We organise the R files as follows: `main.R` is the main entry point of everything: it sources other files and actually load the data, call the functions, and produces graphs and confidence intervals numbers and so on. Other files are no-side-effect definitions of functions, which should be called in `main.R` – except `readdat.R`, which actually read and preprocesses the CSV files and store the clean data as data frames in the current R environment.

1 Libraries

The following libraries are needed to run this analysis:

⟨*main.R*⟩≡
```
suppressPackageStartupMessages({
  library('stringr')
  library('purrr')
  library('reshape2')
  library('gtools')
  library('lattice')
  library('tibble')
  library('latticeExtra')
  library('RColorBrewer')
  library('outliers')
})
```

2 Configuring file paths

All file paths to the data files goes to `filepath-def.R`. The file should be self-explanatory.

⟨*filepath-def.R*⟩≡
```
deathfile='../dat/eurostat-20210527T1802.csv'
popufile='../dat/2020-06-22_population.csv'
##covidswefile='../dat/covid-sweden-2020-09-14.csv'
##polishfile='../dat/polish-20200924T1733.csv'
##swedcovidregionfile='../dat/Folkhalsomyndigheten_covid19_modified_2021-02-07.rds'
## Härje Widing's COVID-19 death data
coviddeathfiles=list(
  'BE'='../dat/Härje-covid-snapshot-20210531T1502/COVID-19-BELGIUM-level-2.csv',
  'DE'='../dat/Härje-covid-snapshot-20210531T1502/COVID-19-GERMANY-level-2.csv',
  'ES'='../dat/Härje-covid-snapshot-20210531T1502/COVID-19-SPAIN-level-2.csv',
  'CH'='../dat/Härje-covid-snapshot-20210531T1502/COVID-19-SWITZERLAND-level-2.csv',
  'UK'='../dat/Härje-covid-snapshot-20210531T1502/COVID-19-UK-level-2.csv'
)
```

3 Loading the data into the R environment

To load and clean all the data, we source the file `readdat2.R`. This file should produces important data frames in the current R environment such as `mf`, `mf.1`, `mf.2` etc., which contains the excess death per age-group per region. the trailing `.1` and `.2` represents NUTS level. For example, `mf` contains country level such as `SE`; and `mf.1` contains `SE1`, `SE2` and so on.

⟨*main.R*⟩+≡
```
source('readdat2.R')
```

In the remaining of this section, we implement this `readdat2.R`.

3.1 Convert CSV files into suitable R data frame

The `readdat2.R` is layed out as follows:

⟨*readdat2.R*⟩≡
```
⟨Initialise and include libraries⟩
⟨Read EuroStat death and population data⟩
⟨Merge the EuroStat death and population data for convenience⟩
⟨Read COVID-19 death data by Härje Widing⟩
⟨Define country-age-group mapping for COVID-19 data⟩
```

First, we include the libraries and initialisation. Note that we set the `stringAsFactors` option to true to change the behaviour of R's readcsv function. This is labelled deprecated in R, but in 2021 it is still usable.

⟨*Initialise and include libraries*⟩≡
```
suppressPackageStartupMessages({
  library('stringr')
  library('purrr')
  library('reshape2')
  library('tibble')
})

source('utils.R')        # Various utility functions
source('filepath-def.R')
cat('Preprocessing data...\n')
```

The EuroStat death and population data is stored in two big, seperate, CSV files

⟨*Read EuroStat death and population data*⟩≡
```
deathraw = read.csv(deathfile)
popuraw = read.csv(popufile)
# Typo in the CSV file, I guess
deathraw$age[deathraw$age=='NK']   = 'UNK'
deathraw$age[deathraw$age=='OTAL'] = 'TOTAL'
# I want level names in death matches populatuion
for (i in seq_along(deathraw$age))
    deathraw$age[i] = str_replace(deathraw$age[i], '-', '_')
```

We merge the two data sets so that the death and population of the same age-region-week (abbrev. ARW) is on the same row. This is convenient because most of the time we select our data by ARW.

