SARS-CoV-2 (Novel Coronavirus) C Notes

Geoffrey J. W. Noseworthy*, Sanjeev S. Seahra†

Department of Mathematics and Statistics, University of New Brunswick
Fredericton, NB, Canada, E3B 5A3

March 2, 2021

*gnosewo1@unb.ca
†sseahra@unb.ca

Abstract

These notes represent research in progress. Conclusions are tentative and are subject to change. They are not meant to be used by anyone for anything without prior approval.

Contents

1 The Software

1.1 Introduction

The purpose of this project is to develop a roughly Agent-Based Model that can simulate COVID-19 disease spread in a community, which, in this case, could represent any city in the provice of New Brunswick, Canada. Previously completed sections of the project worked to do this in the Python programming language, but due to complexity, speed, and inefficiency, all caused by the number of packages required for simply Python implementation, this section of the project will aim to begin anew in the C programming language (and later, the C++ programming language).

All used data is sourced within the code it is used, or when it is first presented in this document (such as population sizes). Numbers shall be obtained through Statistics Canada, Service New Brunswick, or any other provincial statistical branches.

This model, unlike the Python implementation, will aim to make use of no packages beyond the generally required ones for C: <stdio.h>, <stdlib.h>, <math.h>, <time.h>, <string.h>. These libraries are the most standard for C, and are common in many applications due to their implementation of reading and printing to files or the command line, easy access to commands, access to mathematicla functions, access to time functions, and access to more advanced memory manipulation.

1.2 The Code

The code is organized into 4 sections:

- Network Generation (NetworkGen.c)
- Dynamic Network Simulation (DNet.c)
- Driver Programs (NetworkTest.c, DNetTest.c)
- Control (makefile)

Each of these files will be discussed in turn, so that their purpose is understood.

1.2.1 Network Generation

The Network Generation code is, by far, the most complicated code of this document, and will be the most difficult to understand. A simple list of the functions located within this code can be found in NetworkGeneration.h.

To begin, this file is the one that defines what a person is within our simulation. It includes the necessary data for our purposes, such as their age, their number of connections, a simple counter for general use, a list of indexes to other people that they are connected to, and a list of what "types" of connections they have with those people. This is shown below:

```
typedef struct person {
  int age;
  int cCount;
  int status;
  int counter;
  int connections[MAX_C];
  int con_type[MAX_C];
} Person, *pPerson;
```

Following this, we have a number of programs designed to create a person or a network of people (or to copy them) and to put them into the heap memory for future access. These functions feature general memory management (Malloc and Calloc), as well as measures to ensure efficiency. For one looking to simply understand the network generation, these simply give us a framework of the network itself.

```
pPerson mallocPerson();
void copyPerson(pPerson dest,
        pPerson src);
pPerson* mallocNetwork(int size);
pPerson* createNetwork(int size);
void copyNetwork(pPerson* dest,
        pPerson* source, int size);
pPerson* createCopy(pPerson* src,
            int size);
```

Following this, we have the first of the important code sections. The first in this section in the fprintPerson command, which is a function to print out a person in an "index - age - connection number - connections with type" format. The network version simply does that for every single person. Both of these functions take file inputs, so, if you do not wish to print to a file and rather prefer the command line, then you simply must input "stdout" as the file name.

```
void printNetwork(pPerson* network,
          int size, String fileName);
void fprintPerson(pPerson person,
        FILE *file);
```

Now we have the important functions for the network generation. When creating a robust network, we obviously need to build a set of algorithmically strong connections, ones which

make sense in their generation. This is completed through the 5 "gen*" functions, where the star represents the rest of the name of the function. Each function, described using its own comment system, takes data from Statistics Canada and other organizations to create an appropriate set of connections for those in (or working in) Long Term Care-Homes, those in standard Households, those in Schools, those going to work, and then a set of friends.

For care homes, from this data, we see approximately 69 people per home, with 51 employees throughout the week. This allows us to build carehomes with a connection network of 120 connections, though the number of people within the population to bring about the need for a single care home is quite large, sitting around 11k.

For households, we see that the average household size is 2.3, and the age distribution is about 15% below 18, 65% between 18 and 65, and 20% aboce 65. We use these numbers (with a standard deviation of 0.5 in a normal random generator) to generate households for anyone not in a care home, and to assign each person an age. Currently, no protections exist to ensure that a household is not full of children.

For schools, we use an average class size of 22 people, with a standard deviation of 5. With this, we simply get that many children, and connect them.

For work, this is the exact same as school, except for adults and 20% of older individuals. Here we use a mean of 9.7, and a standard deviation of 2.

Finally, for friends, we simply give every person 2 connections with people not within any other group.

Each connection also has an assigned "type". As lockdowns begin within our simulation, certain connections are no longer viable. Thus, in phase 0, all connections are allowed. In phase 1, people lose their friend connections, In phase 2, all but household / care home connections are removed.

```
void genCareHomes(pPerson *base, int size,
        int* wNet, int* wP, int* hP);
void genHouseHolds(pPerson *base,int size,
        int* hNew, int* hp,int* childlist,
        int* cpos);
void genSchools(pPerson *base,
        int size, int* childlist);
void genWork(pPerson *base,
        int size, int* wNet, int* wpos);
void genFriends(pPerson *base, int size);
```

Next is the section for assistive functions. These swap input values, shuffle arrays, and give us random normal numbers.

```
double normalRandom();
void shuffle(int* array, int size);
void swap(int* a,int* b);
```

Finally, there are the "freedom functions". These take people or networks assigned to memory and free them, so that the memory can be used for something new. They are simple in code, but drastically important.

```
void freePerson(pPerson input);
void freeNetwork(pPerson* input,int size);
```

1.2.2 Dynamic Network Simulation

This next set of code concerns the actual simulation. The list of functions is comparitively small, but the depth of each individual function is more significant, and a number of comments are present to ensure full understanding at all times.

