Expansion

Zachary Levine, Jason Pekos, Ariel Pillitteri

08/04/2021

"Mathematical Models for Assessing Vaccination Scenarios in Several Provinces in Indonesia N. Nuraini, K. Khairudin, P. Hadisoemarto, H. Susanto, A. Hasan, and N. Sumarti"

A version of this submission with code hidden for easier reading was submitted alongside this.

First, Zach modified our initial model to incorporate a robust awareness function, in which people become more aware — and therefore have lower transmissibility — when people from their same age cohort are catching COVID at high rates. This adds dampened oscillation to the model; high spikes in fatality and new infections are followed by large decreases in case growth.

This was incorporated when we noticed that the initial model was a poor fit for real world data: the python optimizer from last time was fully implemented, leading to poor fits to data.

It turns out our fitted infection curves totally failed to capture the underlying dynamics in the simpler model. This is largely because — following the initial spike in deaths during the spring — transmisibility stayed low due to public panic, only for it to creep up again in the fall. This is not captured well in a standard SIR model, and so this extension is about quantifying this problem and dealing with it.

Expanding the old gradient functions from the first replication assignment to incorporate awareness, we get the following full model output:

```
library("deSolve")
library("tibble")
theN <- 3565000
make_params <- function(Beta = 0.4215,
                        N = theN,
                        gamma = 0.0876,
                        delta = 0.0028,
                        ##Awarness constant.
                        delta_c = 1/N,
                        ##For each age class
                        deltas = c(0.15, 0.16, 0.627, 1.2, 3.1),
                        v = NULL,
                        q = 0.2,
                        nu = 0.85,
                        ##Sharpness constant
                        k = 3,
                        reinf = 0.05,
                        mu = 0,
```

```
                        natural_death = 0
                        ){
  return(c("Beta" = Beta,
              "gamma" = gamma,
              "delta" = delta,
               "N" = N,
              ##N * delta_c = 50,
              "delta_c" = delta_c,
              "q" = q,
              "nu" = nu,
              "k" = k,
            "deltas" = deltas,
              "reinf" = reinf,
              "mu" = mu,
             "natural_death" = natural_death))
}
make_inits <- function(S = theN - I,
                       I = 100,
                       Q = 0,
                       R = 0,
                       D = 0,
                       N = theN){
    return(c(
      "S" = S,
      "I" = I,
      "Q" = Q,
      "R" = R,
      "D" = D))
}
make_beta <- function(betavals = c(0.0251, 0.0355, 0.122, 0.1698, 0.4488)){
  contact_matrix <- matrix(data = 5, nrow = 5, ncol = 5)
  Beta <- matrix(data = NA, nrow = 5, ncol = 5)
  for (i in 1:5){
    for (j in 1:5){
      Beta[[i,j]] <- betavals[[i]]*contact_matrix[i,j]
    }
  }
  return(Beta)
}
make_age_inits <- function(I0= c(5, 45, 35, 25, 35), N = theN){
  ##Cite Dr. Bolker's Feedback on Milestone 1
  n_age <- length(I0)
 state <- unlist(c(S=(N-I0)/n_age, I=I0, Q=rep(0,n_age), R=rep(0, n_age),
D=rep(0,n_age)))
 return(state)

}
paramNames <- c("Beta", "gamma", "delta", "N", "q", "nu", "reinf", "mu", "natural_death")
stateNames <- c("S",  "I", "Q", "R", "D")
run_sim <- function(params = make_params(),
                       use_awareness = FALSE,
                       Beta1 = 0.451,
                       Beta2 = 0.655,
```

```
                    Beta3 = 0.5122,
                    Beta4 = 0.71698,
                    Beta5 = 0.7488,
                    betavals = c(Beta1, Beta2, Beta3, Beta4, Beta5),
                    ##A function of time for the vaccination strategy.
                    vaccine_strategy = NULL,
                    use_age_structure = TRUE,
                    age_init = make_age_inits(),
                    init = make_inits(),
                    length = 100){
##First decide vaccine strategy.
if (!is.null(vaccine_strategy)){
    v_func <- vaccine_strategy
}
else{
  ##Return the null vaccination strategy.
  v_func <- function(t){
    return(0)
    }
}
if (use_age_structure){
  if (!use_awareness){
    theModel <- function(time, state, parameters){
    S <- as.matrix(state[1:5])
    I <- as.matrix(state[6:10])
    Q <- as.matrix(state[11:15])
    R <- as.matrix(state[16:20])
    D <- as.matrix(state[21:25])
    with(as.list(c(parameters)), {
      v <- v_func(time)
      v_hat <- nu * v
    theBeta <-  make_beta(betavals = betavals)
    beta_I_sums <- numeric(5)
    for (i in 1:5){
      beta_I_sums[i] <- sum(theBeta[i,]%*%as.matrix(I/N))
    }
    dS <-  natural_death * (N - D) + reinf*R - (beta_I_sums + nu*v + mu)*S
    dI <- beta_I_sums*S - (q + mu) *I
    dQ <- q*I - (gamma + delta + mu) * Q
    dR <- gamma*Q + nu*v*S - (reinf+mu) * R
    dD <- delta * Q
    return(list(c(dS, dI, dQ, dR, dD)))
    })
  }
  }
  else{
   ##Use age structure and awareness.
    theModel <- function(time, state, parameters){
    S <- as.matrix(state[1:5])
    I <- as.matrix(state[6:10])
    Q <- as.matrix(state[11:15])
    R <- as.matrix(state[16:20])
    D <- as.matrix(state[21:25])
```