⟨*Merge the EuroStat death and population data for convenience*⟩≡

```
loclvl = function (l) nchar(l) - 2                              # NUTS code to NUTS level
yr     = function (s) str_extract(s, '\\d{4}')                  # Get year from XyyyyWww format
wkno   = function (s) str_remove(str_extract(s, 'W\\d{2}'), 'W') # Get week from XyyyyWww format

mkmf = function (deathraw, popu, lvl) {
  allloc = unique(deathraw$geo.time)     # All available NUTS codes
  loc = allloc[loclvl(allloc) %in% lvl]  # Target NUTS codes
  agegrp = unique(deathraw$age)          # Target age group
  sex    = c('F','M')                            # Target sex. we don't use 'T'
  availmask     = deathraw$age %in% agegrp &
                  deathraw$geo.time %in% loc &
                  deathraw$sex %in% sex
  deathraw      = data.frame(deathraw)           # Copy the data frame before changing it.
  deathraw      = deathraw[availmask,]
  deathraw.mlt    = melt(deathraw, id.vars=c('sex', 'age', 'geo.time'))
  deathraw.mlt$yr   = yr(deathraw.mlt$variable)
  deathraw.mlt$popyr= ifelse(deathraw.mlt$yr == '2020', '2019', deathraw.mlt$yr)
  deathraw.mlt$wkno = wkno(deathraw.mlt$variable)
  popu.mlt = melt(popu, id.vars=c('sex', 'age', 'geo.time'))
  popu.mlt$variable = yr(popu.mlt$variable)
  colnames(popu.mlt)[colnames(popu.mlt) == 'variable'] = 'popyr'

  # Inner-joining the two tables
  tmp.mlt = merge(popu.mlt, deathraw.mlt, by = c('sex','age','geo.time','popyr'),
                  suffixes=c('.popu','.death'))
  mdf = tmp.mlt[tmp.mlt$sex == 'M',];  fdf = tmp.mlt[tmp.mlt$sex == 'F',];
  mdf$sex      = NULL;                  fdf$sex      = NULL;
  mdf$variable = NULL;                  fdf$variable = NULL;
  mdf$popyr    = NULL;                  fdf$popyr    = NULL;
  rm(tmp.mlt); rm(popu.mlt); rm(deathraw.mlt)
  # Inner-join again
  mf.mlt = merge(mdf, fdf,
                 by = c('age','geo.time','yr', 'wkno'),
                 suffixes = c('.m', '.f'))
  # Replace the "90" age group to "90_Inf", for later processing
  mf.mlt[['age']][which(mf.mlt[['age']] == '90')] = '90_Inf'
  mf.mlt
}
mf = mkmf(deathraw, popuraw, 0)
mf.1 = mkmf(deathraw, popuraw, 1)
mf.2 = mkmf(deathraw, popuraw, 2)
mf.3 = mkmf(deathraw, popuraw, 3)
```

On the other hand, Widing's data set is a list of CSV files. The "level 2" CSV contains national-level per age-sex-day COVID-19 mortality. Here we convert it to per-week and also clean up the age group for our purpose.

⟨*Read COVID-19 death data by Härje Widing*⟩≡
```
coviddeathraw = lapply(coviddeathfiles, read.csv2)
⟨Exclude Switzerland and Germany⟩
normalise_härje_data = function (coviddeathraw) {
    lapply(coviddeathraw, function (dframe) {
        dframe = ⟨Normalise age groups in national COVID data⟩
        aggregate(dframe[c('male_covid_death','female_covid_death','total_covid_death')],
                  by  = dframe[c('WeekNo','AgeGrp_min','AgeGrp_max','Year')],
                  FUN = sum)
    })
}
coviddeath = normalise_härje_data(coviddeathraw)
```

In the next section, we will implement the age group normalisation algorithm, as well as other age-group-related processing. But first, notice that there is an error in the Switzerland level 2 data set, in which for the age group "80+" there are two different representations:

AgeGrp_min	AgeGrp_max
"80"	NA
"80+"	"80+"

Also, all other countries has integer age groups but Switzerland has strings. So we simply exclude Switzerland at this moment. Germany, on the other hand, has a problem that, in the EuroStat data set there is no per-age population size; therefore we don't have enough information to interpret the COVID mortality count.

⟨*Exclude Switzerland and Germany*⟩≡
```
coviddeathraw[['CH']] = NULL
coviddeathraw[['DE']] = NULL
```

4 Processing age groups from different data sets

The EuroStat data and COVID death data set has different definitions of age groups. EuroStat has a fine-grained age group, each of which has spans five years, from 0–4, 5–9, 10–14, and so on, until 90+. On the other hand, the COVID-19 data set have different age group for different countries, but fortunately, all of their beginning ages and end ages's second digits are 0, 4, and 9 respectively. In other words, the EuroStat age groups are "sub-partition" of that of the COVID-19 death data for all countries.