To repeat our previous format, we will go through each function one-by-one to explain what each does, but then will add a section in which we give a number of comments for how the code works. The final goal will be to create a simple interface to manage this for the user, but, until then, this must be done through comments and regular checking of formatting.

Unlike before, we will begin with the list of commands first.

```
void simulate(pPerson* orig, int size,
        int runs, String file, double data[]);

void day(pPerson* network,
        int size, double data[]);

void infect(pPerson person, double beta,
        double rate, int* final);

void test(pPerson person, double beta,
        int* final);

void fPrintNetworkStatus(pPerson* network,
        int size, FILE* fPtr);
```

The first of our functions is simulate. This command takes in a network, its size, a number of simulations, the output file, and a LOT of data. The order of this data is provided in the

final section.

The purpose of this function in particular is to open the desired file, organize the data appropriately, prepare for simulation, create the loading bar that took way too long to design, then pass this all to the "day" command in a loop for however many days are desired within our simulation. It then runs calculations to set an emergency phase, prints out the final results to a file, and prepares for the next simulation.

Following this, we have the day function. This function is effectively the simulation day-by-day. It takes the data, generates 2 random numbers for every single person to ensure that the stochastic nature of the disease is fully realizable, creates an array to hold the changes for each person, and then runs through the disease states for each person appropriately. This is the first time the disease itself shall be mentioned, so it deserves a moment of attention.

The COVID-19 part of this simulation is achieved using a significantlly modified SEIR model. Each person has a "state", starting in **susceptible** (having never encountered the disease). If one has an infected connection, their is a percent chance per day that they too become infected. If this happens, that person becomes **exposed**. An exposed person has a random chance per day to switch to the pre-symptomatic phase, in which a person in infectious, but cannot show symptoms, even if they are "fated" to become symptomatic. But, after 2 days are pased, this person may become symptomatic or asymptomatic. A **symptomatic** person is one who shows symptoms of COVID-19 and is the most infectious of any category. These indivisuals have a high percent chance of infecting their connections, and a random chance to recover or be tested each day. An **asymptomatic** person in the same, but shows no symptoms, doesn't get randomly tested, and has a lower (1/3) chance of infecting their connections.

This is the end of standard statuses, but there are a few others of importance. A person who is **tested** has been identified as one with COVID-19 by the local medical body. They are quarenteed and, after two days, their connections are tested as well to see if they have COVID-19. A **removed** individual has healed from the disease and is no longer active, while a **tested removed** invidual has also been removed, but their connections may be soon tested.

With this completed, we can now discuss infect and test. These commands are the ones where an infectious person has a random chance of infecting their neighbours (based on who they can contact in the current emergency phase, see the Network Generation section for that), or where a tested individual's neighbours are tested because they were.

Finally, there is fPrintNetworkStatus, which prints the number of people in each of the above categories, and then the number NOT susceptible at the end of the simulation. This is saved in a CSV friendly format for quick data analysis.

With this completed, we now have the comments. There are a number of things that one must know when using these files, and the easiest way to express this is through the comment code:

```
/*
  List of Disease Statuses:
  0 - Susceptible
  1 - Exposed
  2 - Pre-symptomatic
  3 - Symptomatic
  4 - Asymptomatic
  5 - Removed
  6 - Tested
  7 - Tested Removed
*/

/*
  Connection Types:
  0 - Yellow, Orange, or Red Phase
  1 - Yellow or Orange Phase
  2 - Only Yellow Phase
 */

/*
  State Types:
  0 - Yellow
  1 - Orange
  2 - Red
 */

/* DATA ORDER - HIGHLY IMPORTANT
   0 - Days
   1 - Beta (Infection Chance)
   2 - Alpha (Exposed to
            Pre-Symptomatic Chance)
   3 - Gamma (Recovery Chance)
   4 - Asymptomatic Chance
   5 - Asymptomatic Infection Rate
   6 - Symptomatic (Voluntary) Test
       Chance
   7 - Random Test Chance
   8 - Test Success Chance
   9 - Pre-symp Time (Days)
```

```
   10 - Border Crossings Per Day
   11 - Importation Infection Chance
   12 - Starting Phase
        (0 = yellow,
         1 = orange,
         2 = red)
   13 - Orange Case Threshold
   14 - Red Case Threshold
   15 - Travel Isolation Time
        (0 = full house,
         1 = just travel,
         2 = none)
   16 - Average Travel Time
*/
```

1.2.3 Drivers

1.2.4 Makefile

For our final file, we have the Make File. This is simply a file designed to make it easier to compile and run the code provided, without a need for always knowing the order of files and their dependecies.

The best way to explain what this file does is through the help command. Essentially, just type "make ____", where the black is filled with a listed command, to do whatever you need with the provided code. The commands are below:

```
-------------------HELP MENU-------------------
TERM - MEANING
Net - Just Network,
DNet - Simulation [Dynamic Network]

TASK - COMMAND
Compile and Run - NetTestFull and
                  DNetTestFull
Compile - NetTest and DNetTest
Run - NetTestRun and DNetTestRun
Remove Files - clean
```