Demography 213
IPUMS and Graphs in R

Carl Mason
carlm@demog.berkeley.edu

October 3, 2016

Contents

Abstract

This week you will extend our results from last week's IPUMS investigation by creating informative graphs of transcendent beauty.

1 Introduction

Last week you identified an interesting rate or probability and wrestled it free from the clutches of an IPUMS data extract. This week we will work on presenting that result in a way that makes it seem even more important than it is. You have already made many graphs in this class and have been exposed to many of R's obscure graphics options. So this week should be

pretty easy. But also quite useful as graphs are what makes your results worth thinking about.

The assignment for this week is to quickly work through the examples in this file and in the `demonstraion.r` file (which contains the code from this handoout), and then to use the ideas and skills thereby acquired to refine or extend your analysis from last week and express your (perhaps) new and (perhaps) improved result in a graph that excites and delights the scientific mind. As usual send email to cmason@berkeley.edu with thoughtful reflections or pleas for help.

Here are some issues to consider when graphing rates:

- The size of the different risk sets (population at different ages/times)
- The natural ordering of the groups by which the rates are disaggregated
- The use of color
- The use of smoothers
- Plotting differences or ratios
- Showing time
- Showing confidence bounds

Some of the above issues are illustrated in the graphs below. Your assignment for this week is display the rates that you just produced in at least two graphs both of which must be **better** than any of the examples below.

Don't forget to Google. There are lots of graphing packages (most of which I don't know about) that will allow fancier and sometimes even better versions of common types of plots. Search for "R plot examples" for things that will surprise us.

```
library(foreign)
## read the ACS2009 data that we used in the TFR example
## cleanse key variables of pernicious peculiarities.

acs09<-read.dta(file="/data/carlm/ACS2009/acs09.dta")
names(acs09)
```

```
##  [1] "year"      "datanum"   "serial"    "hhwt"      "region"
##  [6] "statefip"  "gq"        "pernum"    "perwt"     "famsize"
## [11] "nchild"    "nchlt5"    "age"       "sex"       "marst"
## [16] "marrno"    "marrinyr"  "widinyr"   "divinyr"   "fertyr"
## [21] "race"      "raced"     "bpl"       "bpld"      "citizen"
## [26] "yrnatur"   "yrimmig"   "hispan"    "hispand"   "hcovany"
## [31] "hcovpriv"  "hinsemp"   "hinspur"   "occ"       "migrate1"
## [36] "migrate1d" "diffmob"

##GOTCHA 1
acs09$ageN<-as.numeric(as.character(acs09$age))
table(acs09$ageN)

##
##    15    16    17    18    19    20    21    22    23    24    25    26
## 20231 20318 20815 20648 19442 17924 17160 16530 17209 17110 17088 17239
##    27    28    29    30    31    32    33    34    35    36    37    38
## 17532 17824 18005 18402 17777 17792 17265 17429 17828 17983 18853 20534
##    39    40    41    42    43    44    45    46    47    48    49    50
## 20579 20932 19360 20286 20679 22050 23425 22969 23360 23886 23918 24931

## GOTCHA 2
acs09$fertyrL<-ifelse(acs09$fertyr=="Yes",TRUE,FALSE)
table(acs09$fertyrL)

##
##  FALSE   TRUE
## 665705  39608

## GOTCHA 3
acs09$citizenF<-factor(as.character(acs09$citizen))
levels(acs09$citizenF)

## [1] "Born abroad of American parents" "N/A"
## [3] "Naturalized citizen"             "Not a citizen"

levels(acs09$citizenF)[2]<-"US Born"
table(acs09$citizenF)

##
```

```
## Born abroad of American parents                        US Born
##                             7146                         596433
##             Naturalized citizen                 Not a citizen
##                           41749                         59985


## And the magic that we learned last week.
## since we're going to do this a few times Let's
## create a file called doASFR.R that holds this code
# and then we'll source() it when we need it.

# NOTE that we *could* instead write a function, BUT since
# we want to use several intermediate calculations as well as
# the main result, it will be easier in this case to store the code
# and re run it via source()


## code to a file called : doASFR.r and source() it

## create a character vector that holds R code
## Note the single ' marks that enclose the whole thing

if(FALSE){
##  DROP THIS CODE INTO A NEW FILE CALLED doASFR.R

  num.den<-with(acs09,
                tapply(perwt,
                       list(ageN,citizenF,fertyrL),
                       sum,na.rm=TRUE)
              )

  numer<-num.den[,,"TRUE"]
  denom<-apply(num.den,MARGIN=1:2,FUN=sum,na.rm=TRUE)
  ASFR<-numer/denom

## END CODE SNIPPET
}
## run the code in the file called doASFR.R
source("doASFR.R")
apply(ASFR,MARGIN=2,FUN=sum,na.rm=TRUE)
```

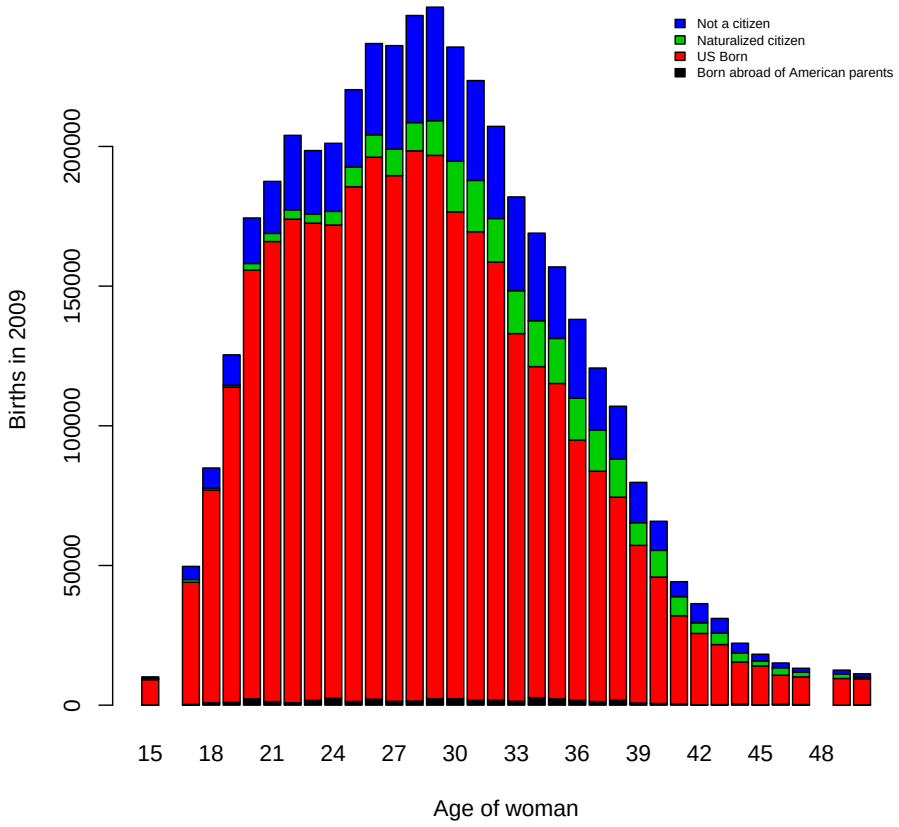
```
## Born abroad of American parents                        US Born
##                         1.989181                       2.004676
##             Naturalized citizen                  Not a citizen
##                         2.142767                       2.865514
```