Therefore, for each country, we can simply leave the COVID-19 data set's age group 'untouched', but normalised in format, and then later on merge the corresponding EuroStat sub groups' death counts. This results in different age groups in different countries – we may normalise it further later on when we do statistical testing or visualisation– but at least for visualisation, this normalisation gives us the more fine-grained age group that we can get out of the data.

But first, we define an utility function which check if a row exists exactly a data frame.

⟨*utils.R*⟩≡
```
rowexists = function (row, df) nrow(merge(row, df))>0
```

⟨*Normalise age groups in national COVID data*⟩≡

```
{
    ## Produce an data frame of unique age group with two columns:
    ## 'AgeGrp_min' and 'AgeGrp_max'
    ## Remove the row if both min and max are NA
    dframe = data.frame(dframe)
    dframe = dframe[!(is.na(dframe[['AgeGrp_min']]) &
                      is.na(dframe[['AgeGrp_max']])),]
    ## In UK there is an 0~1 and 1~4 group. Merge them in this case.
    dframe.ageonly = dframe[c('AgeGrp_min','AgeGrp_max')]
    if (rowexists(list('AgeGrp_min'=0,'AgeGrp_max'=1), dframe.ageonly) &&
        rowexists(list('AgeGrp_min'=1,'AgeGrp_max'=4), dframe.ageonly)){
        which_zeroone = which(dframe[['AgeGrp_min']] == 0 &
                              dframe[['AgeGrp_max']] == 1)
        which_onefour = which(dframe[['AgeGrp_min']] == 1 &
                              dframe[['AgeGrp_max']] == 4)
        dframe[which_zeroone,'AgeGrp_max']      = 4
        dframe[which_onefour,'AgeGrp_min'] = 0   # We'll sum them up later
    }
    ## If AgeGrp_max is NA and min is not then replace NA with infinity
    which_veryold = which((!is.na(dframe[['AgeGrp_min']])) &
                          is.na(dframe[['AgeGrp_max']]))
    dframe[['AgeGrp_max']][which_veryold] = Inf
    ## If AgeGrp_min is NA and max is not then replace NA with 0
    which_baby = which((!is.na(dframe[['AgeGrp_max']])) &
                       is.na(dframe[['AgeGrp_min']]))
    dframe[['AgeGrp_min']][which_baby] = 0
    dframe
}
```

It is also convenient to have a list which contains all the age groups of various countries, so the testing or visualisation routine can find use different age group depending on the country.

⟨*utils.R*⟩+≡

```
unique_row = function (df) df[!duplicated(df),]
```

⟨*Define country-age-group mapping for COVID-19 data*⟩≡

```
country_age_group_map = lapply(coviddeath, function (dframe) {
    X = unique_row(dframe[c('AgeGrp_min','AgeGrp_max')])
    rownames(X) = NULL
    X
})
```

Because the EuroStat data has different age group, sooner or later we will need to combine the sub-partition of EuroStat according to the country level age group. The following is a high-level function which, given a COVID-19 age group (one of the elements in the variable `country_age_group_map`), returns the corresponding EuroStat age groups.

⟨*utils.R*⟩+≡

```
agegrp_tostr = function (minag,maxag)
    paste(minag, maxag, sep='_')

make_eurostat_age_group_map = function (coarse_map) {
    minag = (0:18)*5
    maxag = minag + 4
    maxag[length(maxag)] = Inf
    usedmask = integer(length(minag))
    result = list()
    for (j in seq_len(nrow(coarse_map))) {
        included = list()
        for (i in seq_along(minag)) {
            if (coarse_map[j,'AgeGrp_min'] <= minag[i] && coarse_map[j,'AgeGrp_max'] >= maxag[i]) {
                included[[length(included)+1]] = agegrp_tostr(minag[i], maxag[i])
                usedmask[i] = 1
            }
        }
        result[[j]] = unlist(included)
    }
    if (prod(usedmask) != 1)
        warning('make_eurostat_age_group_map: Supplied age group does not span all possible ages')
    names(result) = mapply(agegrp_tostr, coarse_map[['AgeGrp_min']], coarse_map[['AgeGrp_max']])
    result
}
```

5 Iterators to Loop through Pairs of Binomial Data

The tests we conduct have a general structure. Fixing an ARW combination, let Y_i, Y_C, Y^* be, respectively, the all-cause male death count of Year i in which COVID-19 were absent, male death count officially announced as due to COVID-19, and the amount by which the male death count of 2020 exceeds that of a baseline. Also let n_i, n_C, and n^* be the corresponding total, sex-aggregated death count (male plus female); with, optionally, (m_M, m_F) and (r_M, r_F), which is the male and female population size of the excess death's population and the COVID-19 death's population.