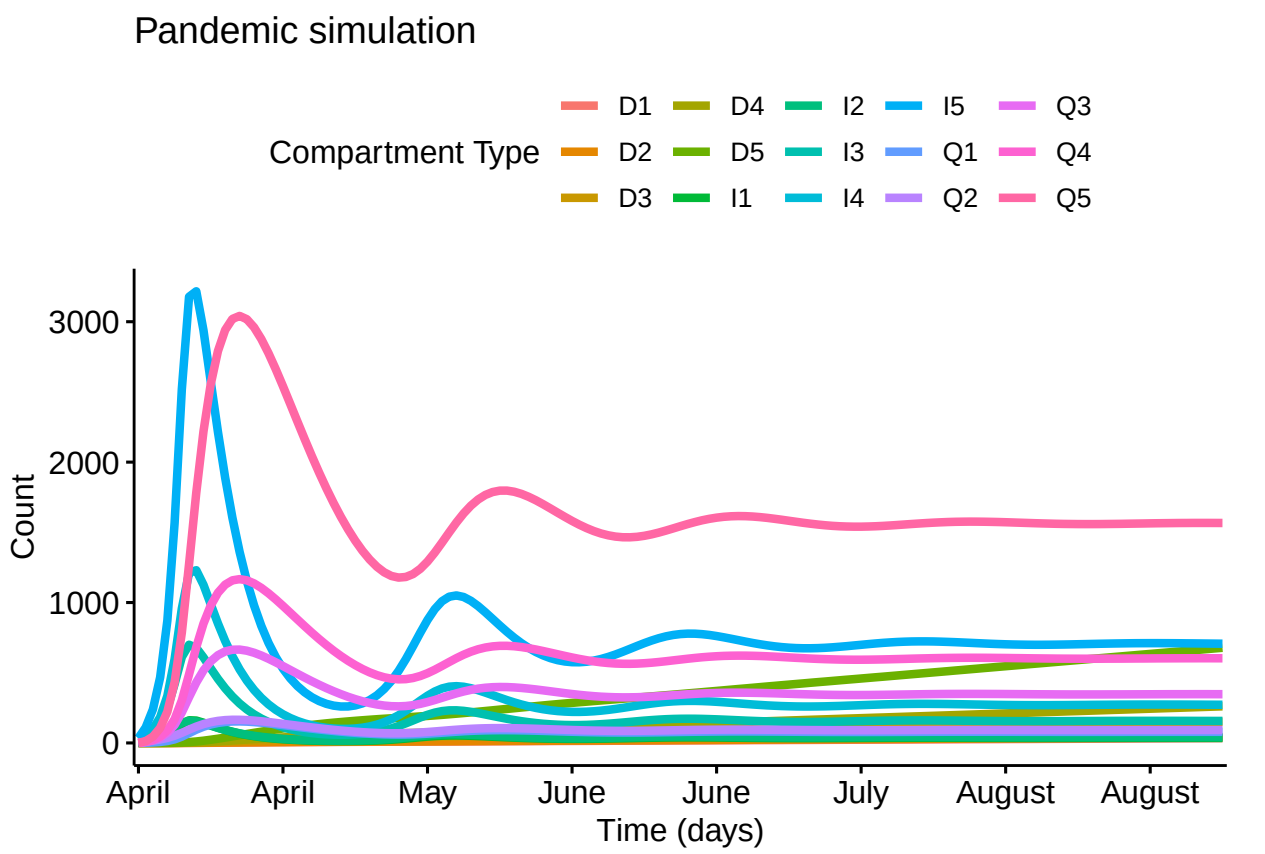
```
      with(as.list(c(parameters)), {
        v <- v_func(time)
        v_hat <- nu * v
      theBeta <-  make_beta(betavals = betavals)
      beta_I_sums <- numeric(5)
      ##Do this first
      dD <- delta * Q
    ##Cite Dr. Bolker's Milestone 1 Feedback.
    keep <-  grepl(names(parameters), pattern = paste("deltas",collapse="|"))
    deltas <- parameters[keep]
      for (i in 1:5){
        beta_I_sums[i] <- sum(theBeta[i,]%*%as.matrix(I/(N)))/((1+(dD[i]/deltas[i])^k))
      }
      dS <-  natural_death * (N - D) + reinf*R - (beta_I_sums + nu*v + mu)*S
      dI <- beta_I_sums*S - (q + mu) *I
      dQ <- q*I - (gamma + delta + mu) * Q
      dR <- gamma*Q + nu*v*S - (reinf+mu) * R
      return(list(c(dS, dI, dQ, dR, dD)))
      })
    }
    }
  }
  else{
    if (!use_awareness){
        theModel <- function(time, state, parameters) {
      with(as.list(c(state, parameters)), {
        v <- v_func(time)
        v_hat <- nu * v
        dS <-  natural_death*(N - D) + reinf*R - (Beta*(I)/(N) + nu*v + mu)*S
        dI <- Beta*I*S / N - (q + mu) *I
        dQ <- q *I - (gamma + delta + mu) * Q
        dR <- gamma*Q + nu*v*S - (reinf+mu) * R
        dD <- delta * Q
        return(list(c(dS, dI, dQ, dR, dD)))
      })
    }
    }
    else{
      theModel <- function(time, state, parameters) {
      with(as.list(c(state, parameters)), {
        v <- v_func(time)
        v_hat <- nu * v
        dD <- delta * Q
        dS <-  natural_death*(N - D) + reinf*R - (Beta*(I)/(N*(1+(dD/delta_c)^k)) + nu*v + mu)*S
        dI <- Beta*I*S /(N*((1+(dD/delta_c))^k)) - (q + mu) *I
        dQ <- q *I - (gamma + delta + mu) * Q
        dR <- gamma*Q + nu*v*S - (reinf+mu) * R
        return(list(c(dS, dI, dQ, dR, dD)))
      })
    }
    }
  }
time <- seq(from = 0, to = length, by = 1)
```

```
if (use_age_structure){
   res <- as.data.frame(ode(y = age_init,
                        times = time,
                         func = theModel,
                          parms = params))


}
else{
  res <- as.data.frame(ode(y = init,
                        times = time,
                         func = theModel,
                          parms = params))
  colnames(res) <- c("time", stateNames)
}
return(res)
}

############PLOTTING STARTS HERE
plot_sim <- function(use_age_structure =TRUE,
                     length = 150,
                     use_awareness = FALSE,
                     vaccine_strategy = NULL,
                    Beta1 = 0.0251,
                    Beta2 = 0.0355,
                    Beta3 = 0.122,
                    Beta4 = 0.1698,
                    Beta5 = 0.4488,
                    betavals = c(Beta1, Beta2, Beta3, Beta4, Beta5),
                     params = make_params(),
                     sim = run_sim(use_age_structure = use_age_structure,
                                   length = length,
                                   betavals = betavals,
                                   params = params,
                                   use_awareness = use_awareness,
                                   vaccine_strategy = vaccine_strategy),
                     drop = c("S", "R")){
  library("dplyr")
  library("ggpubr")
  library("tidyr")
  library("directlabels")
  library("lubridate")
  ##Do this first.
  sim <- as_tibble(sim)
  if (!use_age_structure){
    sim$"Immune" <- sim$R
    sim$"Cumulative Immune" <- cumsum(sim$R)
    sim$"Death" <- sim$D
    sim$"Cumulative Deaths" <- cumsum(sim$D)
    ##Drop original columns
    sim <- sim[,!colnames(sim) %in% c("S", "D", "R", "Cumulative Recovered", "Cumulative Deaths")]
  }
  ##This will only work if our data is still in the long form.
```

```
  else{
    ##Cite Dr. Bolker's Milestone 1 Feedback.
    throwout <-  grepl(colnames(sim), pattern = paste(drop,collapse="|"))
    sim <- sim[,!throwout]
  }
  ncurves <- length(sim)
  ##If we haven't done this already
  sim <- tidyr::pivot_longer(sim, col = !time)
  sim$"Compartment Type" <- sim$name
  ##Rename columns.
  p <- ggline(data = sim, x= "time", y = "value", color = "Compartment Type", size = 1.5, plot_type = "
    labs(title = "Pandemic simulation",
         x = "Time (days)",
         y = "Count")
  p <- p + scale_x_discrete(breaks = seq(from = 0,
                                         to = length, by = 20),
                            labels = month.name[month(lubridate::today() + days(seq(from = 0, to = leng
    p
}
compare_vaccine_plot <- function(strat = NULL,
                                 length = 150,
                                 use_age_structure = FALSE,
                                 strat1 = NULL,
                                 strat2 = NULL,
                                 drop = c("S", "R", "D","Q")){
  library("tidyr")
  ##Allow for customs strategies to be coded in for simplicity
  if(!is.null(strat)){
      if (strat == "t_null"){
    strat1 <- function(t){return(0)}
    strat2 <- function(t){return(t)}
      }
    else{
    }
  }
else{
}
      sim1 <- run_sim(use_age_structure = use_age_structure,
                                   length = length,
                                   vaccine_strategy = strat1)
      sim2 <- run_sim(use_age_structure = use_age_structure,
                                   length = length,
                                   vaccine_strategy = strat2)
      if (use_age_structure){
      ##If we're using age structure, do this first before we convert to long form.
      sim1 <- sim1[,!grepl(colnames(sim1), pattern = paste(drop,collapse="|"))]
      sim2 <- sim2[,!grepl(colnames(sim2), pattern = paste(drop,collapse="|"))]
      }
      else{
        sim1 <- sim1[,!colnames(sim1) %in% drop]
        sim2 <- sim2[,!colnames(sim2) %in% drop]
      }
      ##Add the vaccination strategy to the labels of whatever is left.
```

```
colnames(sim1) <- c("time", paste0(colnames(sim1)[2:length(colnames(sim1))], ": Vaccination Strategy 1")
colnames(sim2) <- c("time", paste0(colnames(sim2)[2:length(colnames(sim2))], ": Vaccination Strategy 2")
    sim1 <- tidyr::pivot_longer(sim1, col = !time)
    sim2 <- tidyr::pivot_longer(sim2, col = !time)
    sim <- dplyr::bind_rows(sim1, sim2)
    sim$"Compartment" <- sim$"name"
    p <- ggline(data = sim, x= "time", y = "value", color = "Compartment", size = 1.5, plot_type = "l")
    labs(title = "Pandemic simulation",
         x = "Time (days)",
         y = "Count")
  p <- p + scale_x_discrete(breaks = seq(from = 0,
                                         to = length, by = 20),
                            labels = month.name[month(lubridate::today() + days(seq(from = 0, to = leng
    p

}
```