2 Improving barplots

2.1 Ordering categories

Last week, we created a bar graph to display the values in the numerator (counts of women who had babies in the previous year by age and citizenship status).

```
barplot(t(numer),xlab="Age of woman",ylab="Births in 2009",
        col=(leg.coll<-1:length(levels(acs09$citizenF))))
## NOTE that in the legend command we can use rev to reverse the order
## in which the legend items appear to match the order of the bars.
legend(x="topright",fill=rev(leg.coll),legend=(rev(levels(acs09$citizenF))),
       cex=.6,bty='n')
```

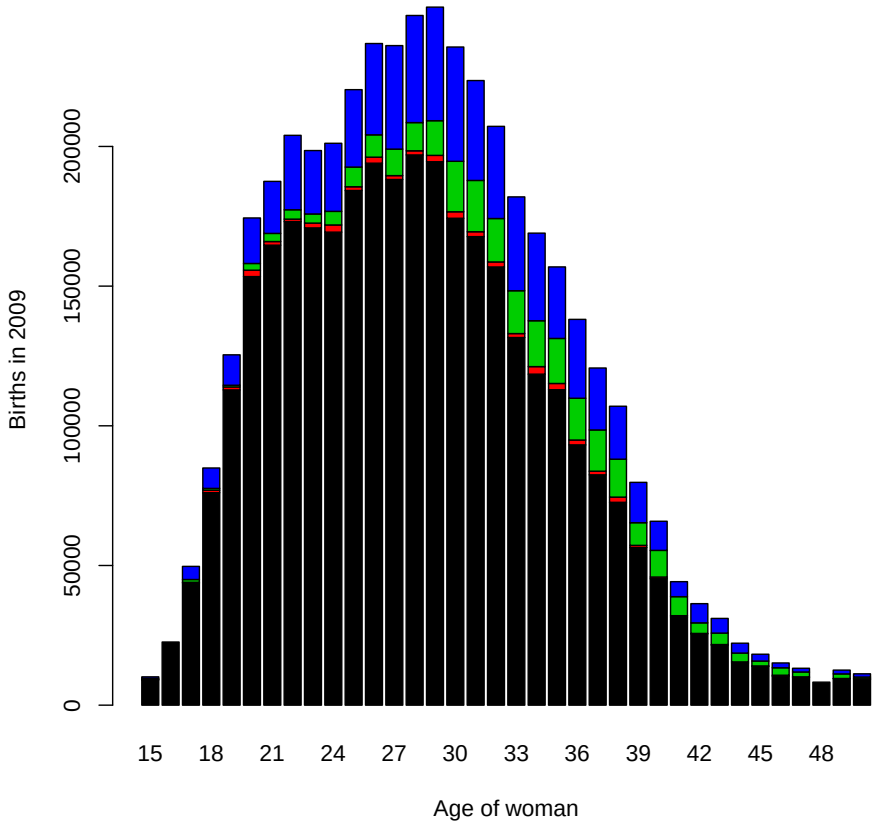


While this is a nice graph, notice the very small black bars (representing "born abroad of American parents") at the bottom of each stack. Wouldn't it be nice if we could reorder the bars according to some scheme other than the arbitrarily integers that R assigned the factor when it was created? For example, one might want to order the stacked bars with the largest bar on the bottom and the smallest on the top, or one might want to sort from biggest to smallest within sub categories such as immigrant and non-immigrant.

The easiest way to accomplish this is with `relevel()`. This function simply fiddles with the integer values associated with the level of a factor.

```
## Born Abroad is the first category, but it is very small
table(acs09$citizenF)

##
```

```
## Born abroad of American parents                        US Born
##                             7146                         596433
##             Naturalized citizen                  Not a citizen
##                           41749                         59985

## Let's put US Born first: the ref argument indicates which level
## should become the first
## and all other move down as  necessary

acs09$citizenF<-relevel(acs09$citizenF,ref="US Born")
table(acs09$citizenF)

##
##                         US Born Born abroad of American parents
##                          596433                           7146
##             Naturalized citizen                  Not a citizen
##                           41749                         59985

## to get them exactly as I want them requires at most N calls to relevel
## remember with barplot() the first category will be on the bottom so...
acs09$citizenF<-relevel(acs09$citizenF,ref="Not a citizen")
acs09$citizenF<-relevel(acs09$citizenF,ref="Naturalized citizen")
acs09$citizenF<-relevel(acs09$citizenF,ref="Born abroad of American parents")
acs09$citizenF<-relevel(acs09$citizenF,ref="US Born")
## now recalculate ASFR with the new levels
source(file="doASFR.R")

## And the same barplot() code from above but

barplot(t(numer),xlab="Age of woman",ylab="Births in 2009",
        col=(leg.coll<-1:length(levels(acs09$citizenF))))
```




```
legend(x="topright",fill=rev(leg.col),legend=rev(levels(acs09$citizenF)),
       cex=.6,bty='n')

## Error in rev(leg.col):  object 'leg.col' not found
```

2.2 Colors

Color is valuable tool to display information, but it is also a subtle one. Ideally we would like to be able to use color to indicate magnitude as well as category, but we are limited in this respect by our culture's inability to agree whether red is greater than blue or by precisely how much "paleviolet" exceeds "peachpuff".