The file `arw-iterators.R` defines high-order functions which calls

$$f(Y_1, \cdots, Y_n, n_1, \cdots, n_n, Y_C, n_C, Y^*, , n^*, r_M, r_F, m_M, m_F) \tag{1}$$

for some f and returns a list of lists, each of which contains the ARW and the function's evaluation result.

⟨*arw-iterators.R*⟩≡

```
iter_ageweek = function (target_geo, get_m, get_r,
                         get_history_death, get_covid_death, get_excess_death,
                         fn,
                         age_groups, ...) {
  excess_nonpos   = list() # These 3 var. contains all error ARW.
  excess_notavail = list()
  covid_notavail  = list()
  wkno_uniq = as.character(1:52)
  results   = list()
  for (age in age_groups) {
    for (wk in wkno_uniq) {
      covidgrp_popsiz.m    =get_r(target_geo, age, wk, 'M')
      covidgrp_popsiz.f    =get_r(target_geo, age, wk, 'F')
      excessgrp_popsiz.m   =get_m(target_geo, age, wk, 'M')
      excessgrp_popsiz.f   =get_m(target_geo, age, wk, 'F')
      ## The following are vectors of numbers, each element represent the
      ## death count of a year. Year is encoded as names of the vectors
      death.m.history      =get_history_death(target_geo, age, wk, 'M')
      death.f.history      =get_history_death(target_geo, age, wk, 'F')
      death.m.excess       =get_excess_death(target_geo, age, wk, 'M')
      death.f.excess       =get_excess_death(target_geo, age, wk, 'F')
      death.m.covid        =get_covid_death(target_geo, age, wk, 'M')
      death.f.covid        =get_covid_death(target_geo, age, wk, 'F')
      ⟨Check death toll consistency⟩
      obj = fn(death.m.history,   death.m.history + death.f.history,
               death.m.covid,     death.m.covid + death.f.covid,
               death.excess,      death.excess,
               covidgrp_popsiz.m, covidgrp_popsiz.f,
               excessgrp_popsiz.m, excessgrp_popsiz.f,
               ...)
      info = list(age = age, geo = target_geo, wk = wk, result=obj)
      results[[length(results)+1]] = info
    }
  }
```

```
  class(results) = c('iter_ageweek_result')
  attr(results,'miss') = list(excess_nonpos=excess_nonpos,
                              excess_notavail=excess_notavail,
                              covid_notavail=covid_notavail)
  results
}
```

Sometimes excess death can be negative, depending on what baseline one chooses; or it can be NA if the data set itself contains NA for whatever reasons. When this happens, we want to record them and continue to next iteration.

⟨*Check death toll consistency*⟩≡

```
if (is.na(death.m.excess) || is.na(death.f.excess)) {
  excess_notavail[[length(excess_notavail)+1]] = c(wk = wk, age = age)
  cat(sprintf('Excess death not available, skip (wkno, age) = (%s,%s) \n', wk, age))
  next
} else if (!(death.m.excess > 0 && death.f.excess > 0)) {
  excess_nonpos[[length(excess_nonpos)+1]] = c(wk = wk, age = age)
  cat(sprintf('Non-positive excess, skip (wkno,age,excess.m,excess.f) = (%s,%s,%f,%f)\n',
              wk, age, death.m.excess, death.f.excess))
  next
}
if (is.na(death.m.covid) || is.na(death.f.covid)) {
  covid_notavail[[length(covid_notavail)+1]] = c(wk = wk, age = age)
  cat(sprintf('COVID-19 death not available, skipping (wkno, age) = (%s,%s) \n', wk, age))
  next
}
```

5.1 Extract death counts from the pre-processed Eurostat data frame

In order to use the `iter_ageweek` function defined above, it is neccessary to supply it with the `get_*` functions.

⟨*arw-iterators.R*⟩+≡

⟨*Define get historical death function for EuroStat*⟩

⟨*Define get excess death function for EuroStat*⟩

⟨*Define get m function for EuroStat*⟩

where "get m" m refers to the population size of the excess population, as discussed above.

