FPTP to AV
A Tool for Performing Simulations of Alternative Vote using the actual FPTP votes *

Daniel MARCELINO†

May 9, 2014

Date Performed: May 8, 2014

1 Introduction

Simulations provide a means for calculating uncertainties in situations where real-life changes are simply too risky or costly to try. In the next paragraphs, I describe a software writing in R for performing simulations of election outcomes under the rules of the Alternative Vote electoral system using the actual FPTP votes as input (AV $\sim$ FPTP). How does it work? The *fptp2av()* algorithm takes two inputs: (1) the actual FPTP vote in a rectangular matrix form, and (2) a vector of transitive political preferences as $A < B < C$, which will be used to inform the random process of followers attribution. The following sections delve into the mechanics of the software. Installation and replication code is provided in the appendix.

2 The mechanics

For each district, the function 'District' generates a list of follower groups for each candidate/party in the constituency. The number of such groups for every candidate equals to the votes recorded for him or her. Then, a pseudo-function, '*count_votes_AV'*, proceeds as follow:
(1) It counts the total of votes from all candidates in a constituency. This is done by counting the number of groups of followers generated by 'District'.
(2) It counts the votes for each candidate by connoting the number of his or her groups of followers.
(3) If there is one candidate with more than half of the total votes, he or she is the winner; then the function '*count_votes_AV'* returns the winner and the number of winning

*This program is part of the package SciencesPo; Thus, it is free of charge and multi-platforms. It first appeared in the 2013 APSA meeting.

†Dept. of Political Science, Univ. of Montreal, Canada, E-mail: dm.silva@umontreal.ca.

votes and the program stops or move to the next unit. Note that if there is a winner at this stage, the "AV Winner" is the same as the 'FPTP Winner'.
(4) Otherwise, the weakest candidate (with the lowest votes) will be eliminated from the list. His or her follower groups will be redistributed to other candidates. That is, the second vote preference will be distributed to the first party/candidate that appears. To put it in terms of a concrete example: say in '$vote\_spread'$ there are parties $\{A,B,C\}$, each with a number of follower groups. **C** is the weakest candidate with the smallest number; therefore, it is eliminated right after the first round. One of **C**'s followers has the following preference order $D > B > E > A$. This vector of preferences will be redistributed to party **B**, since **D** does not appear to have any candidate in this district according to the information stored in '$vote\_spread'$ matrix. In this example, the votes for **D** and **E** would be wasted as both parties are not competing in that constituency.
(5) In this list of follower groups, each group will be redistributed to the remaining candidate which name first appears in the list.
(6) Each remaining party/candidate will have no less follower groups than before, but some of them might not inherit any vote. Their votes are recounted and steps 4 and 5 are repeated until one candidate has more than half of the total votes.
(7) After going through these steps, the function returns the winner and final count of votes, which are printed in the screen (as figure in the appendix).

3 The role of randomness

First, the follower groups generated for each party/candidate are distributed by chance based on how many votes he or she received. That is, if a candidate has 4000 votes in *vote_spread*; then 4000 groups of followers will be randomly attributed. However, these groups will be randomly generated from the list of party relations or preference chains. This is the trickiest part of the algorithm, but also the potential one. If we know a lot about the electorate behaviour, we can build very likely chains. Otherwise, we can simply try different scenarios.
Second, the number of followers in each group is also randomly determined up to the total number turnout voters in the constituency, which is informed by *max_vote_length* object.

4 The output format

The simulations outcome is printed on the R prompt for each constituency as the software proceeds. By the end, a final file (CSV) is produced and stored in the working directory for further analysis.

A Syntax for replication:

The following is a plain example, but I usually run it inside of a bootstrap loop so to get robust results and spawn the process across nodes to perform it quickly.

```
# In R type or paste:
install.packages("SciencesPo")
require(SciencesPo)
#Let's use the 2010 UK election
data(ge2010)
# Now depending on you machine it will take a while to go through all the 605 constituencies, so you may want to run only a portion as that:
# Let's take a small portion of the data.
IR <-ge2010[ge2010$Region=="Northern Ireland",];

APNI<- c("SF", "DUP", "SDLP", "TUV", "UCUNF");
BNP<-c("Ch P", "Con", "CPA", "NF", "UKIP");
Ch_P<-c("BNP", "CPA", "Con", "ED") #Christian Party;
Con<-c("LD", "UKIP", "ED", "Ch P", "CPA") #Conservative Party;
CPA<-c("BNP", "Ch P", "Con", "ED");
DUP<-c("SDLP", "TUV", "UCUNF", "APNI");
ED<-c("Con", "BNP", "NF", "CPA", "Ch P") #English Democrats;
Grn<-c("LD", "Lab", "SDLP", "PC", "SSP", "TUSC", "TUV");
Lab<-c("LD", "Grn", "SDLP", "Soc", "SSP", "TUSC", "TUV");
LD<-c("Lab", "Con", "Grn", "PC", "SDLP") #Lib Dem;
MRLP<-c("Con", "Lab", "LD", "Grn", "PC", "SDLP", "SNP", "SSP", "UKIP")
NF<-c("BNP", "Con", "ED", "UKIP") #National Front;
PC<-c("Con", "Lab", "LD");
Respect<-c("Con", "Lab", "LD");
SDLP<-c("Grn", "Lab", "LD") #Social Democratic & Labour Party;
SF<-c("APNI", "USUNF");
Soc<-c("Lab", "SSP", "SDLP", "TUSC", "TUV") #Lab Socialist Labour Party;
SNP<-c("Ch P", "CPA", "UKIP") #Scottish National Party;
SSP<-c("Grn", "Lab", "LD", "SDLP", "TUSC", "TUV");
TUSC<-c("SDLP", "Soc", "SSP", "TUV");
TUV<-c("SDLP", "Soc", "SSP", "TUSC");
UCUNF<-c("APNI", "SF", "DUP");
UKIP<-c("BNP", "NF", "Con");

# We feed the program with a transitory party preference list, which I call party_chains:
party_chains <-structure(list(APNI,BNP,Ch_P,Con,CPA,DUP,ED,Grn, Lab,LD,MRLP,NF,PC, Respect,SDLP,SF,Soc,SNP,SSP,TUSC,TUV,UCUNF,
  UKIP), .Names = c("APNI","BNP","Ch_P","Con","CPA","DUP","ED","Grn","Lab",
"LD","MRLP","NF","PC","Respect","SDLP","SF","Soc","SNP","SSP","TUSC",
"TUV","UCUNF","UKIP" ));

# Finally, run it:
 fptp2av(data=ge2010, link=party_chains)
```