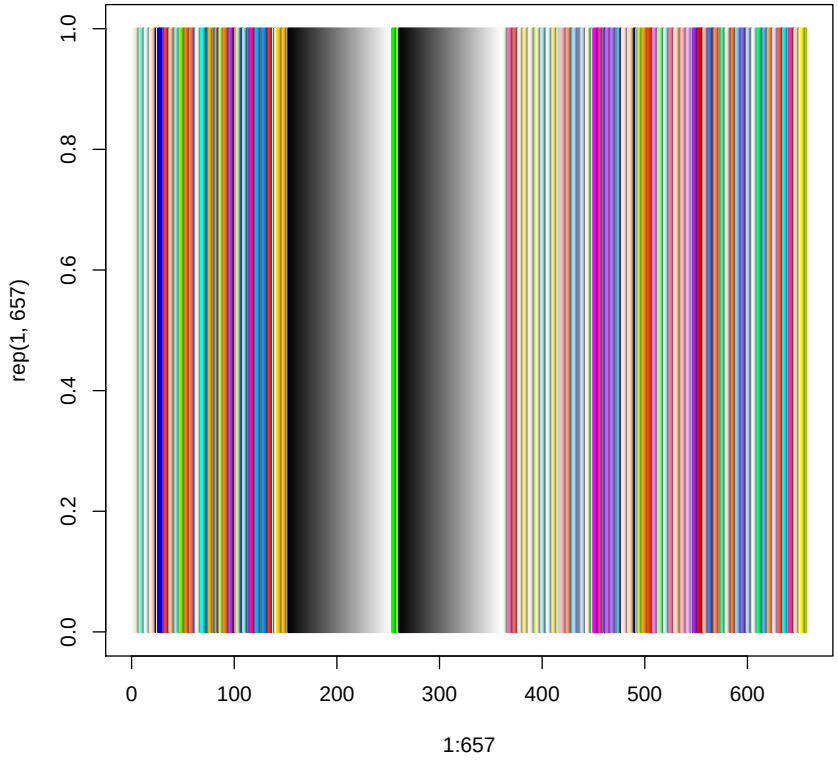
There are also limitations one the number of colors that can be distinguished by the human eye – especially when displayed on electronic devices or printers whose specifications we do not know.

Luckily, color is fun to play with and easy to use. Most graphic functions take the argument: `col` to indicate color. Generally, one must set col equal to a vector of character strings with the same length as the number of variables being plotted. Of course, col will recycle if the length of its argument is too short. Some functions (for example lattice function) take more than one color specifier, in which one must check the manual for what to call whatever it is that behaves like the more standard "col" argument.

The "col" argument to R's graphic functions understand color in this familiar format: "#68FA00BC". As is surely obvious this is simply four 2-digit hexadecimal numbers, each of which determines the intensity of the red, green or blue component plus a fourth number that you can learn all about by go ogling "ICE standard 61966". Happily, R provides several civilized ways for specifying color without resorting to hexadecimal barbarism:

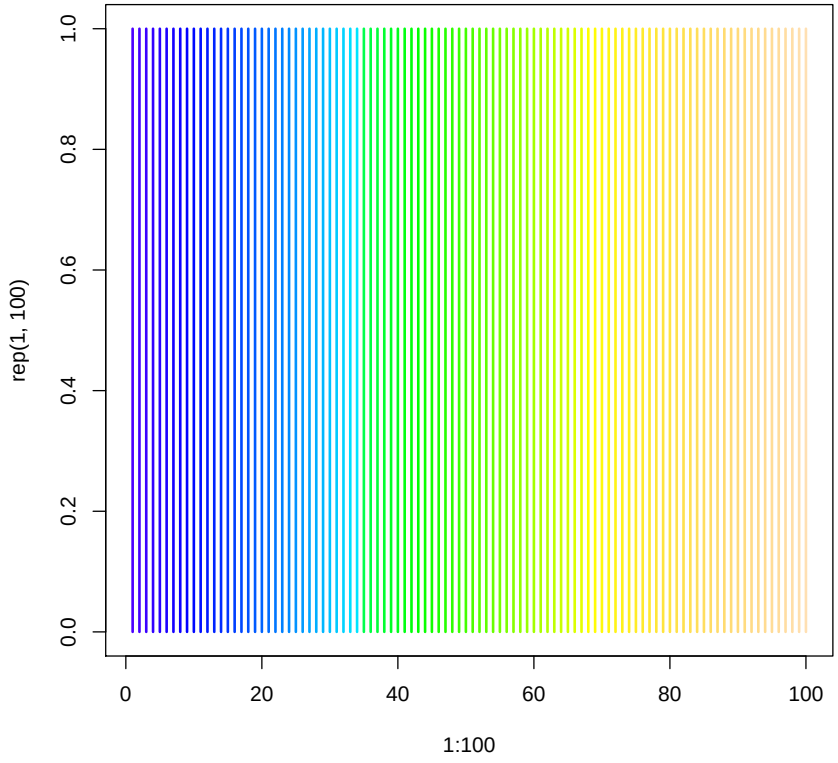
- By number: the `col` option accepts integer numbers which correspond to the colors used in the previous graph. These are high contrast Fisher-Price sort of colors.
- By name: the `colors()` function will display 657 different named colors which the `col` option will recognize if they are in quotes. Names unfortunately do not carry a whole lot of information about what a color actually looks like. Note that expressions like `colors()[c(45,12,3)]` can help you avoid typing color names.

```
## 657 named colors
plot(y=rep(1,657),x=1:657,type='h',col=colors()[1:657],ylim=c(0,1),lwd=2)
```



- `rainbow(),topo.colors(),heat.colors()`: These are functions that produce "palettes" of related colors. They are easy to use in that they will produce colors that are distinct from one another and in the latter two cases, colors that attempt to indicate relative magnitude.

```
## topo.colors try to indicate relative magnitudes
plot(y=rep(1,100),x=1:100,type='h',col=topo.colors(100),ylim=c(0,1),lwd=2)
```