The Eurostat data frames `mf.*` is sufficient to provide the functions `get_history_death` and `get_excess_death`.

Now let's define a function which get returns the historical death count. The names `eshist` stands for Eurostat history.

⟨*Define get historical death function for EuroStat*⟩≡

```
mk_eshist_getter = function (mf, agegrp_map, fromwhichyear = 2020) {
    function (target_geo, age, wk, sex) {
        idx = which(mf$age %in% agegrp_map[[age]] &
                    mf$geo.time == target_geo &
                    mf$wkno == sprintf('%02d', as.integer(wk)) &
                    as.integer(mf$yr) < fromwhichyear)
        if (length(idx) == 0) { return(NA); }
        key = if (sex == 'M') 'value.death.m' else 'value.death.f'
        mfaggr = aggregate(mf[[key]][idx], list(mf[['yr']][idx]),
                           sum, SIMPLIFY=T)
        yrorder = order(mfaggr[['Group.1']])
        result = mfaggr[['x']][yrorder]
        yrname = paste0('Y', mfaggr[['Group.1']][yrorder])
        names(result) = yrname
        result
    }
}
```

Note that the `mf.*` data frames from `readdat.R` instead of `mfexc.*` should be passed to the above function. And the excess death data is similar. Notice that the result is rounded to the nearest integer.

⟨*Define get excess death function for EuroStat*⟩≡

```
mk_esexcess_getter = function (mf, agegrp_map, whichyear=2020, baseline_fn=mean) {
    get_history = mk_eshist_getter(mf)
    function (target_geo, age, wk, sex) {
        baseline = baseline_fn(get_history(target_geo, age, wk, sex))
        if (is.na(baseline)) return(NA)
        idx = which(mf$age %in% agegrp_map[[age]] &
                    mf$geo.time == target_geo &
                    mf$wkno == sprintf('%02d', as.integer(wk)) &
                    as.integer(mf$yr) == whichyear)
        if (length(idx) == 0) { return(NA); }
        key = if (sex == 'M') 'value.death.m' else 'value.death.f'
        death_total = sum(mf[[key]][idx])
        round(death_total - baseline)
    }
}
```

The Eurostat database contains the population size of both male and female at all NUTS regions. This information is stored in `popuraw` data frame produced by `readdat.R`. We can use this to implement the `get_m` argument of `iter_ageweek`.

⟨*Define get m function for EuroStat*⟩≡
```
mk_m_getter = function (mf, agegrp_map, whichyear=2020) {
    function (target_geo, age, wk, sex) {
        idx = which(mf$age %in% agegrp_map[[age]] &
                    mf$geo.time == target_geo &
                    mf$wkno == sprintf('%02d', as.integer(wk)) &
                    as.integer(mf$yr) == whichyear)
        if (length(idx) == 0) { return(NA); }
        key = if (sex == 'M') 'value.popu.m' else 'value.popu.f'
        total = sum(mf[[key]][idx])
        total
    }
}
```

5.2 Extract death counts from the pre-processed COVID-19 data frame

Because of the different data sources, different countries might need a different processing functions. But in this GitHub repository by Härje Widing, the format of these data are normalised from all the data sources except for the NUTS code.

5.2.1 Sweden

Due to the lack of clean regional data, we will use the national COVID-19 mortality first. Here we implement the `get_covid_death` and `get_r` funcions needed by the iterator using the national-level data.

⟨*arw-iterators.R*⟩+≡
⟨*Define get covid death function*⟩
⟨*Define get r function*⟩

First, we define a utility function which converts any regional-level NUTS codes to country code. Say, "SE2" should map to "SE", and so on.

⟨*utils.R*⟩+≡
```
nationalise_NUTS = function (code)
    substr(code, start = 1, stop = 2)
```

Also it is convenient to have a function which splits age group strings like "90_Inf" to `c(90,Inf)`.

⟨*utils.R*⟩+≡
```
agegrp_tonum = function (agegrp_str) {
    ans = sapply(strsplit(agegrp_str, '_')[[1]], as.numeric, simplify=T)
    names(ans) = c('AgeGrp_min','AgeGrp_max')
    ans
}
```

In the following functions, all NUTS codes are converted to national-level in this manner. These data-getters does not need to be supplied with age group, as it expects the argument `age` matches the actual age group.