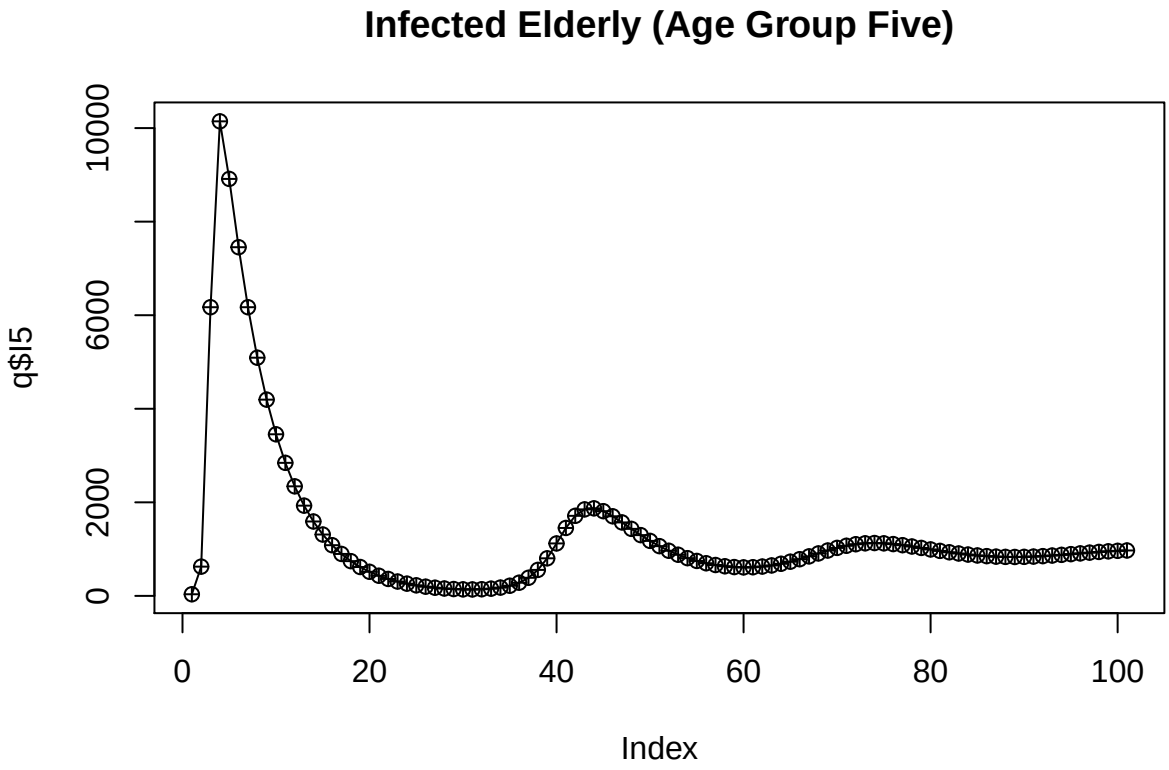
```
plot_sim(use_awareness = TRUE )
```




```
q = run_sim(use_awareness = TRUE)
```

Isolating one term, note the dampened oscillation; this was not present in the initial model, and it better matches the periodic behaviour of COVID:

```
plot(q$I5, pch = 10, type = "o")
title( "Infected Elderly (Age Group Five)")
```



The next few code chunks provide a nice framework for plotting the CN covid data by town, with plenty of options. This is meant for checking spacial heterogeniety — as mentioned in meetings, we looked a lot at how we could get better and more accurate model fit, using results from the King et al paper. This paper notes that in pandemics with high spacial heterogeniety, models fail to predict pandemic outcomes due to many small local flairups vitiating the usual logic around herd immunity in an SIR model, which requires a high level of mixing to be accurate. (Note, some of these code chunks moved to the start to fascilitate the flow of this overview, specifically the python plotting).

```
#This chunk handles python imports
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
ByTown = pd.read_csv("ConCovidTowns.csv")
ccv = pd.read_csv("ConCOVID.csv")
AgeDeathDat = pd.read_csv("ageDeaths.csv")
```

This is the main code for plotting.

```
#read in the covid data organized by town.
#These are graph settings for Seaborn. Can ignore if trying to
#understand code --- they mask
#matplotlib features I'm pretty sure.
rc={'lines.linewidth': 4, 'axes.labelsize': 20, 'axes.titlesize': 18}
sns.set(rc=rc)
sns.set_style("whitegrid", {'axes.grid' : False}) #remove grey grid
sns.set_context("paper") #changes the theme
plt.rcParams['figure.figsize'] = [16, 7] #changes the size
#Function for separating out towns.
def getTown(df: pd.DataFrame, n: int) -> pd.DataFrame:
```

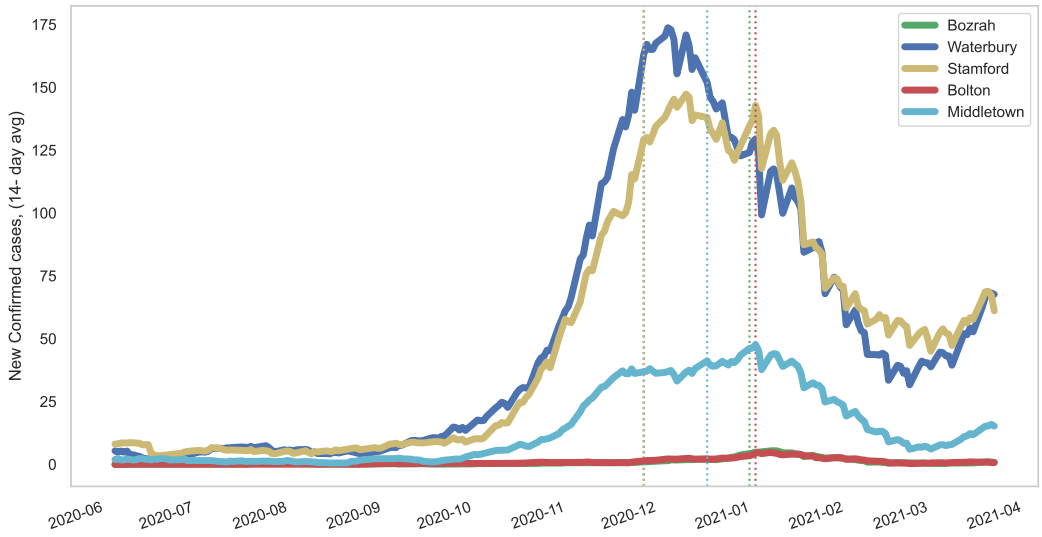
```
    '''
    Give the dataframe and a town number.
    Returns all columns for that town.
    '''
    #Rename Date Column
    df.rename(columns = {'Last update date':'Date'}, inplace = True)
    #Make date a proper datetime object
    df['Date'] = pd.to_datetime(df.Date)
    #Sort by Date
    df.sort_values(by='Date')
    #Reset Index
    #Grab all elements of big array with this town number
    j = df.loc[df["Town number"] == n]
    #Reset index now that other towns are gone
    j.reset_index(drop=True, inplace=True)
    return(j)

def cleandatAge(df: pd.DataFrame,
                    diff = False,
                    loess = False) -> pd.DataFrame:
    ''' Take in a dataframe that sucks and output a
        dataframe that has the right dates,
        is in the right order,
        is disagregated if diff = True,
        and is smoothed if loess = True.

        Also fix some column names.
    '''
    df['Date'] = pd.to_datetime(df.Date)
    df = df.sort_values(by='Date')
    j = df.reset_index(drop=True)
    return(j)


def getPeak(df:pd.DataFrame, n:int, metric:str) -> pd.DataFrame:
    '''
    given original dataframe df, town number n, and metric str,
    return the day at which the most NEW (metric) were recorded.
    For plotting lines on graphs :D
    '''
    maximum = getTown(ByTown, n)[metric].diff().max()
    q = getTown(ByTown, n).loc[getTown(ByTown, n)[metric].diff() == maximum]
    q.reset_index(drop=True, inplace=True)

    #Return the first time the max comes up --- one or two towns reported the
    #same max cases twice, always within a week. So this removes the ambiguity.

    return(q.head(1))


def getFullPeakDF(df:pd.DataFrame, metric:str) -> pd.DataFrame:
    '''
    Given original dataframe df, and a metric (eg Confirmed cases), return a
```

```
    dataframe containing the row in which the most new people were registered
    as having [metric] eg, testing postitive for Confirmed cases.
    '''
    MaxDF = getPeak(ByTown, 1, metric)
    for i in range(2,ByTown["Town number"].max()):
        MaxDF = pd.concat([MaxDF, getPeak(ByTown, i, metric)])

    MaxDF['Date'] = pd.to_datetime(MaxDF.Date)
    MaxDF = MaxDF.sort_values(by='Date')
    MaxDF.reset_index(drop=True, inplace=True)

    return(MaxDF)
def plotTownCases(df: pd.DataFrame,
                  n:int,
                  metric: str,
                  col:str,
                  bins:int = 14,
                  maxcaseline = False,
                  IncludeLegend = True):
    '''
    Give df(big dataframe) and town number (n) and return a plot
    of rolling average over (bins) days for daily change in column (metric)
    '''

    UnC = getTown(ByTown, n)[metric].diff()

    plt.ylabel("New {}, ({}- day avg)".format(metric, bins))
    plt.xticks(rotation=19)
    if IncludeLegend == True:
        plt.plot(getTown(ByTown, n)["Date"],
         UnC.rolling(bins).mean(),
         label = getTown(ByTown, n)["Town"][0],
         color = col,linewidth= 4)
        plt.legend()
    else:
        plt.plot(getTown(ByTown, n)["Date"],
         UnC.rolling(bins).mean(),
         color = col)


    if maxcaseline == True:
        plt.axvline(x=getPeak(df,n,metric)._get_value(0,"Date"),
                    ls =':',
                    color= col)

def plotAll(df:pd.DataFrame,
            n:int,
            metric:str,
            col:str = 'b',
            bins:int = 14,
            mcl = True):
    '''
    Plot all the towns and highlight town (n) in (col) with rolling avg over
```

```
    9bins) and maybe a line where there are the most new cases (not avgd)
    '''
    for i in range(1, ByTown["Town number"].max()):
        if i == n:
            plotTownCases(ByTown, i, metric, col, maxcaseline = mcl,
                          IncludeLegend = True)

        else:
            plotTownCases(ByTown,
                          i,
                          metric,
                          'grey',
                          maxcaseline = False,
                          IncludeLegend = False)

#makes raw data into nice data for optimizer :)
def npTownForLM(df:pd.DataFrame, n:int, metric: str) -> np.array:
    #Want np.ndarray with
    q = getTown(df, n)[metric].tail(len(getTown(df, n)[metric])-70)
    q = q.to_numpy()
    return(np.nan_to_num(q))

def getAgeGroupDF(df:pd.DataFrame, group: str, bins:int = 14) -> pd.DataFrame:
    '''
    Make the death data nice
    '''

    df.rename(columns = {'DateUpdated' : 'Date'}, inplace = True)
    df['Date'] = pd.to_datetime(df.Date)
    df = df.sort_values(by = "Date")
    j = df.loc[df['AgeGroups'] == group]
    j.reset_index(drop = True, inplace  = True)

    dict0 = {"Date": j["Date"][1:], "Deaths": np.nan_to_num(np.diff(j["Total deaths"]))}
    _ =  pd.DataFrame(dict0)
    dict1 = {"Date": _["Date"][bins:], "Deaths": _["Deaths"].rolling(14).mean()[bins:]}

    return(pd.DataFrame(dict1))
```