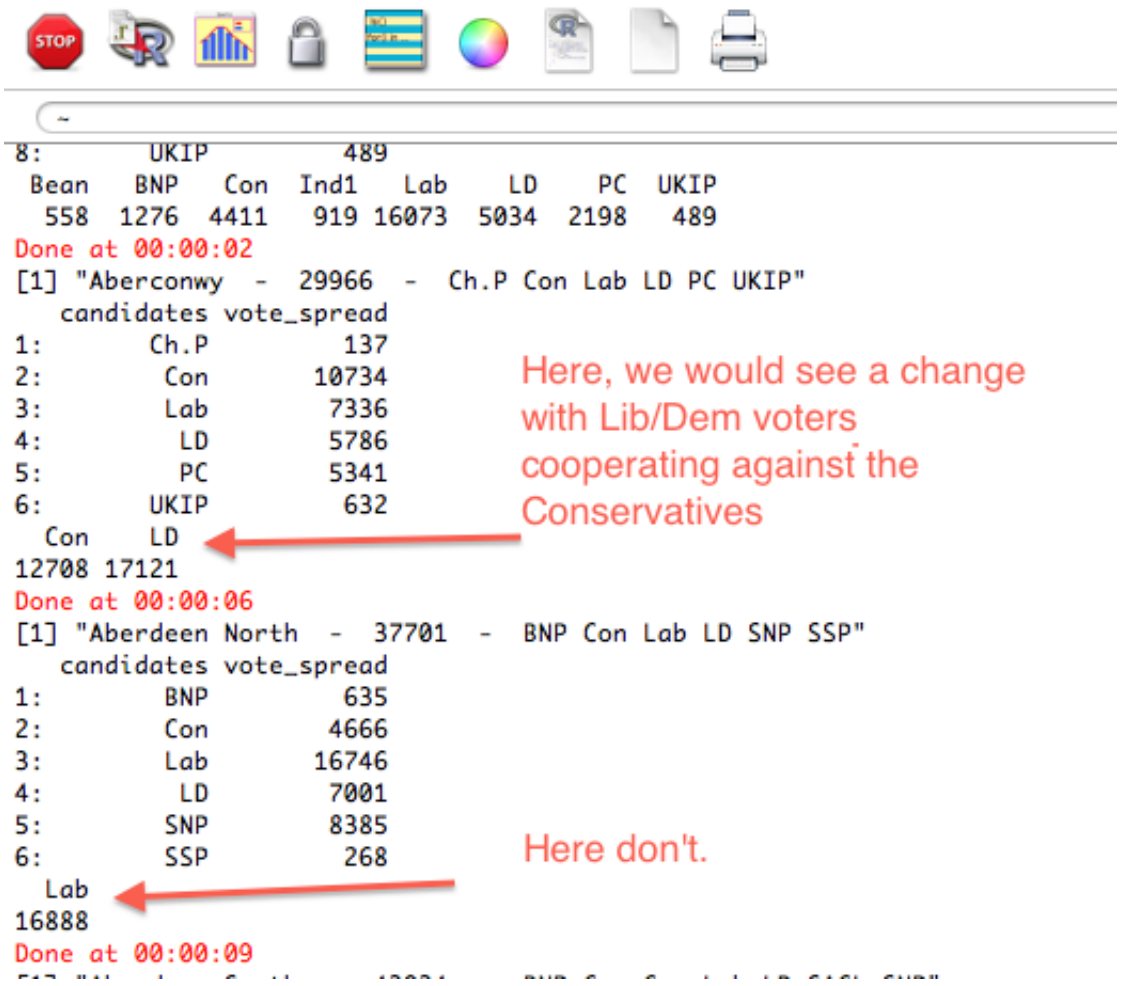

Figure 1: *Running simulations*