FRED Tutorials

['Epistemix Inc.']

Aug 01, 2022

CONTENTS:

Welcome to the FRED tutorials documentation.

FRED is a programming language and modeling platform for simulating how a population changes over time. This document discusses the tutorials provided for the FRED Modeling Language™ to provide examples and guidance for FRED users.

CHAPTER

ONE

FRED TUTORIALS

The FRED-tutorials repository presents a collection of models that demonstrate essential features of the FRED Modeling Language™. The intended audience for these tutorials is new users of the FRED Modeling Platform™ looking for practical examples of how the FRED language features described in the user guide can be used to address real-world modeling problems. Each example builds on the previous examples by introducing new model components and making use of additional language features. Consequently new users will benefit from reading the tutorials in the following order:

1. *Simple Flu*
2. *Flu with Behavior*
3. *Flu with School Closure*
4. *Flu with Vaccine*

We also hope experienced users find the models described here, and the programming patterns employed in their implementation, a useful reference.

The models discussed in these tutorials aim to represent a flu epidemic spreading through a population, with each focusing on a different aspect of the epidemiological and behavioral phenomena surrounding the epidemic. The Simple Flu tutorial presents a foundational model of flu infection. The Flu with Behavior and Flu with School Closure tutorials introduce social distancing and the effects of policies intended to control the epidemic into the model. The Flu with Vaccine tutorial allows users to explore the use of a vaccine to influence the dynamics of the epidemic.

The tutorials occasionally refer to additional documentation in the FRED Guide. If you don't have access to the guide, and would like to, please contact support@epistemix.com.

CHAPTER

TWO

SIMPLE FLU

This example introduces the **Simple Flu** model, a standalone model for influenza and the basis for the `../flu-with-behavior`, `../school-closure`, and `../vaccine` tutorials.

2.1 Introduction

This model defines a condition, `INFLUENZA`, which agents can contract either via the meta agent (a source of initial infections) or interaction with other agents. Having contracted the condition, agents may develop symptoms or remain asymptomatic before recovering, at which point they are immune and can no longer contract `INFLUENZA`.

2.2 Review of code implementing the model

The code that implements the **Simple Flu** model is contained in two **`.fred`** files:

- **`main.fred`**
- **`simpleflu.fred`**

2.2.1 main.fred

The **`main.fred`** file organizes the model and specifies the location and time period to be simulated. The `simulation` block handles the latter part of this. Such a block is required in all FRED programs to define the simulated location and time period.

```
simulation {
    locations = Jefferson_County_PA
    start_date = 2020-Jan-01
    end_date = 2020-May-01
}
```

The only additional content in this file is the line `include simpleflu.fred`, which inserts the contents of **`simpleflu.fred`**. This has the effect of adding the `INFLUENZA` condition to our model. Multiple `include` statements can added when building a complex model.

2.2.2 simpleflu.fred

This file defines the `INFLUENZA` condition. `INFLUENZA` is a condition that can be passed by coming into contact with other agents. This type of transmission is specified by the statement `transmission_mode = proximity`. At the start of the run, all agents are assigned a susceptibility of 1.0 in the `agent_startup` block via the statement `INFLUENZA.sus = 1`. All agents begin in the `Susceptible` state (`start_state = Susceptible`). Agents wait in this state indefinitely. Exposure, either via the meta agent or another transmissible agent, moves an agent to the `Exposed` state (`exposed_state = Exposed`). It is not required that the state agents move to after exposure be called "Exposed" as it is in this case.

```
agent_startup {
    INFLUENZA.sus = 1
}

condition INFLUENZA {
    transmission_mode = proximity
    transmissibility = 1.0
    start_state = Susceptible
    exposed_state = Exposed
    meta_start_state = Import

    state Susceptible {
        INFLUENZA.sus = 1
        wait()
        next()
    }
```

Once in the `Exposed` state, an agent loses susceptibility to `INFLUENZA`. The agent then waits through an incubation period and either moves to a symptomatic and infectious state, `InfectiousSymptomatic`, with a probability of 0.33 or to an asymptomatic and infectious state, `InfectiousAsymptomatic`, with a probability of 0.67. Explicitly defining transition probabilities is one way of introducing stochastic behavior into the model. Another way is to draw values from a probability distribution, demonstrated by setting the incubation period for each agent as a sample from a `lognormal()` distribution

```
    state Exposed {
        INFLUENZA.sus = 0
        wait(24 * lognormal(1.9,1.23))
        next(InfectiousAsymptomatic) with prob(0.33)
        default(InfectiousSymptomatic)
    }
```