Next, we produce plots to check spacial heterogeneity for some of the larger towns:

```
#Plots
plt.figure(figsize=(10,5))

plotTownCases(ByTown, 13, "Confirmed cases", 'g', maxcaseline = True)
plotTownCases(ByTown, 151, "Confirmed cases",'b', maxcaseline = True)
plotTownCases(ByTown, 135, "Confirmed cases", 'y', maxcaseline = True)
plotTownCases(ByTown, 12, "Confirmed cases", 'r', maxcaseline = True)
#plotAll(ByTown, 15, "Confirmed cases")
plotTownCases(ByTown, 83, "Confirmed cases", 'c', maxcaseline = True)
plt.show()
```



Every large town has a pretty similar outbreak pattern — this is a great sign for our model, which assumes dynamics like this. We didn't end up using this space-structured data in fitting, but it's good to see that this model assumption is validated.

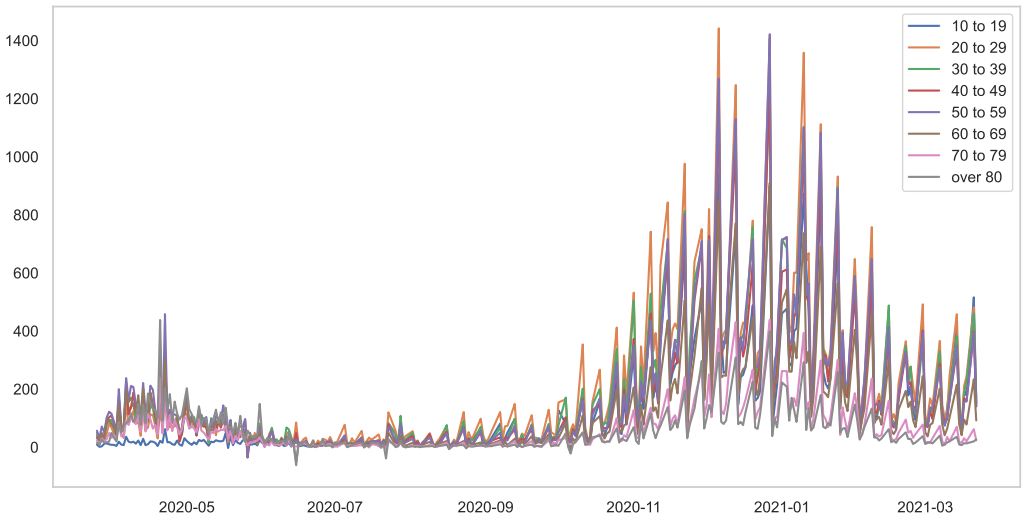
Our model is age-structured, so we need to import the CN data broken down by age group. We do this and plot below:

```
#dataframe stuff:
agedf = pd.read_csv("agedat.csv")
ageDeaths = pd.read_csv("ageDeaths.csv")
j = cleandatAge(agedf, diff = False)

dict0 = {"Date": j["Date"][1:], "Cases":  np.diff(j["cases_age0_9"])}
df0 = pd.DataFrame(dict0)

dict1 = {"Date": j["Date"][1:], "Cases":  np.diff(j["cases_age10_19"])}
df1 = pd.DataFrame(dict1)

dict2 = {"Date": j["Date"][1:], "Cases":  np.diff(j["cases_age20_29"])}
df2 = pd.DataFrame(dict2)

dict3 = {"Date": j["Date"][1:], "Cases":  np.diff(j["cases_age30_39"])}
df3 = pd.DataFrame(dict3)

dict4 = {"Date": j["Date"][1:], "Cases":  np.diff(j["cases_age40_49"])}
df4 = pd.DataFrame(dict4)

dict5 = {"Date": j["Date"][1:], "Cases":  np.diff(j["cases_age50_59"])}
df5 = pd.DataFrame(dict5)

dict6 = {"Date": j["Date"][1:], "Cases":  np.diff(j["cases_age60_69"])}
df6 = pd.DataFrame(dict6)
```

```
dict7 = {"Date": j["Date"][1:], "Cases":  np.diff(j["cases_age70_79"])}
df7 = pd.DataFrame(dict7)

dict8 = {"Date": j["Date"][1:], "Cases":  np.diff(j["cases_age80_Older"])}
df8 = pd.DataFrame(dict8)

plt.plot(df1["Date"],df1["Cases"], label ="10 to 19")
plt.plot(df2["Date"],df2["Cases"], label ="20 to 29")
plt.plot(df3["Date"],df3["Cases"], label ="30 to 39" )
plt.plot(df4["Date"],df4["Cases"], label = "40 to 49")
plt.plot(df5["Date"],df5["Cases"], label ="50 to 59")
plt.plot(df6["Date"],df6["Cases"], label ="60 to 69")
plt.plot(df7["Date"],df7["Cases"], label ="70 to 79")
plt.plot(df8["Date"],df8["Cases"], label = "over 80")
plt.plot()
```

```
## []
```

```
plt.legend()

#agedf.columns
```



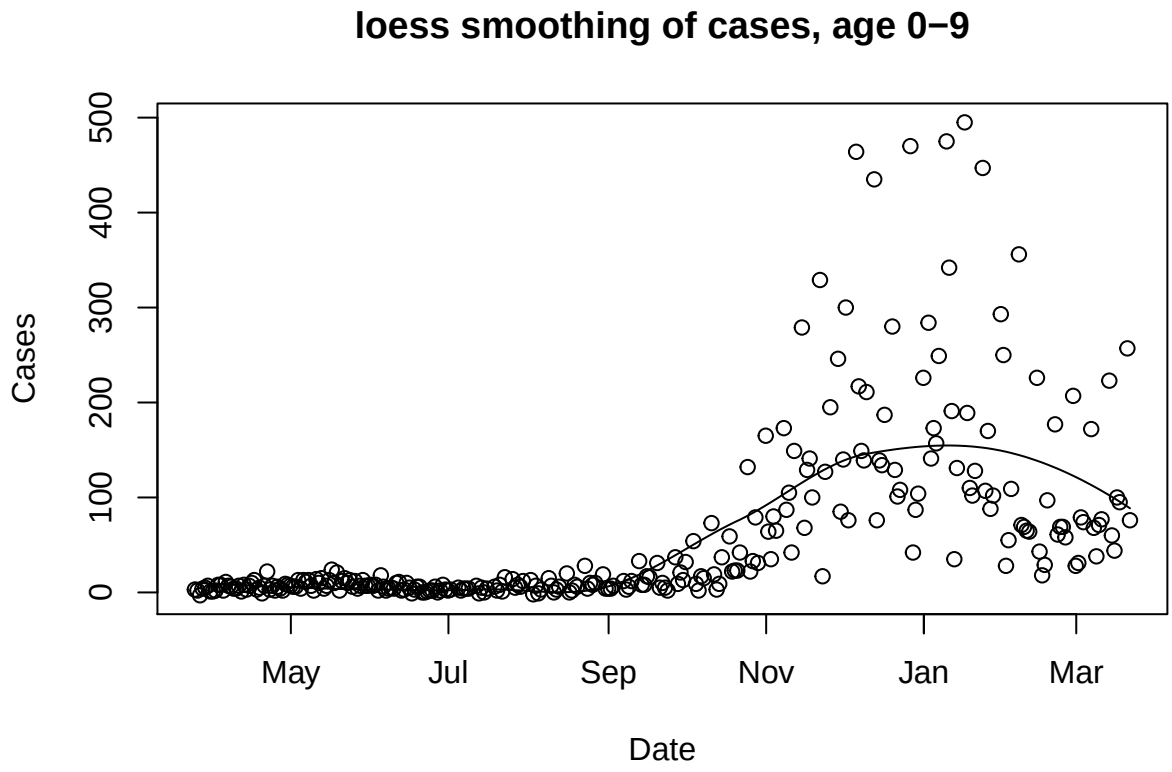
Note how the data is wildly noisy — this can make it hard to get good model fit. Artificially low new case totals in a day pull can impact model fit by pulling the curve down below the true value. See the first graph in the milestone for a look at this in action.

