NA values into hits.

```
16 hit_maker <- function(hitline){
17   for (i in 4:length(hitline)){
18     hitline_dupe <- hitline;
19     if (is.na(hitline[,i]) == TRUE){
20       hitline[,i] <- "C";
21       hitline$EBV <- hitline_dupe$EBV*0.413;
22       hitline_dupe[,i] <- "B";
23       hitline_dupe$EBV <- hitline_dupe$EBV*0.587;
24       hitline <- rbind(hitline, hitline_dupe);
25     }
26   }
27   return(hitline)
28 }
```

That seems to work.

It was at this point that I turned away from the document and began going in on the problem of applying functions to an entire dataframe. I spent literally a day and a half on this. I tried nesting for loops (which I know you're not supposed to do in R, but I am bad at coding), I tried just throwing a dataframe in there to see what would happen, I tried prayer. The answer was before my eyes: using a dummy variable! With this, I have brought a monster into the world.

Behold: row_applier()!

```
row_applier <- function(df,func){
  big_dummy <- tibble(
    EBV = numeric(),
    strike_count = numeric(),
    ball_count = numeric(),
    `1` = character(),
    `2` = character(),
    `3` = character(),
    `4` = character(),
    `5` = character(),
    `6` = character(),
    `7` = character(),
  )
  for (i in 1:nrow(df)){
    dummy <- func(df[i,])
    big_dummy <-  rbind(dummy, big_dummy)
  }
  return(big_dummy)
}
```

I like to think of code as little machines that I create and maintain, and like any moderately experienced mechanic, I get a good feel for the quality of the machine. This is a piece of shit. This function is absolute fucking garbage. Works though!

I also added strike count and ball count columns in the greater sheet's 2 and 3 column positions respectively and added code to calculate whether or not a given hitting line is a strike or a walk:

```
ball_strike_counter <- function(hitline){
  for (j in 1:nrow(hitline)){
      for (i in 4:length(hitline)){
      if (hitline[j,3] == 4){
        break;
      };
      if (hitline[j,2] == 3){
        hitline[j,1] <- 0
        break;
      };
      if (hitline[j,i] == "C"){
        hitline[j,2] <- hitline[j,2]+1
      }
      if (hitline[j,i] == "B"){
        hitline[j,3] <-  hitline[j,3]+1
      }
    }
  }
  return(hitline)
}
```

Straightforward enough, and this one does work on the entire matrix.

Fuck, I forgot to deal with HBPs. Uhhhh shit. Let me see if I can work around this.

```
hbp_splitter <- function(hitline){
  for (i in 4:length(hitline)){
    if (hitline[,i] == "H" && is.na(hitline[,i])==FALSE){
      hitline <- tibble(
        EBV = c(1),
        strike_count = c(0),
        ball_count = c(0),
        `1` = c("H"),
        `2` = c("H"),
        `3` = c("H"),
        `4` = c("H"),
        `5` = c("H"),
        `6` = c("H"),
        `7` = c("H"))
    }
  }
  return(hitline)
}
```

Hacky garbage again but it does the job. I'm replacing HBPs with Hs across the board to stop the function from running on it again.

Let's put it all together! Drumroll please...

```
> bond_results <- row_applier(bondhit,hbp_splitter)
> bond_results <- row_applier(bond_results,swung_splitter)
> bond_results <- row_applier(bond_results,hit_maker)
There were 50 or more warnings (use warnings() to see the first 50)
> bond_results <- bond_results %>% ball_strike_counter()
> View(bond_results)
> bond_results$EBV %>% sum()
[1] 366.8707
> 366.8707 / 613
[1] 0.598484
```

Without a bat, Barry gets a single-season OBP of 0.5985 in 2004. Still the best in baseball history, but a smidge behind his real-life numbers.

FAQ:

- The final product took like five minutes to run and now my laptop smells a little like burnt plastic

- Yes, I did run sum(sheet$EBV) after each row_applier() to make sure that the EBVs were still adding up to 613 before it went into the ball/strike counter. They did, don't worry
- You could generalize this out into being generic for any player with some tweaks, but I'm not going to. Maybe someday.
- Yes, I do have a girlfriend. Her review of this experiment was that it is "totally sick and has totally sick results" but that if I "explained it to her step by step she would ask me to stop".