⟨*Define get covid death function*⟩≡

```
mk_national_covid_death_getter = function (coviddeath, whichyear=2020) {
    function (target_geo, age, wk, sex) {
        national_geo = nationalise_NUTS(target_geo)
        if (is.null(coviddeath[[national_geo]])) {
            stop(sprintf('We do not have national-level COVID mortality of "%s"',
                         national_geo))
        }
        agegrp_num = agegrp_tonum(age)
        whichrow = which(coviddeath[[national_geo]][['AgeGrp_min']] == agegrp_num[['AgeGrp_min']] &
                         coviddeath[[national_geo]][['AgeGrp_max']] == agegrp_num[['AgeGrp_max']] &
                         coviddeath[[national_geo]][['WeekNo']] == wk &
                         coviddeath[[national_geo]][['Year']] == whichyear)
        if (length(whichrow) == 0) {
            stop(sprintf(
                  'covid_death_getter: no record for (Geo, Age_min, Age_max, Week, Year) = (%s,%s,%s,%s,%s)',
                  target_geo, agegrp_num[['AgeGrp_min']], agegrp_num[['AgeGrp_max']], wk, whichyear))
        } else if (length(whichrow) > 1) {
            stop(sprintf(
                  'covid_death_getter: >1 records for (Geo, Age_min, Age_max, Week, Year) = (%s,%s,%s,%s,%s)',
                  target_geo, agegrp_num[['AgeGrp_min']], agegrp_num[['AgeGrp_max']], wk, whichyear))
        }
        key = if (sex == 'M') 'male_covid_death' else 'female_covid_death'
        coviddeath[[national_geo]][[key]][whichrow]
    }
}
```

Similar to the "get_m" function, we define the "get_r" function which returns the population size. But because this function uses the EuroStat population size, there is a need to pass it the age group map. The function takes, as arguments, the COVID-19 age group, maps it to EuroStat age group, and return the total of population size. Any non-country-level NUTS code passed to the function will be converted to country-level; in other words, the returned number will always be the national-level population size.

⟨*Define get r function*⟩≡

```
mk_national_r_getter = function (mf, agegrp_map, whichyear=2020) {
    function (target_geo, age, wk, sex) {
        national_geo = nationalise_NUTS(target_geo)
        idx = which(mf$age %in% agegrp_map[[age]] &
                    mf$geo.time == national_geo &
                    mf$wkno == sprintf('%02d', as.integer(wk)) &
                    as.integer(mf$yr) == whichyear)
        if (length(idx) == 0) { return(NA); }
        key = if (sex == 'M') 'value.popu.m' else 'value.popu.f'
        total = sum(mf[[key]][idx])
        total
    }
}
```

5.3 Testing out the iterator

The following is a test to see if the iterator actually works.

⟨*TEST-iter.R*⟩≡

```
source('arw-iterators.R')
iter_ageweek('SE1', get_m, get_r,
             get_history_death, get_covid_death, get_excess_death,
             fn,
             age_group_map = EUROSTAT_AGEGROUP_MAP, ...)
```

6 Bayesian test of sex ratio

⟨*bayesratio.R*⟩≡

```
## set.seed(777)
set.seed(5201314)

source('filepath-def.R')
source('sedat.R')
source('mixture-ci.R')

suppressPackageStartupMessages({
    library('lattice')
    library('latticeExtra')
})
## TODO: write this!
```

7 Confidence interval of the mixture model

The mixture code is in the file `mixture-ci.R`.