Because of this, we implemented two methods for data smoothing as a form of preprocessing.

First, LOESS, as recommended on Zulip.

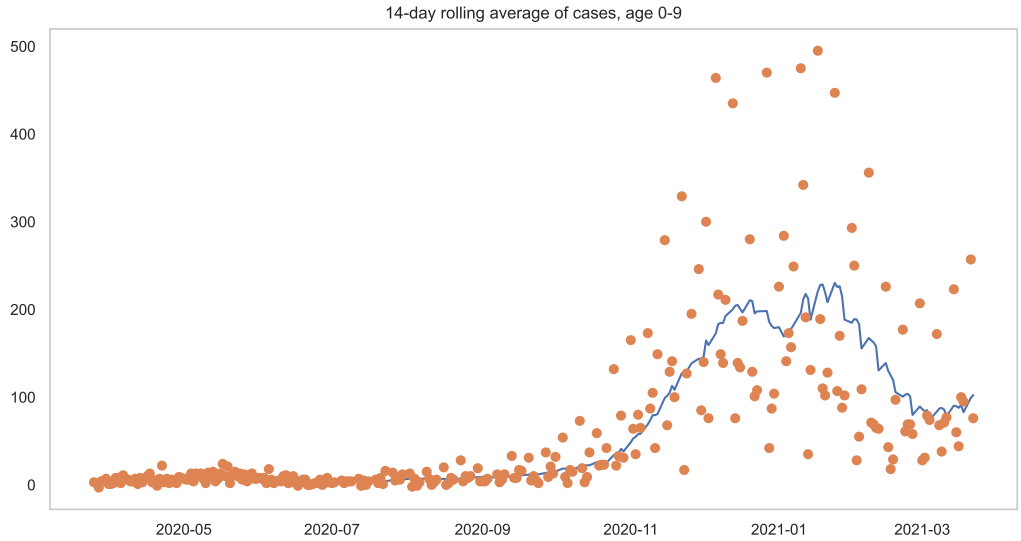
```
library(reticulate)
#LOESS SMOOTHING
#Actually really not great. Will compare to rolling average:
```

```r
#based on code by Ben Bolker on Zulip
dd = py$df0
x = dd$Date
y = dd$Cases
plot(Cases~Date, data=dd)

m = loess(Cases~as.numeric(Date),data=dd)
lines(dd$Date,predict(m))
title("loess smoothing of cases, age 0-9")
```



Compared to rolling avg with two-week bin:

```r
#Seaborn titles are very weird in reticulate Rmd. Maybe plot seperatley and
#include as images for final report.
plt.figure(figsize=(10,5))
plt.plot(df0["Date"], df0["Cases"].rolling(14).mean())
plt.plot(df0["Date"], df0["Cases"], marker = "o", linestyle="None")
plt.title("14-day rolling average of cases, age 0-9")
plt.show()
```

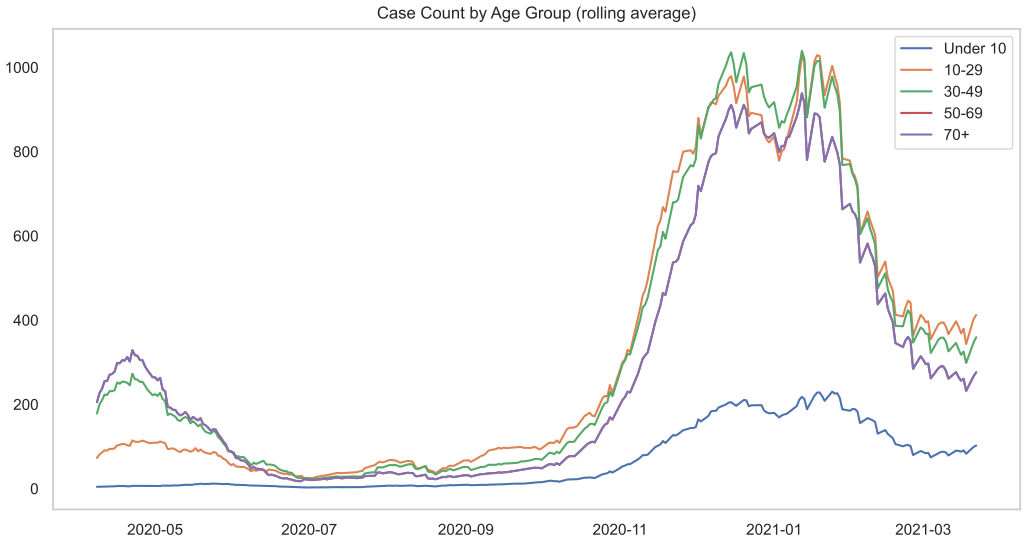


It seems like the rolling average might capture more of the model structure compared to the LOESS method. Moving forwards here, we'll use moving averages, though this is subject to change for the final project.

Plotting all the rolling averages for every age group:

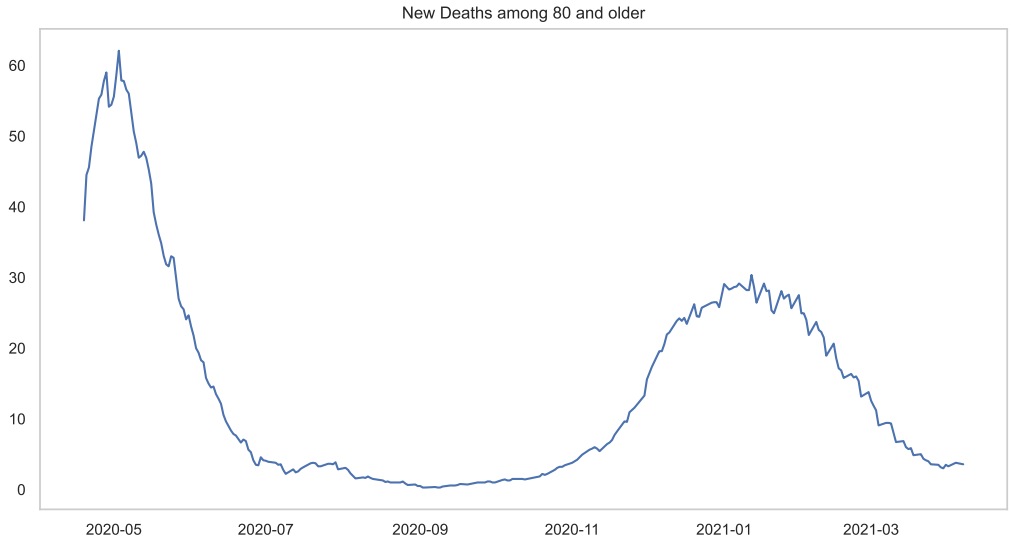
```
###Make  dataframe for 14-day rolling average dataset of cases
##Have five age groups so cluster:
# 0-9  kids stay together
# 10-19 and 20-29 cluster
# 30-39 and 40-49 cluster
# 50-59 and 60-69 cluster
# 70-79 and 80+ cluster
bins = 14

dict = {"Date": df0["Date"][bins:], "Cases":  df0["Cases"].rolling(14).mean()[bins:]}
df0_9 = pd.DataFrame(dict)

dict = {"Date": df1["Date"][bins:], "Cases":  (df1["Cases"] + df2["Cases"]).rolling(14).mean()[bins:]}
df10_29 = pd.DataFrame(dict)

dict = {"Date": df3["Date"][bins:], "Cases":  (df3["Cases"] + df4["Cases"]).rolling(14).mean()[bins:]}
df30_49  = pd.DataFrame(dict)

dict = {"Date": df1["Date"][bins:], "Cases":  (df5["Cases"] + df6["Cases"]).rolling(14).mean()[bins:]}
df50_69  = pd.DataFrame(dict)

dict = {"Date": df1["Date"][bins:], "Cases":  (df7["Cases"] + df8["Cases"]).rolling(14).mean()[bins:]}
df70_89  = pd.DataFrame(dict)

plt.figure(figsize=(10,5))
plt.plot(df0_9["Date"], df0_9["Cases"], label = "Under 10")
plt.plot(df10_29["Date"], df10_29["Cases"], label = "10-29")
plt.plot(df30_49["Date"], df30_49["Cases"], label = "30-49")
```

```
plt.plot(df50_69["Date"], df50_69["Cases"], label = "50-69")
plt.plot(df70_89["Date"], df50_69["Cases"], label = "70+")
plt.legend()
plt.title("Case Count by Age Group (rolling average)")
plt.show()
```