- `colorRampPalette`: This is the most flexible. It can generate a palette that runs from say "blue" to "red" in 13 "even" steps. This function is particularly useful when you have two or more broad categories, such as immigrant vs non-immigrant, and sub-categories within such as naturalized and not naturalized.

```
## colorRampPalette is ODD in that it is a function that returns a function
colorRampPalette(colors=c("violet","green"))

## function (n)
## {
##     x <- ramp(seq.int(0, 1, length.out = n))
##     if (ncol(x) == 4L)
##         rgb(x[, 1L], x[, 2L], x[, 3L], x[, 4L], maxColorValue = 255)
```

```
##     else rgb(x[, 1L], x[, 2L], x[, 3L], maxColorValue = 255)
## }
## <bytecode: 0x8fe6460>
## <environment: 0x8fe55f8>

## but for the most part it is a function that you onely want to call once
## generally I use it to create a vector of a known number of colors
## which means calling it like this:

pal.citz<-colorRampPalette(colors=c("magenta","green"))(4)

pal.citz

## [1] "#FF00FF" "#AA55AA" "#55AA55" "#00FF00"
```

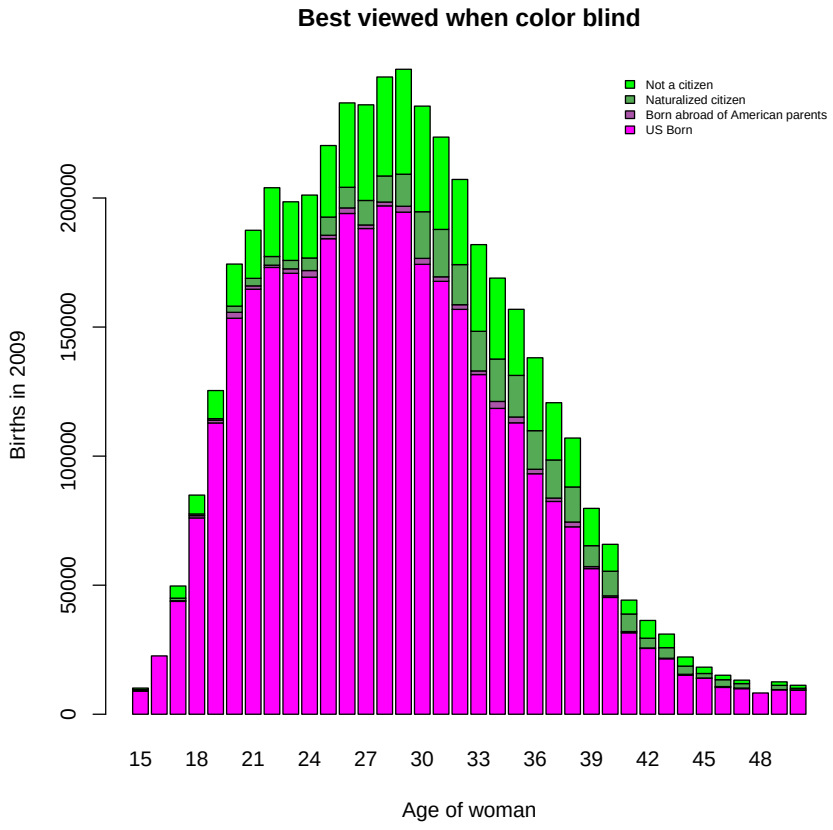
A very valuable trick when working with vectors of colors such as those produced by `color(), heat.color(), rainbow()` or `colorRampPalette()` is to **name the colors according to what they will represent**. For example each element of `pal.citz` represents a particular citizenship status.

```
## It is very useful to assign names to color vectors based on what they will be
## in this case the colors in pal.citz will refer to the levels of acs09$citizen
## which are also the colnames(numer)
names(pal.citz)<-levels(acs09$citizenF)
pal.citz

##                          US Born Born abroad of American parents
##                        "#FF00FF"                        "#AA55AA"
##             Naturalized citizen                   Not a citizen
##                        "#55AA55"                        "#00FF00"
```

This trick allows us to use colors in clearer and less error prone ways

```
## And the same barplot() code from above
barplot(t(numer),xlab="Age of woman",ylab="Births in 2009",
        col=(leg.col<-pal.citz[colnames(numer)]),
```

```
        main="Best viewed when color blind")
legend(x="topright",fill=rev(leg.col),legend=rev(names(leg.col)),
       cex=.6,bty='n')
```




```
## note that rev()  reverses the order of a vector s
## Perhaps not entirely successful esthetically. Oh well.
```

2.3 The width of bars

The widths of the bars can carry information but in our present example, it does not. Would it be cool to make the widths of the bars convey the size of the risk set?