⟨*mixture-ci.R*⟩≡

```
suppressPackageStartupMessages({
  library(lattice)
  library(latticeExtra)
  library(RColorBrewer)
  library(parallel)
})
Lstat = Vectorize(function (p,alpha,nc=100,ns=130,nsamp=400000) {
  yc = rbinom(n=nsamp, size=nc, p=p)
  ys = rbinom(n=nsamp, size=ns, p=p)
  Tall = (yc+ys)/(nc+ns)
  logTall = log(yc+ys) - log(nc+ns)
  Tc = yc/nc
  logTc = log(yc)-log(nc)
  Ts = ys/ns
  logTs = log(ys)-log(ns)
  logL = yc * (logTc - logTall) + ys*(logTs - logTall) +
    (nc-yc)*(log(1-Tc) - log(1-Tall)) +
    (ns-ys)*(log(1-Ts) - log(1-Tall))
  k = quantile(logL, probs=alpha,na.rm=T)
  names(k) = NULL
  k
}, 'p')

# xs = seq(0.08, 0.92, length.out=200)
# Q = Lstat(xs,0.95)
# crit = max(Q)
# plot(y=Q, x=xs, type='l', xlab = expression(p_C), ylab='95th quantile of lik. ratio stat.')

pwrfn = (function (pn, pc, alpha, c, nc=100, ns=130, nsamp=100000) {
  yc = rbinom(n=nsamp, size=nc, p=pc)
  ys = rbinom(n=nsamp, size=ns, p=alpha*pn + (1-alpha)*pc)
  Tall = (yc+ys)/(nc+ns)
  logTall = log(yc+ys) - log(nc+ns)
  Tc = yc/nc
  logTc = log(yc)-log(nc)
  Ts = ys/ns
  logTs = log(ys)-log(ns)
  logL = yc * (logTc - logTall) + ys*(logTs - logTall) +
        (nc-yc)*(log(1-Tc) - log(1-Tall)) +
        (ns-ys)*(log(1-Ts) - log(1-Tall))
  sum(logL>c)/nsamp
})
cmapply <- function(FUN, ..., MoreArgs = NULL, SIMPLIFY = TRUE,
                    USE.NAMES = TRUE)
```

```
{
  l <- expand.grid(..., stringsAsFactors=FALSE)
  r <- do.call(mapply, c(
    list(FUN=FUN, MoreArgs = MoreArgs, SIMPLIFY = SIMPLIFY, USE.NAMES = USE.NAMES),
    l
  ))
  if (is.matrix(r)) r <- t(r)
  cbind(l, r)
}

##pwr = cmapply(pn=seq(0.15, 0.85, by=0.05), pc=seq(0.15,0.85,by=0.04), alpha=seq(-0.5,1.5,by=0.1),
##         FUN= function (pn, pc, alpha) {
##            print(c(pn,pc,alpha))
##            pwrfn(pn, pc, alpha, c=1.98)
##          })
##colnames(pwr) = c('pN', 'pC', 'alpha', 'pwr')

# my.settings <- canonical.theme(color=FALSE)
# my.settings[['strip.background']]$col <- 'white'
# my.settings[['strip.border']]$col<- 'black'
# levelplot(pwr~pC*alpha|pN, data=pwr, panel = panel.2dsmoother, col.regions = colorRampPalette(brewer.pal(11,'Spect
#           par.settings = my.settings,
#           strip = strip.custom(strip.levels = c(TRUE, TRUE)),
#           par.strip.text=list(col='black'))
#
#
#
# xs2 = seq(0.08, 0.92, length.out=200)
# Q2 = Lstat(xs2,0.95,nc=400,ns=600,nsamp=400000)
# crit2 = max(Q2)
# plot(y=Q2, x=xs2, type='l', xlab = expression(p_C), ylab='95th quantile of lik. ratio stat.')
# pwr2 = cmapply(pn=seq(0.15, 0.85, by=0.05), pc=seq(0.15,0.85,by=0.04), alpha=seq(-0.5,1.5,by=0.1),
#               FUN= function (pn, pc, alpha) {
#                 print(c(pn,pc,alpha))
#                 pwrfn(pn, pc, alpha, c=1.96, nc=400, ns=600)
#               })
# colnames(pwr2) = c('pN', 'pC', 'alpha', 'pwr')
# levelplot(pwr~pC*alpha|pN, data=pwr2, panel = panel.2dsmoother, col.regions = colorRampPalette(brewer.pal(11,'Spec
#           par.settings = my.settings,
#           strip = strip.custom(strip.levels = c(TRUE, TRUE)),
#           par.strip.text=list(col='black'))

loglik_mainpart = function (pn,pc,a,yn,yc,ys,nn,nc,ns) {
  yn*log(pn) + (nn-yn)*log(1-pn) + yc*log(pc) + (nc-yc)*log(1-pc) + ys*log(a*pn+(1-a)*pc) +
      (ns-ys)*log(1-a*pn-(1-a)*pc)
}
mle_alpharestricted = function (a0,yn,yc,ys,nn,nc,ns) {
  obj = function (p) - loglik_mainpart(p[1],p[2],a0,yn,yc,ys,nn,nc,ns)