Fitting to new cases introduced a lot of issues, given that our model has no direct analogue. Multiple composite measures such as $Infected + Quarantined + Dead$ were tried, but in the end it seemed easier to fit to deaths. This is available from a different dataset online, which is pulled processed,and plotted below.:

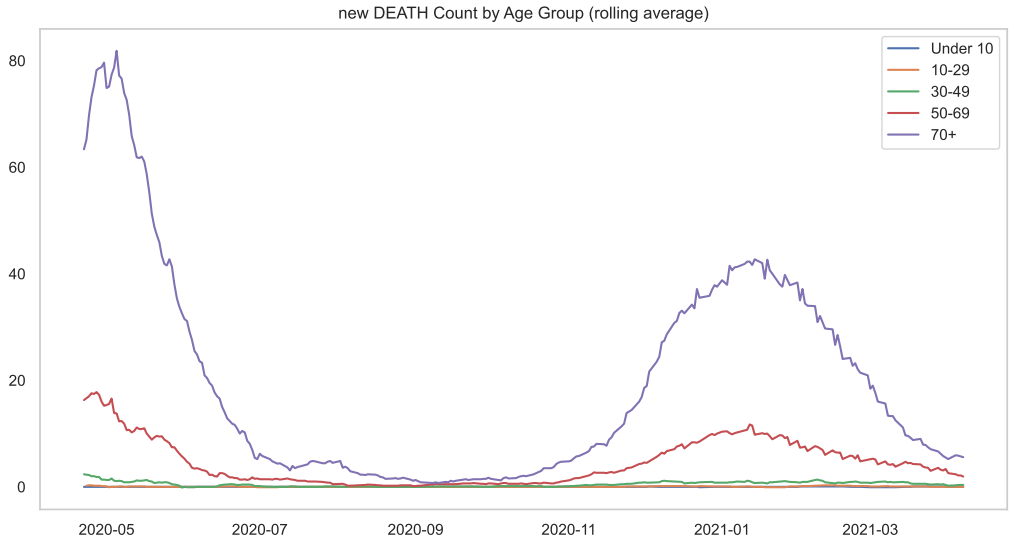
```
#Test code to get data from the age-structured deaths dataset. (auto deagregated)
q = getAgeGroupDF(AgeDeathDat, group = "80 and older", bins = 1)
plt.figure(figsize=(10,5))
plt.title("New Deaths among 80 and older")
plt.plot(q["Date"],q["Deaths"])
```



We separate out every age category and recombine into five new composite categories to represent the five age groups we have in our model. These new death counts are plotted below:

```
#DEATHS
binsNum = 14

#make dataframes:
df0 = getAgeGroupDF(AgeDeathDat, group = "0-9", bins = binsNum)
df1 = getAgeGroupDF(AgeDeathDat, group = "10-19", bins =binsNum)
df2 = getAgeGroupDF(AgeDeathDat, group = "20-29", bins =binsNum)
df3 = getAgeGroupDF(AgeDeathDat, group = "30-39", bins =binsNum)
df4 = getAgeGroupDF(AgeDeathDat, group = "40-49", bins =binsNum)
df5 = getAgeGroupDF(AgeDeathDat, group = "50-59", bins =binsNum)
df6 = getAgeGroupDF(AgeDeathDat, group = "60-69", bins =binsNum)
df7 = getAgeGroupDF(AgeDeathDat, group = "70-79", bins =binsNum )
df8 = getAgeGroupDF(AgeDeathDat, group = "80 and older", bins = binsNum)


##Case dictionaries
dict = {"Date": df0["Date"], "Deaths":  df0["Deaths"].fillna(0)}
Deaths0_9 = pd.DataFrame(dict)

dict = {"Date": df1["Date"], "Deaths":  (df1["Deaths"] + df2["Deaths"]).fillna(0)}
Deaths10_29 = pd.DataFrame(dict)

dict = {"Date": df3["Date"], "Deaths":  (df3["Deaths"] + df4["Deaths"]).fillna(0)}
Deaths30_49  = pd.DataFrame(dict)

dict = {"Date": df5["Date"], "Deaths":  (df5["Deaths"] + df6["Deaths"]).fillna(0)}
Deaths50_69  = pd.DataFrame(dict)

dict = {"Date": df7["Date"], "Deaths":  (df7["Deaths"] + df8["Deaths"])[:-2].fillna(0)}
Deaths70_89  = pd.DataFrame(dict)
```

```
plt.figure(figsize=(10,5))
plt.plot(Deaths0_9["Date"], Deaths0_9["Deaths"], label = "Under 10")
plt.plot(Deaths10_29["Date"], Deaths10_29["Deaths"], label = "10-29")
plt.plot(Deaths30_49["Date"], Deaths30_49["Deaths"], label = "30-49")
plt.plot(Deaths50_69["Date"], Deaths50_69["Deaths"], label = "50-69")
plt.plot(Deaths70_89["Date"], Deaths70_89["Deaths"], label = "70+")
plt.legend()
plt.title("new DEATH Count by Age Group (rolling average)")


#Deaths0_9.tail()
#Deaths70_89.tail()
plt.show()
```



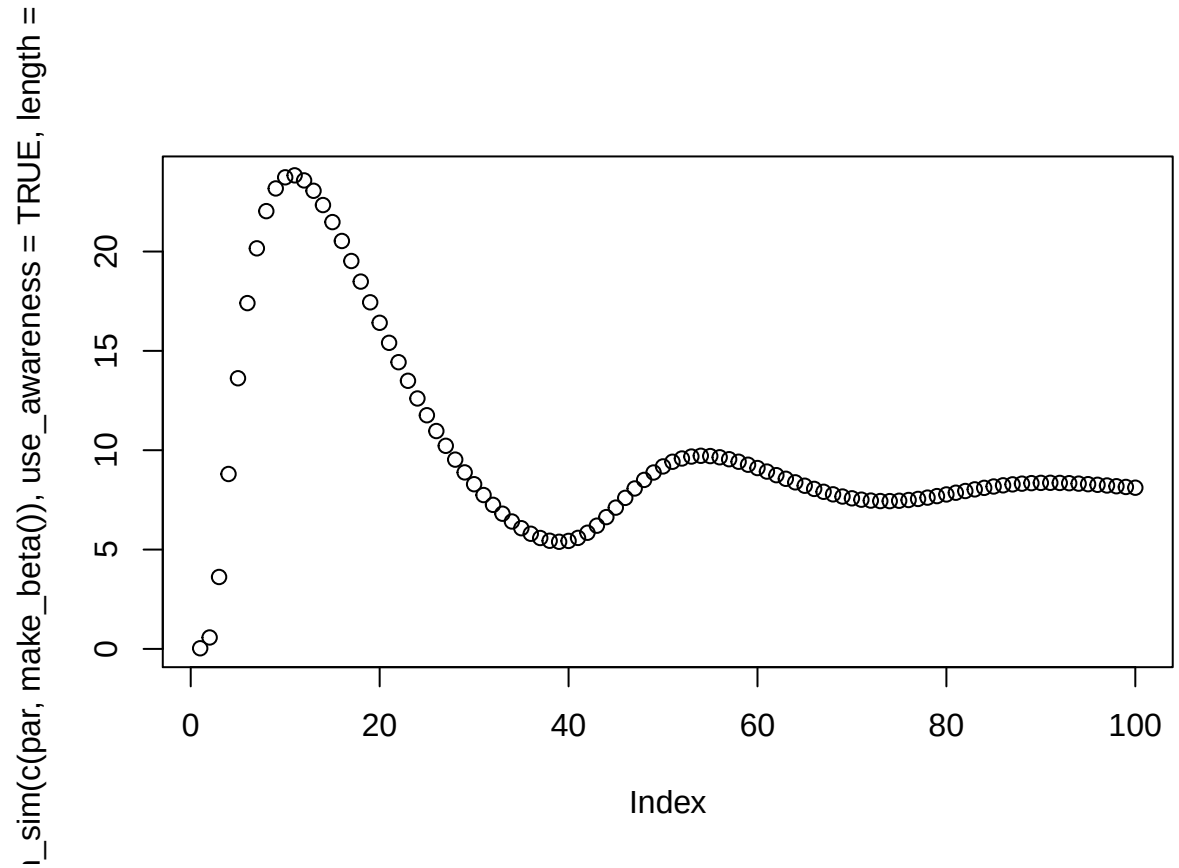
Total length of time series:

```
##Testing
library(reticulate)
length(py$Deaths70_89$Deaths)
```