Both infectious states are identical with the exception of the effect on transmissibility. Once entering one of the infectious states after the wait period in `Exposed`, agents gain a non-zero transmissibility and wait through an infectious period before recovering. In this model, asymptomatic infections are set to be half as transmissible as infectious ones. In one of these states agents may transmit the condition to another susceptible agent.

```
    state InfectiousSymptomatic {
        INFLUENZA.trans = 1          # this value is .5 for InfectiousAsymptomatic
        wait(24* lognormal(5.0,1.5))
        next(Recovered)
    }
```

Once the infectious period is over, agents move to the `Recovered` state. This state reduces transmissibility of `INFLUENZA` for an agent to zero indefinitely.

```
    state Recovered {
        INFLUENZA.trans = 0
        wait()
        next()
    }
```

The only remaining state in INFLUENZA is the Import state, which is the starting state for the meta agent. This is specified by the meta_start_state = Import statement at the top of the condition. This state prompts the meta agent to infect ten random agents at the beginning of the simulation before waiting indefinitely.

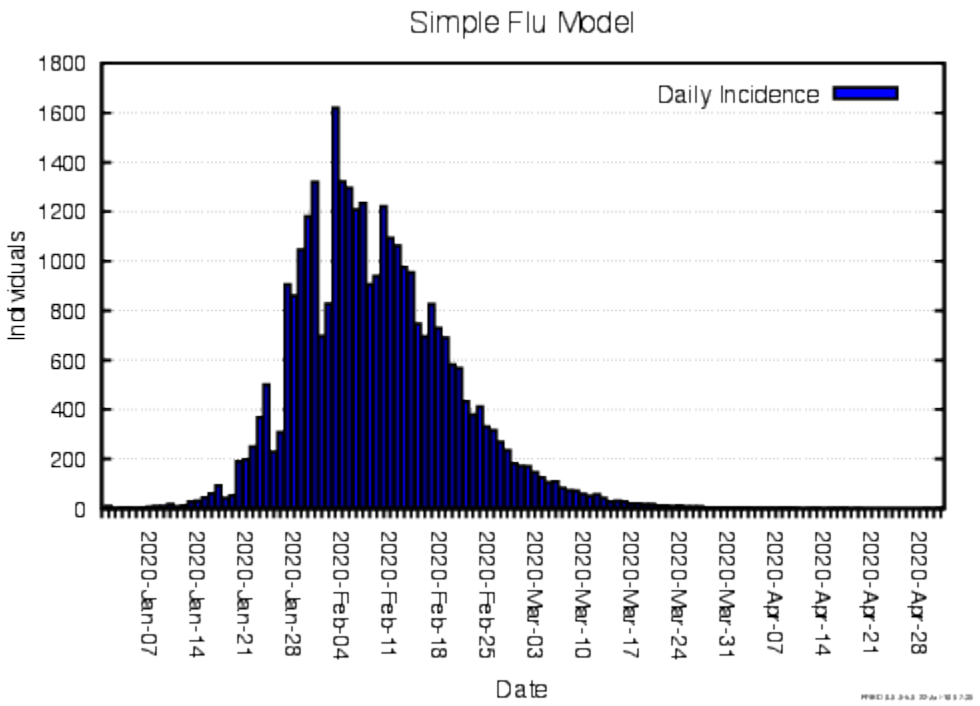
```
    state Import {
        import_exposures(INFLUENZA,10)
        wait()
        next()
    }
```

2.3 Sample Model Outputs

This model can be run using the FRED Local product. As discussed in the FRED Local Guide, you can gain direct shell access to the FRED Local Container and then use the **METHODS** file to execute the model. The **METHODS** file is a bash script that runs the model, using fred_plot to generate a histogram of the number of new infections per day.

```
$ docker exec -it fred /bin/bash
root@a48b40b88a53:/fred/models# ./METHODS

fred_job: starting job simpleflu at <date>

fred_compile -p main.fred
No errors found.
No warnings.

fred_job: running job simpleflu id 833 run 1 ...
fred_job: running job simpleflu id 833 run 2 ...
run_set 0 completed at <date>

fred_job: running job simpleflu id 833 run 3 ...
fred_job: running job simpleflu id 833 run 4 ...
run_set 1 completed at <date>

fred_job: finished job simpleflu <job key> at <date>

fred_plot: image_file = daily.pdf

root@a48b40b88a53:/fred/models#
```

The results are consistent with the INFLUENZA condition propagating through the specified population, resulting in agents moving to the Recovered state. Eventually, a large fraction of the population is immune and the condition can no longer transmit.