```

```
  res = simpleError('Encountered infinite value?')
  counter <- 1
  max_tries <- 1000
  while(inherits(res, 'error') & counter < max_tries) {
    res <- tryCatch({ optim(runif(2,min=0.001,max=0.999),
                            obj,
                            method='L-BFGS-B',
                            lower=c(0.001,0.001),upper=c(0.999,0.999)) },
                    error = function(e) e)
    counter <- counter + 1
  }
  if (inherits(res, 'error'))
    stop('Cannot optimise alpha-restricted likelihood')
  else
    res
}

logL_alpharestricted = function (a0,yn,yc,ys,nn,nc,ns) {
  mle_H0_optobj = tryCatch({
    mle_alpharestricted(a0,yn,yc,ys,nn,nc,ns)
  }, error = function (cond) { NULL })
  if (is.null(mle_H0_optobj)) return(NA)
  if (mle_H0_optobj[['convergence']] != 0) return(NA)
  mle    = loglik_mainpart(yn/nn, yc/nc, (nn*(ns*yc - nc*ys))/(ns*(nn*yc - nc*yn)), yn,yc,ys,nn,nc,ns)
  mle + mle_H0_optobj$value
}

critval_logL_alpharestricted = function (a0,nn,nc,ns, nsamp=3000) {
  ## Estimate of distribution of logL_alpharestricted under different values of the null hypothesis.
  ## For all possible pN and pC, simulate logL_alpharestricted and take quantiles.
  cmapply(pn=seq(0.08, 0.92, by=0.05), pc=seq(0.08, 0.92,by=0.04),
          FUN= function (pn, pc) {
            print(c(pn,pc))
            yn = rbinom(n=nsamp, size=nn, p=pn)
            yc = rbinom(n=nsamp, size=nc, p=pc)
            ys = rbinom(n=nsamp, size=ns, p=a0*pn+(1-a0)*pc)
            S = mapply(function (yn,yc,ys) {
              tryCatch({
                logL_alpharestricted(a0,yn,yc,ys,nn,nc,ns)
              }, error=function (cond) NA)
            },yn,yc,ys)
            quantile(S, prob = 0.95, na.rm=T)
          })
}

# nntest = 600;  yntest = 300
# nctest = 600;  yctest = 550
# nstest = 200;  ystest = 170
```

```
#levelplot(r~pn*pc, data=ct)


## Given an observed Y and n, compute an approximate
## confidence interval from the data.
alpha_ci = function (lvl, yn,yc,ys,nn,nc,ns, fineness=1000) {
  a0_candidates = seq(-0.3, 1.5, length.out = fineness)
  critval = qchisq(lvl, df=1)/2
  rejected = integer(fineness)
  for (i in seq_along(a0_candidates)) {
    LL = logL_alpharestricted(a0_candidates[i], yn,yc,ys,nn,nc,ns)
    rejected[i] = as.integer(LL > critval)
  }
  whichzero = which(rejected==0)
  lowerbidx = whichzero[1]
  upperbidx = whichzero[length(whichzero)]
  c(lower = a0_candidates[lowerbidx],
    upper = a0_candidates[upperbidx])
}
# rej = alpha_ci(0.95, yntest,yctest,ystest,nntest,nctest,nstest)

## Find coverage probability
covg_prob = function (lvl,pn,pc,a,nn,nc,ns, nsamp=600) {
  yn = rbinom(n=nsamp, size=nn, p=pn)
  yc = rbinom(n=nsamp, size=nc, p=pc)
  ys = rbinom(n=nsamp, size=ns, p=a*pn+(1-a)*pc)
  edgecases = (yn == 0) | (yn == nn) | (yc == 0) | (yc == nc) | (yn/nn == yc/nc)
  yn = yn[!edgecases]
  yc = yc[!edgecases]
  ys = ys[!edgecases]
  len_actual = sum(!edgecases)
  i = 0
  covg = mcmapply(function (yn,yc,ys) {
    print(c(finished=i, total=nsamp))
    i <<- i+1
    ci = alpha_ci(lvl, yn,yc,ys,nn,nc,ns)
    (a > ci[1]) || (a < ci[2])
  },yn,yc,ys, mc.cores=6)
  sum(covg,na.rm=T)/len_actual
}
# covg_prob(0.95, 0.55,0.85,0.9,nntest,nctest,nstest)
# cprob_all = sapply(seq(0,1,length.out=20), function (alpha) {
#   print(alpha)
#   covg_prob(0.95, 0.55,0.85,alpha,nntest,nctest,nstest)
# })
```