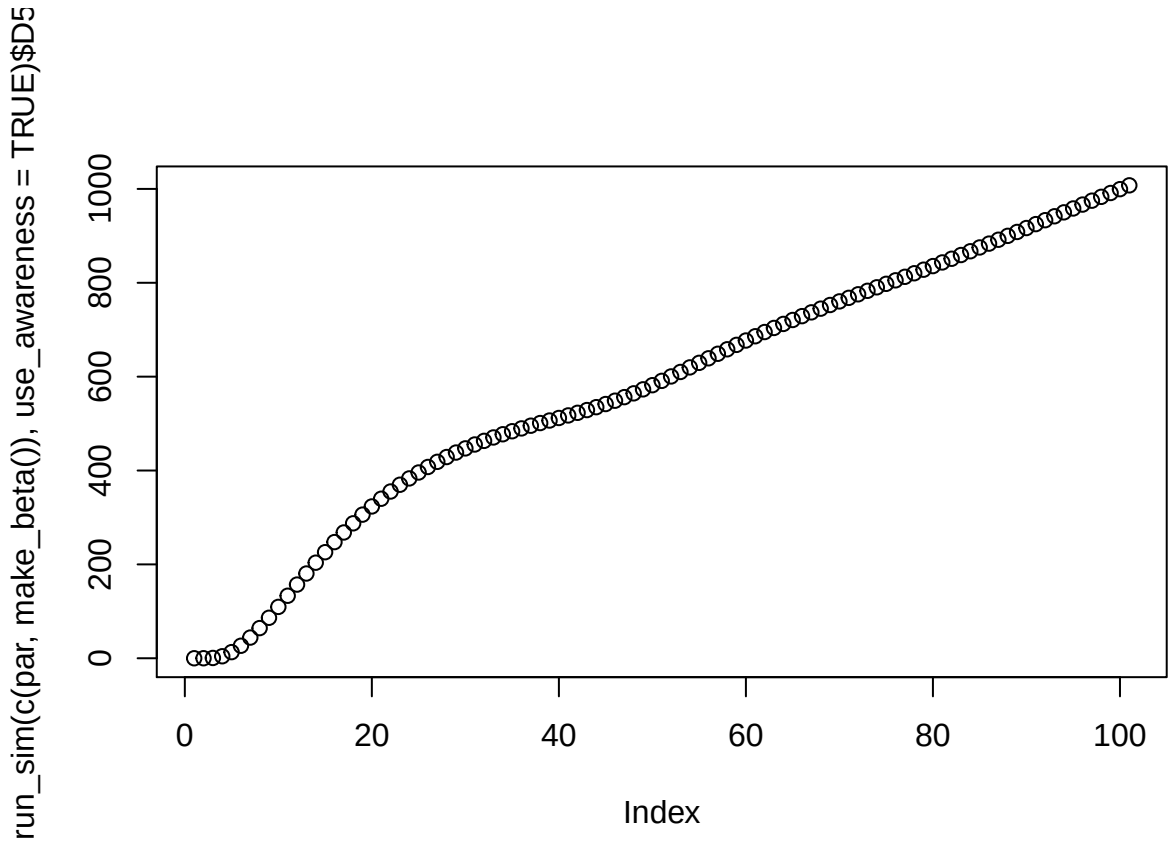
```
## [1] 268
```

Before fitting, we check to see if our model has similar structure among the elderly:

```
#What do the deaths look like for old people in our model before fitting?
#Has awareness and age structure both on.

par = make_params()
par["k"] = 2
par["delta_c"] = 1/(theN**2)

plot(diff(run_sim(c(par, make_beta()),use_awareness = TRUE, length= 100)$D5))
```




```
#large part of this is vital dynamics maybe? Turning those off.
par["natural_death_rate"] = 0
#Doesn't change anything.
#cumulative
plot(run_sim(c(par, make_beta()),use_awareness = TRUE)$D5)
```



Our data levels off sooner and sharper going into the summer. This seems fixable with fitted parameters. There is one place our model does not fit: if you look back at new cases, the largest spike by far is in the second wave; this goes against our implicit assumption of dampend oscillation.

A proposed modification could be to have some sort of marginal death rate, such that as more people die, the rate δ of deaths falls, indicating that those who survive the first wave have increased ability to survive the following waves, or that as the pandemic spreads, technology and care increases such that survival becomes a more likely case outcome. This plays well with our awareness term; as deaths per infection falls, the rate of infections would increase, leading to increasing oscillation before they eventually die out. This matches the data, and seems like a reasonable modification, so we'll look into if this is a reasonable addition next, although it might not make it to the final project.

Next we try fitting to data. After many hours, this didn't end up working like we'd hoped. Our initial model fitter for simple models in python works great now after failing the first milestone, and we are confident we'll have better results here for the final project.

Tests of error functions are below.

```
###Make  error function
library(MLmetrics) #Gives nice MLE
```

```
##
## Attaching package: 'MLmetrics'
```

```
## The following object is masked from 'package:base':
##
```

```
##     Recall
errorDeaths =  function(parameters){

  #run a simulation for input parameters with output the same length as the dataset
  #subtract one because run_sim is too long by one by default. All dfs same length.
  sim_results = run_sim(params = parameters, length = (length(py$Deaths10_29$Deaths) - 1 ),use_awareness

  s1 = (sim_results$D1 )
  s2 = (sim_results$D2 )
  s3 = (sim_results$D3 )
  s4 = (sim_results$D4 )
  s5 = (sim_results$D5 )


  return(sqrt( MSE(s1, py$Deaths0_9$Deaths) +
           MSE(s2, py$Deaths10_29$Deaths) +
           MSE(s3, py$Deaths30_49$Deaths) +
           MSE(s4, py$Deaths50_69$Deaths) +
           MSE(s5, py$Deaths70_89$Deaths)) )
}


errorCases = function(parameters){
  #Back when I was trying to fit to cases I was using this, but
  #fitting to cases is real tough --- I couldn't get a parameter (or combination of parameters) that
  #seemed reasonable to fit diff(Total cases) to. Definitely possible and probably not hard,
  #but vital dynamics making negative diff() in complement of susceptable or similar + people dropping
  #out of infected, cumsum($In) not looking great, and few other things lead me to try with deaths
  sim_results = run_sim(params = parameters, length = (length(py$df10_29$Cases) -1),use_awareness = TRU

  s1 = (sim_results$I1 + sim_results$Q1 )
  s2 = (sim_results$I2 + sim_results$Q2 )
  s3 = (sim_results$I3 + sim_results$Q3 )
  s4 = (sim_results$I4 + sim_results$Q4 )
  s5 = (sim_results$I5 + sim_results$Q5 )

  return(  MSE(s1, py$df0_9$Cases) +
           MSE(s2, py$df10_29$Cases) +
           MSE(s3, py$df30_49$Cases) +
           MSE(s4, py$df50_69$Cases) +
           MSE(s5, py$df70_89$Cases) )
}


###Test these out

#Make parameter list
Para1 = make_params()

#Change value for beta
Para1['Beta'] = 0.42

#Change value for gamma
```

```
Para1['gamma'] = 0.0082

#Print list with new values as check
#print(Para1)

#Calculate error functions at these new values
errorCases(Para1)
```

```
## [1] 42510817
```

```
errorDeaths(Para1)
```

```
## [1] 2737.388
```

```
#Run a simulation with these parameters to check for wierd infinities like optim claims
sim_results = run_sim(params = Para1, length = (length(py$df10_29$Cases) -1))

#Just use default length --- maybe the issue is with length
#run_sim(params = Para1)
#nope

#check results --- these never change for me. big issue.
#print(sim_results)

#check that the issue isn't with the way I was summing.
#print((sim_results$I1 + sim_results$Q1 + py$df10_29$Cases))
```