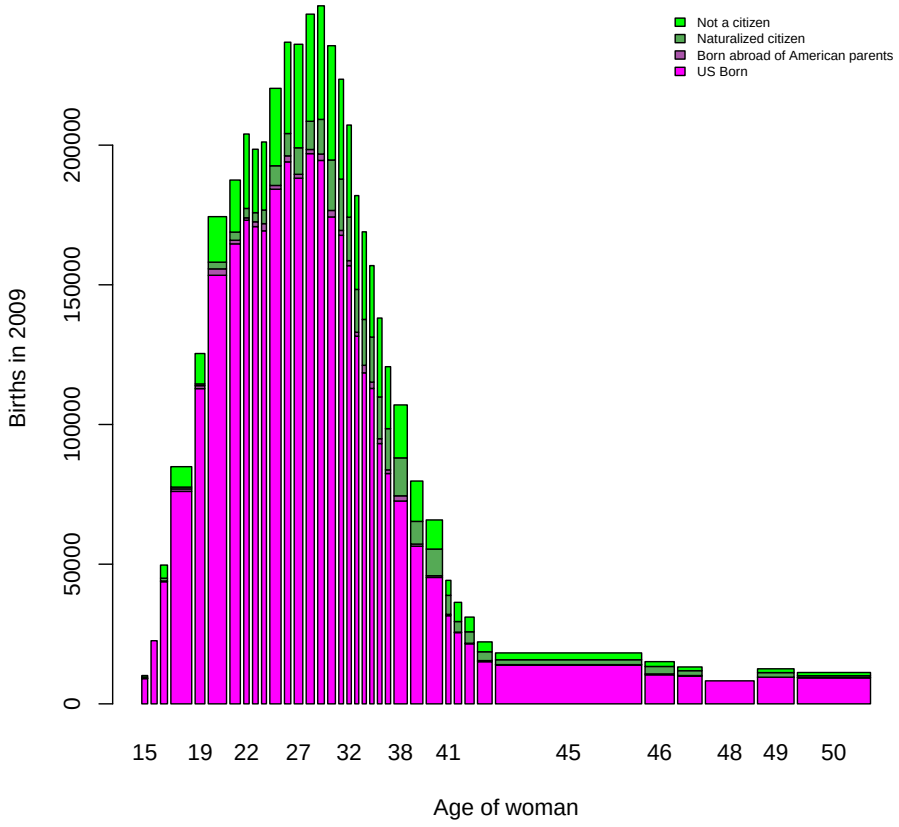
```r
## The width argument determines the width of the bars (brilliant huh?)
## it needs to be a vector the same length as the number of bars.

popByage<-apply(denom,MARGIN=1,FUN=sum,na.rm=TRUE)

popByage
```

```
##      15      16      17      18      19      20      21      22      23
## 2021073 2051224 2072088 2259214 2131152 2248402 2135473 2014928 2017685
##      24      25      26      27      28      29      30      31      32
## 1996535 2142672 2063856 2105031 2103014 2071667 2090188 1968784 1943163
##      33      34      35      36      37      38      39      40      41
## 1886745 1893180 1940198 1944194 2003563 2163633 2156806 2244541 2000525
##      42      43      44      45      46      47      48      49      50
## 2076670 2127071 2216745 2361680 2268307 2260893 2322750 2321711 2349311
```

```r
## Unfortunately there turns out to be distressingly little variation
## in cohort size among women in the childbearing years in 2009
## That there are more 50 year olds than 17 yearolds is cause of both
## of concern AND of employment for demographers.

## for demonstration purposes we'll need
## to make the effect more pronounced (albeit less truthful)
NotpopByage<- 1/rank(-1*popByage) ## rank() assigns 1 to the smallest number
## -1 reverses the ranking
NotpopByage
```

```
##         15         16         17         18         19         20
## 0.04000000 0.04166667 0.04761905 0.14285714 0.06666667 0.12500000
##         21         22         23         24         25         26
## 0.07142857 0.03703704 0.03846154 0.03333333 0.07692308 0.04347826
##         27         28         29         30         31         32
## 0.05882353 0.05555556 0.04545455 0.05263158 0.03225806 0.03030303
##         33         34         35         36         37         38
## 0.02777778 0.02857143 0.02941176 0.03125000 0.03571429 0.09090909
##         39         40         41         42         43         44
## 0.08333333 0.11111111 0.03448276 0.05000000 0.06250000 0.10000000
##         45         46         47         48         49         50
## 1.00000000 0.20000000 0.16666667 0.33333333 0.25000000 0.50000000
```

```r
## so this really distorts the differences in cohort size quite a lot.
barplot(t(numer),xlab="Age of woman",ylab="Births in 2009",
        col=(leg.coll<-pal.citz[colnames(numer)]),
        width=NotpopByage)
legend(x="topright",fill=rev(leg.coll),legend=rev(names(leg.coll)),
       cex=.6,bty='n')
```

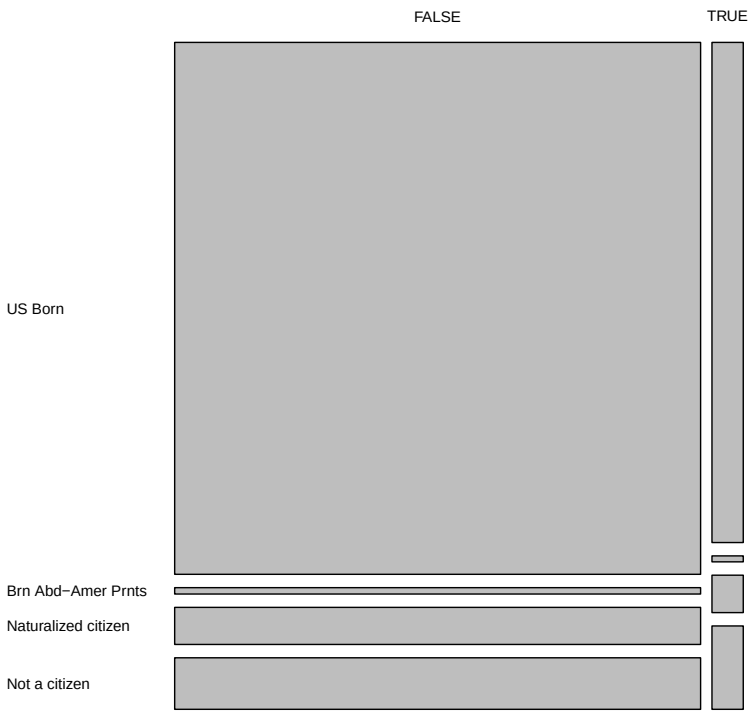


3 Extreme barplots or mosaicplots

A bar plot carried to its logical conclusion is a "mosaicplot". The simplest mosaic plots are constructed from frequency tables. The example below

shows a very simple mosaic plot. We'll get fancier shortly. Note that the this sort of plot works best with the output of table function.

```
## Unfortnuately the barplot does not allow us to see the variation in
## size of the citizenship category within an age group.
## mosaic plots can in theory do this

## A mosaic plot is a great way to show a contingency table where one dimension
## represents a subdivision of the at-risk set (citizenship)
## and another represents an outcome (fertility)

## for example (after shortening one of the level names)
levels(acs09$citizenF)[2]<-"Brn Abd-Amer Prnts"
## here is a simple plot showing the fert outcome
## for the population disagregated by citizenship
print(mosaicplot(table(acs09$fertyrL,acs09$citizenF),
           las=1))
```