2.4 Summary

This tutorial introduces two concepts:

1. a **condition** with **states** that can be transmitted between agents.
2. a **meta agent** that introduces the condition to the population.

Within the states in `INFLUENZA`, we also used:

- `wait()`, which causes the agent to pause in a given state
- `next()`, which causes an agent to transition to a new state
- two forms of probabilistic behavior, using `lognormal()` to generate wait times and the combination of `next() with prob()` and `default()` to probabilistically transition an agent to symptomatic or asymptomatic infectious states.

CHAPTER

THREE

FLU WITH BEHAVIOR MODEL

3.1 Introduction

This model builds on the Simple Flu model (see `../simpleflu`) by equipping agents with rudimentary social distancing behavior. In the Simple Flu model, agents with the flu continue to behave in the same way as agents without the flu, and will continue to visit all the places they usually visit including work and school. In the Flu with Behavior model, agents who become infected with the flu and are symptomatic have a 50% chance of deciding to stay at home for the duration of time it takes them to recover.

3.2 Review of code implementing the model

The code that implements the Flu with Behavior model is contained in three `.fred` files:

- `main.fred`
- `simpleflu.fred`
- `stayhome.fred`

Here we review the code in these files, focusing on how features of the FRED language are used to implement the Flu with Behavior model described above.

3.2.1 `main.fred`

Like most FRED models, the entry point to the Flu with Behavior model is a file called `main.fred`. This file specifies the basic simulation control parameters and coordinates how relevant sub-models are loaded. The `simulation` code block from `main.fred` is given below

```
simulation {
    locations = Jefferson_County_PA
    start_date = 2020-Jan-01
    end_date = 2020-May-01
}
```

This is the same as the `simulation` code block from the Simple Flu model. It specifies that the model will be used to simulate the behavior of the synthetic population for Jefferson County, PA for the period January 1 2020 to May 1 2020. By default FRED produces simulation outputs quantifying the evolution of the number of agents in each state on a daily basis in the directory `$FRED_HOME/RESULTS/JOB/<job_num>/OUT/<run_num>/DAILY`. Here `job_num` and `run_num` are, respectively, the job and run numbers for the simulations and are determined at runtime.

The remaining lines in `main.fred` demonstrate the use of `include` statements to import sub-models.

```
include simpleflu.fred
include stayhome.fred
```

The use of separate files to contain sub-models helps to keep code organized, improves readability and promotes code reuse. Importantly the order that the sub-model files are specified here determines the order in which they will be processed by the compiler. This is important because FRED defines rules for how property definition statements are overridden by subsequent statements, and how additional State execution rules can be specified later in the program to modify behavior.

3.2.2 simpleflu.fred

`simpleflu.fred` is identical to the file of the same name in the Simple Flu model. It specifies the state update rules for the `INFLUENZA` Condition that models agents' status with respect to influenza infection. Specifically, each agent is in one of the following five states:

1. Susceptible to infection with influenza
2. Exposed to influenza
3. Infectious and symptomatic with influenza
4. Infectious and asymptomatic with influenza
5. Recovered following infection with influenza and assumed to be immune

Refer to the tutorial on the Simple Flu model for a complete explanation of the code in `simpleflu.fred`. For the purpose of this tutorial, consider the following code snippet which specifies the State rules for the `InfectiousSymptomatic` and `Recovered` States belonging to the `INFLUENZA` condition:

```
    state InfectiousSymptomatic {
        INFLUENZA.trans = 1
        wait(24* lognormal(5.0,1.5))
        next(Recovered)
    }

    ...

    state Recovered {
        INFLUENZA.trans = 0
        wait()
        next()
    }
```

Here `INFLUENZA.trans = 1` and `INFLUENZA.trans = 0` are action rules that cause agents entering the `INFLUENZA.InfectiousSymptomatic` state to change their transmissibility of influenza to 1, and agents entering the `INFLUENZA.Recovered` state to change their transmissibility of influenza to 0 (i.e. they are non-infectious). The wait rule `wait(24* lognormal(5.0,1.5))` causes each agent that becomes infectious and symptomatic to remain so for a number of days determined by sampling from a lognormal distribution with median=5.0 and dispersion=1.5. Finally the transition rule `next(Recovered)` causes agents to transition, deterministically, to the `Recovered` state once their period of infection has elapsed. The wait rule `wait()` and transition rule `next()` cause agents that enter the `Recovered` state to remain in that state indefinitely.