Our fitted parameters are below. Note that they are just the initial parameters, as our method did not converge:

```
##Using Optim, fit the model
library(stats)

#make new param list with default params
initparams = make_params()

#change any value to make sure this works; optim should change anyways.
initparams['gamma'] = 0.2

#test that I don't get an infinite value error with these params without optim
errorDeaths(c(make_params(), make_beta()))
```

```
## [1] 1207.82
```

```
#run optim
results = optim(par =  c(make_params(), make_beta() ),fn = errorDeaths, method = "Nelder-Mead", control
#, upper = c(1,1,1,1,50/40000000,1,1,8,50,50,50,50,50,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1

#look at results
results
```

```
## $par
##          Beta         gamma         delta             N       delta_c
##  4.215000e-01  8.760000e-02  2.800000e-03  3.565000e+06  2.805049e-07
##             q            nu             k       deltas1       deltas2
##  3.565002e+05  8.500000e-01  3.000000e+00  1.500000e-01  1.600000e-01
##       deltas3       deltas4       deltas5         reinf            mu
```

```
##  6.270000e-01  1.200000e+00  3.100000e+00  5.000000e-02  0.000000e+00
## natural_death
##  0.000000e+00  1.255000e-01  1.775000e-01  6.100000e-01  8.490000e-01
##
##  2.244000e+00  1.255000e-01  1.775000e-01  6.100000e-01  8.490000e-01
##
##  2.244000e+00  1.255000e-01  1.775000e-01  6.100000e-01  8.490000e-01
##
##  2.244000e+00  1.255000e-01  1.775000e-01  6.100000e-01  8.490000e-01
##
##  2.244000e+00  1.255000e-01  1.775000e-01  6.100000e-01  8.490000e-01
##
##  2.244000e+00
##
## $value
## [1] 30.45395
##
## $counts
## function gradient
##       44       NA
##
## $convergence
## [1] 1
##
## $message
## NULL
```

Check to see what parameters changed:

```
#Check to see what parameters it changed
results$par == c(make_params(),make_beta())
```

```
##          Beta         gamma         delta             N       delta_c
##          TRUE          TRUE          TRUE          TRUE          TRUE
##             q            nu             k       deltas1       deltas2
##         FALSE          TRUE          TRUE          TRUE          TRUE
##       deltas3       deltas4       deltas5         reinf            mu
##          TRUE          TRUE          TRUE          TRUE          TRUE
## natural_death
##          TRUE          TRUE          TRUE          TRUE          TRUE
##
##          TRUE          TRUE          TRUE          TRUE          TRUE
##
##          TRUE          TRUE          TRUE          TRUE          TRUE
##
##          TRUE          TRUE          TRUE          TRUE          TRUE
##
##          TRUE          TRUE          TRUE          TRUE          TRUE
##
##          TRUE
```

```
#Look at parameters
#para = results$par
#results
```

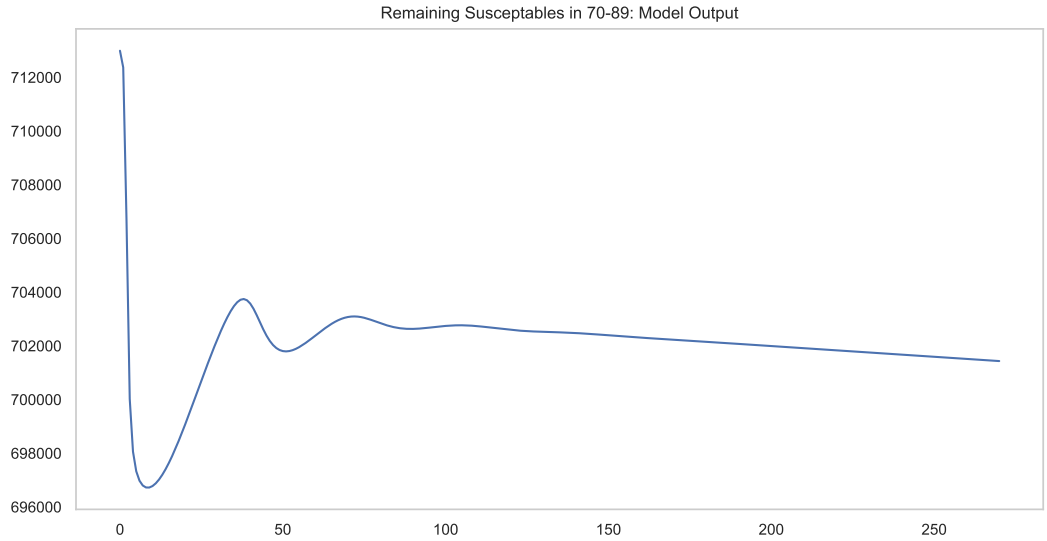
Nothing changed, as expected.

```
#Run simulation with new parameters
par['k'] = 2
sim_results = run_sim(params = par,  use_awareness = TRUE,length = (length(py$df10_29$Cases)))

#para == par
```

```
#Just a bunch of quick plots to look at results
#plot(NewCases2)
res = r.sim_results


plt.plot(np.diff(res["D1"]), label = "Deaths in age group 0-9")
plt.plot(np.diff(res["D2"]), label = "Deaths in age group 10-29")
plt.plot(np.diff(res["D3"]), label = "Deaths in age group 30-49")
plt.plot(np.diff(res["D4"]), label = "Deaths in age group 50-69")
plt.plot(np.diff(res["D5"]), label = "Deaths in age group 70-89")
plt.title("Deaths by age group: Model Output")
plt.legend()
plt.show()
```

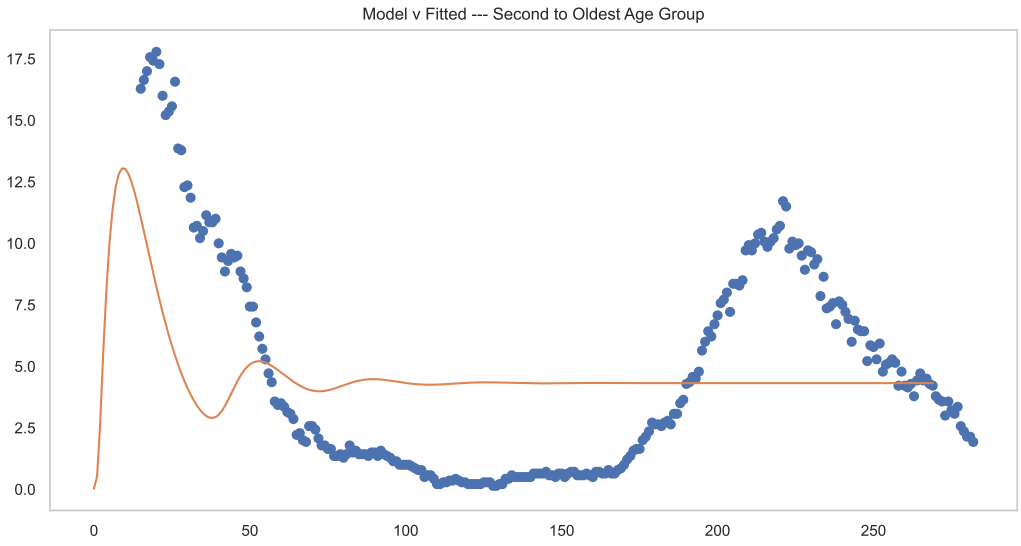
Now checking susceptables to see if persistant deaths are due to vital dynamics or disease dynamics

```
plt.title("Remaining Susceptables in 70-89: Model Output")
plt.plot((res["S5"]))
```



This suggests deaths are due to disease dynamics, not natural death rate — should look at changing δ, but no bugs here.

```
plt.title("Model v Fitted --- Second to Oldest Age Group")
plt.plot(Deaths50_69["Deaths"], marker = 'o', linestyle = 'none')
plt.plot(np.diff(res["D4"]))
#plot(sim_results$I1)
```



So, to conclude this section on parameter estimation, we have major issues with optimizers still. But our last optimizer works, and this one will too!

Our next steps include fixing these numerical issues, looking at different data smoothing methods, looking at adding marginal decreasing mortality, and fleshing out the mathematical analysis around R0 as outlined in our initial proposal.

Thank you!

```
#testing?
#run_sim(params = initparams)
#run_sim(make_params())
```