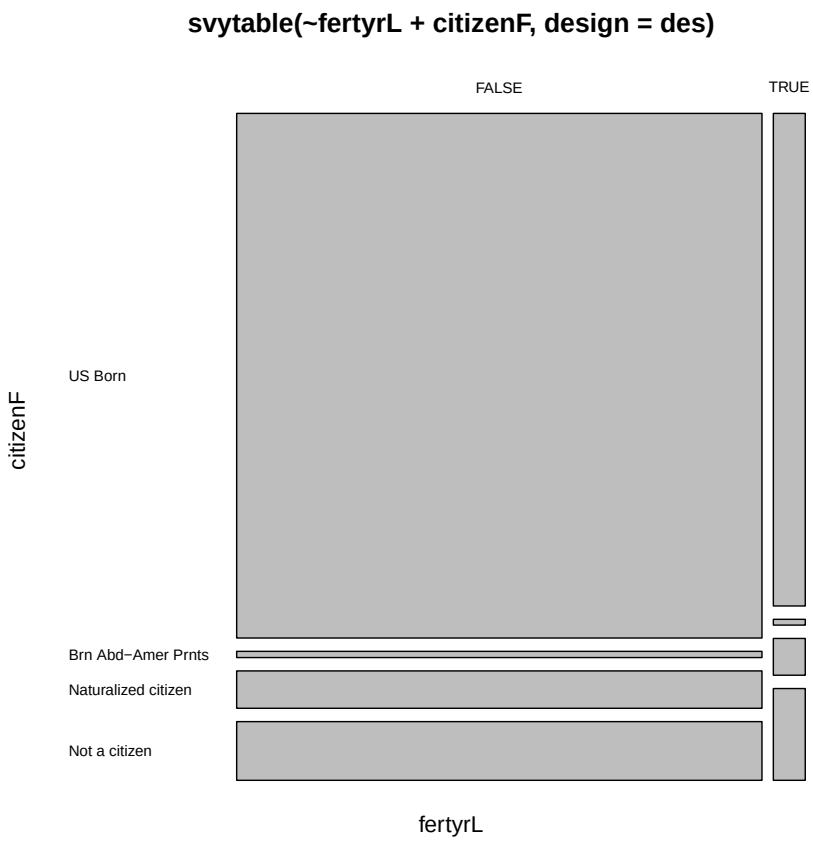



```
## NULL

## the las argument to (mosaic)plot causes the labels to be printed
## horizontally.

## Unfortunately this plot is not weighted so it is quite wrong
```

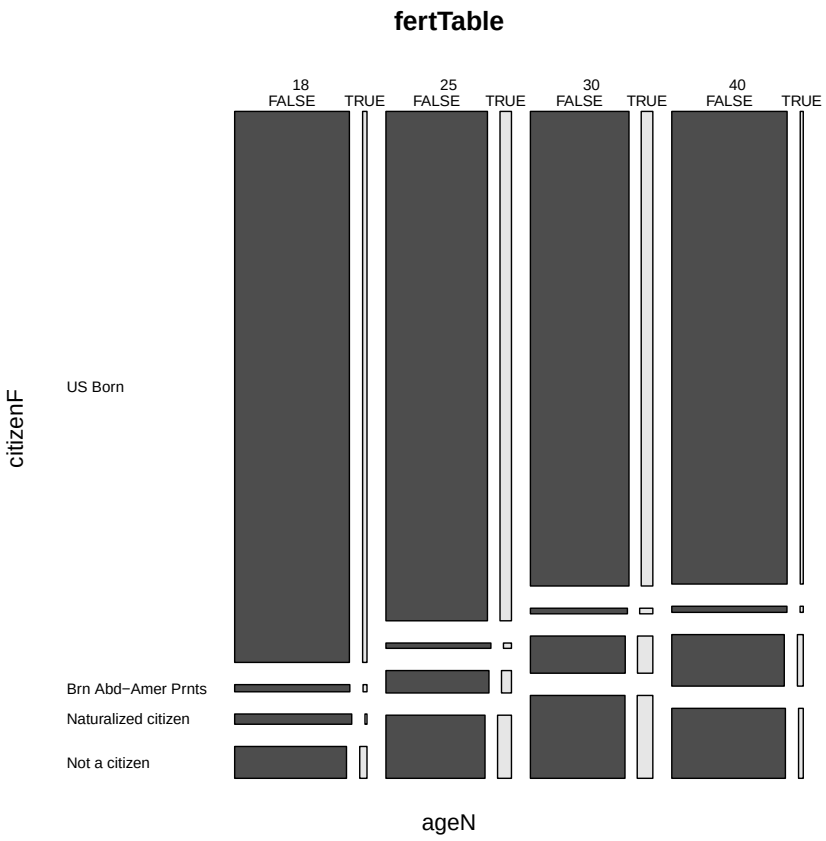
A fancier example of a mosaic plot requires that we come to terms with wights. The `survey` package has some tools for weighting stuff like frequency tables.

```r
##taking account of weights
## The survey package has tools for calculating weighted tables and other stuff
library(survey)

## Warning:  package 'survey' was built under R version 3.3.1
## Loading required package:  grid
## Loading required package:  Matrix
## Loading required package:  survival
##
## Attaching package:  'survey'
## The following object is masked from 'package:graphics':
##
##    dotchart

## all of the "survey" tools require that your data be contained in a data.frame whic
## must then itself be contain ia a "design" object -- which is a list
## object of which one element is your original data.frame
## the other elements of the design object include information on the weights
## and on stratification and whatever else your complicated survey data has.

## Here we create asimple design object -- no strata, weights are made up of a
## single variable. Which reflects how our IPUMS data extract truely is

des<-svydesign(ids=~0,weights=~perwt,data=acs09)
## note the use of ~ this is used to define "formula" in R. It will
## come up again
## note also that des is a list:
is.list(des)

## [1] TRUE

names(des)

## [1] "cluster"    "strata"     "has.strata" "prob"       "allprob"
## [6] "call"       "variables"  "fpc"        "pps"

is.data.frame(des$variables)

## [1] TRUE

dim(des$variables)
```

```
## [1] 705313     40

dim(acs09)

## [1] 705313     40

## svytable is survey's version of table -- in other words it constructs
## frequency tables BUT it understands weights
fertTable<-svytable(~age+citizenF+fertyrL,design=des)
## red face test:
sum(fertTable)

## [1] 75974672

sum(acs09$perwt)

## [1] 75974672

mosaicplot(svytable(~fertyrL+citizenF,design=des),las=1)
```



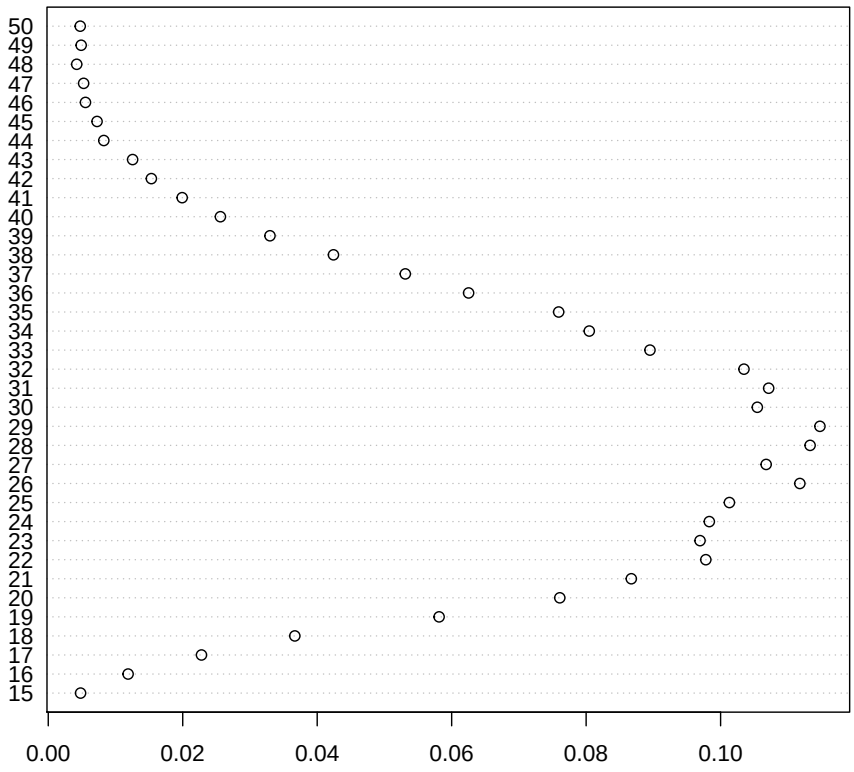

```
## We can also add another dimension such as age to the plot
## the plot is way too cluttered with all the ages we can use the subset
## function to limit our examination to only a few ages
fertTable<-svytable(~ageN+citizenF+fertyrL,
             design=subset(des,ageN %in% c(18,25,30,40)))

## NOTE color not col.  FALSE will give you white boxes.
##
mosaicplot(fertTable ,color=TRUE,las=1)
```



4 Dotplots

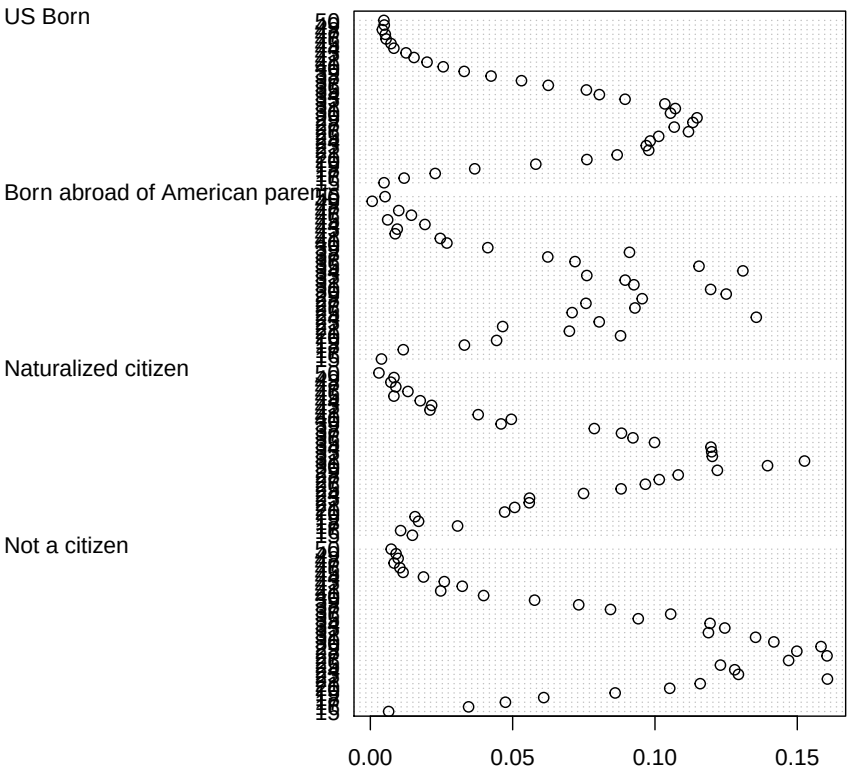
A `dotchart()` is a lot like a bar plot. But in some situations dot charts are easier to understand.

```
## a dotchart is morally eqivalent to a barchart but in some cases
## it is easier compare trends
## here is dotplot of US born ASFRs
dotchart(ASFR[,"US Born"])
```



Unfortunately, the plain old `dotchart()` function does not handle groups in a way that works for me.

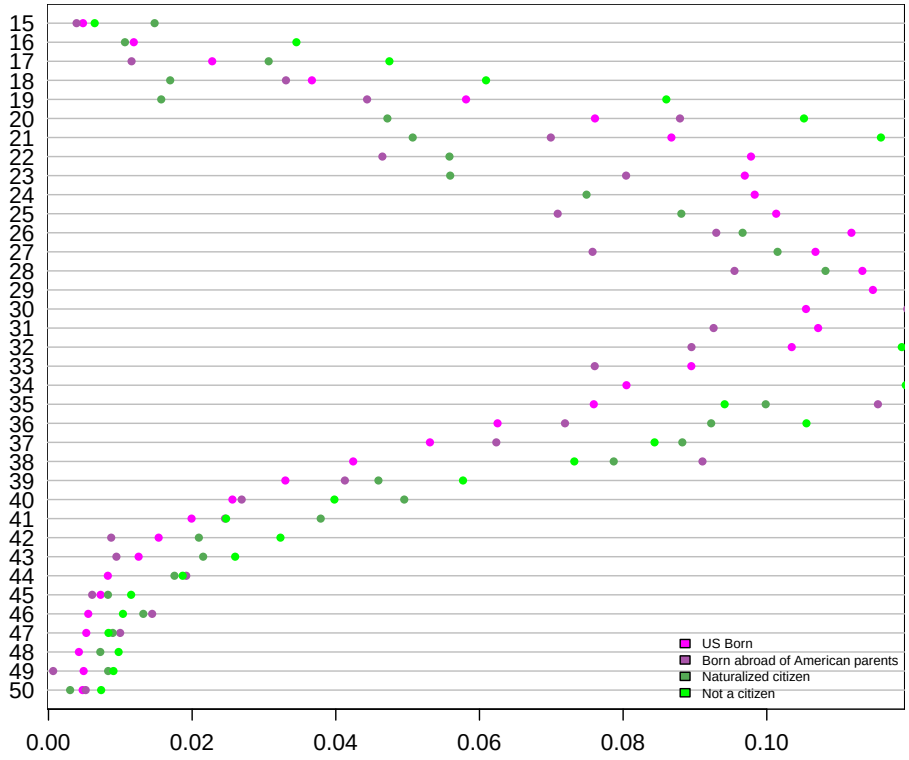
```
## Unfortunately, the vanilla dotchart() does not handle multiple
## categories all that well
dotchart(ASFR) ## like matplot() it does assume that each column
```




```
## is a separate series to plot, but it does not make them very comparable
```

```
## dotchart2() in the Hmisc library has some additional feature one of which
## is "add"
library(Hmisc)

## Loading required package:  lattice
## Loading required package:  Formula
## Loading required package:  ggplot2
##
## Attaching package:  'Hmisc'
```

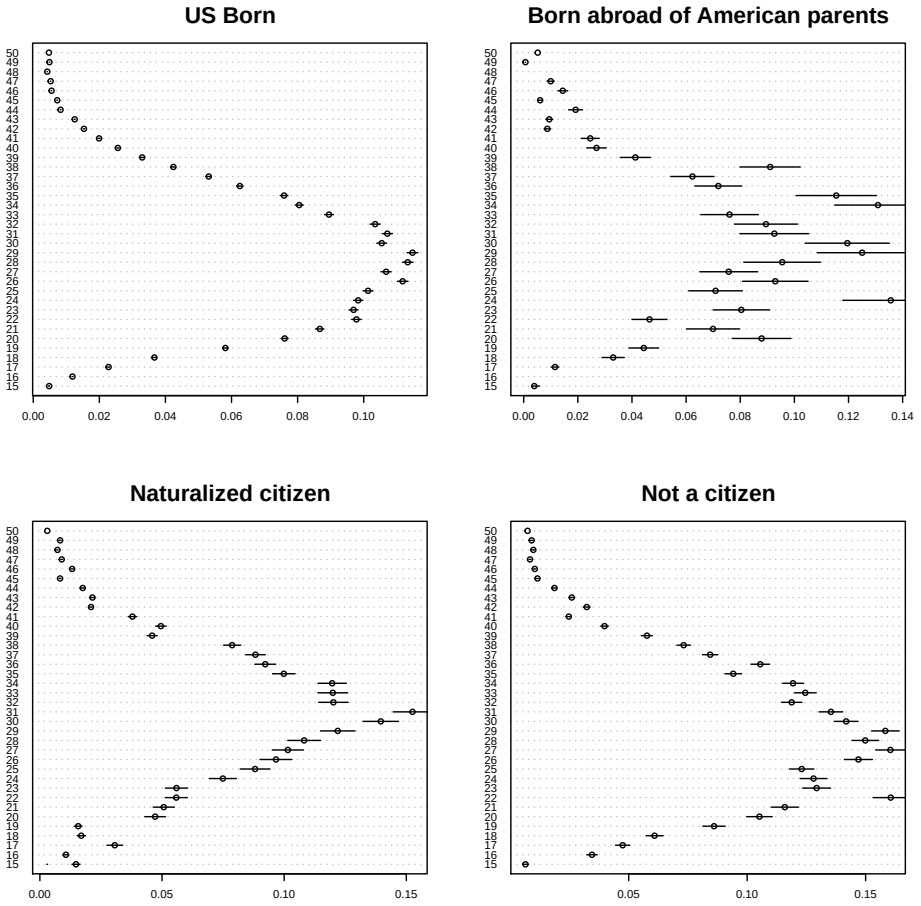
```
## The following object is masked from 'package:survey':
##
##     deff
## The following objects are masked from 'package:base':
##
##     format.pval, round.POSIXt, trunc.POSIXt, units

## sychronize names for col palette
names(pal.citz)<-colnames(ASFR)

dotchart2(ASFR[,"US Born"],col=pal.citz["US Born"],ylim=range(ASFR,na.rm=TRUE))
for(cit in colnames(ASFR)[2:ncol(ASFR)]){
  dotchart2(ASFR[,cit],col=pal.citz[cit],  add=TRUE,yaxt='n')
}

legend(x="bottomright",fill=pal.citz,legend=names(pal.citz),
       cex=.6,bty='n')
```



5 Confidence bounds

Any result with scientific pretensions requires confidence bounds. Fortunately for us, confidence bounds of rates are fairly easy to calculate. Recall from freshman Statistics that the standard error of a proportion is $\sqrt{\frac{\hat{p}(1-\hat{p})}{n}}$ where $\hat{p}$ is the estimated proportion, or in our case the estimated fertility rate. Note that n is the number of observations but since our data are weighted, we need to be careful. If we were to use the the sum of `perwt` we would be pretending that our sample is quite a bit larger than it actually is. This would have the effect of reducing the estimated standard error. We better use the actual number of observations instead.

```r
## the real n uses length rather than sum in tapply
n.real<- tapply(acs09$perwt*0,list(acs09$ageN,acs09$citizenF),length)
## and applying the formula
ASFR.se<-sqrt(((ASFR*(1-ASFR))^2)/n.real)
## 95% confidence bounds are about 1.96 Standard errors around the estimate

ASFR.upper <-ASFR + ASFR.se*1.96
ASFR.lower <-ASFR - ASFR.se*1.96
par(mfrow=c(2,2))
for(cit in colnames(ASFR)[1:ncol(ASFR)]){
  dotchart(ASFR[,cit],ylim=range(ASFR,na.rm=TRUE),cex=.5)
  segments(x0=ASFR.lower[,cit],x1=ASFR.upper[,cit],y0=1:35,y1=1:35)
  title(cit)
}
```


US Born
Born abroad of American parents
Naturalized citizen
Not a citizen