3.2.3 stayhome.fred

This file contains the code that implements the social distancing sub-model that is particular to the Flu with Behavior model. We first specify a new Condition, STAY_HOME

```
condition STAY_HOME {
    start_state = No

    state No {
        attend()
        wait()
        next()
    }

    state Yes {
        skip()
        attend(Household)
        wait()
        next()
    }
}
```

This condition demonstrates a feature of the FRED language which has not been discussed previously: the skip and attend actions. If an agent **skips** one of its usual places, then they do not attend that place even if they otherwise would. The statement skip() causes an agent to avoid *all* of its usual places. Conversely attend(Household) causes agents who were previously absent from their household to resume attending their household. The combination of the skip() and attend(Household) actions in the definition of the STAY_HOME.Yes state cause agents entering that state to *exclusively* attend their household (i.e. stay home). Note that the effects of skip persist until that agent executes an attend within this same condition.

All agents start in the No state. The attend() action causes the agent to resume its attendance at the places is usually attendance, and cancels any skip() action in this condition. It is significant that the start_state assigns No as the default state for the condition. The wait() wait rule and next() transition rule cause agents to remain in each state indefinitely unless their state is changed by another condition.

The remaining code blocks in stayhome.fred specify how agents decide to stay at home or not depending on their state with respect to the INFLUENZA condition.

```
state INFLUENZA.InfectiousSymptomatic {
    if (bernoulli(0.5)==1) then set_state(STAY_HOME,Yes)
}


state INFLUENZA.Recovered {
    set_state(STAY_HOME,No)
}
```

Here set_state is an action that updates an agent's state. The statement set_state(STAY_HOME,No) in the INFLUENZA.Recovered code block causes agents entering the INFLUENZA.Recovered change their STAY_HOME state to No–they are healthy again so they return to work and school. The statement if (bernoulli(0.5)==1) then set_state(STAY_HOME,Yes) in the INFLUENZA.InfectiousSymptomatic code block is an example of a *conditional* action rule. Any action rule can optionally include a conditional statement that causes the action to be executed (on an agent-by-agent basis) only if the associated predicate (or predicates) is true. In this case, each time an agent enters the INFLUENZA.InfectiousSymptomatic state, a Bernoulli trial with 0.5 probability of success is conducted (we might imagine the agent flips a coin). On a success, the agent changes its STAY_HOME state to Yes.

Because `stayhome.fred` is imported into `main.fred` after `simpleflu.fred`, the compiler appends the statements in the code blocks specified above to the end of the statements in the `INFLUENZA.InfectiousSymptomatic` and `INFLUENZA.Recovered` declarations in `simpleflu.fred`. It then interprets them in an order consistent with FRED's State order of execution rules. These stipulate that within a State:

1. all action rules are executed first,
2. then all wait rules are executed, and
3. finally all transition rules are executed.

Consequently the code blocks in the previous code snippet in combination with the relevant statements in `simpleflu.fred` result in the following *effective* definitions of the `INFLUENZA.InfectiousSymptomatic` and `INFLUENZA.Recovered` states for the simulation overall.

```
state InfectiousSymptomatic {
    INFLUENZA.trans = 1  # action rule
    if (bernoulli(0.5)==1) then set_state(STAY_HOME,Yes)  # action rule
    wait(24* lognormal(5.0,1.5))  # wait rule
    next(Recovered)  # transition rule
}


state Recovered {
    INFLUENZA.trans = 0  # action rule
    set_state(STAY_HOME,Yes,No)  # action rule
    wait()  # wait rule
    next()  # transition rule
}
```

This is an example of modifying the state rules of a State that is assumed to have been previously defined in the program.

3.3 Sample model outputs

3.3.1 Running the model

Here we demonstrate how to run the model described above, and explore three use cases showing how the model can be used to help address policy questions. Note that these instructions assume that you have configured your environment as described in the FRED Local Guide , with FRED running in a Docker container.

To run the Flu with Behavior model, run the FRED Local image and directly access the shell as describe in the Direct Shell Access section of the guide. The examples here assume you are running on a Linux or macOS system. These can be adapted to run on Windows as described in the guide. open a terminal and navigate to the directory where the Flu with Behavior model is saved, for example:

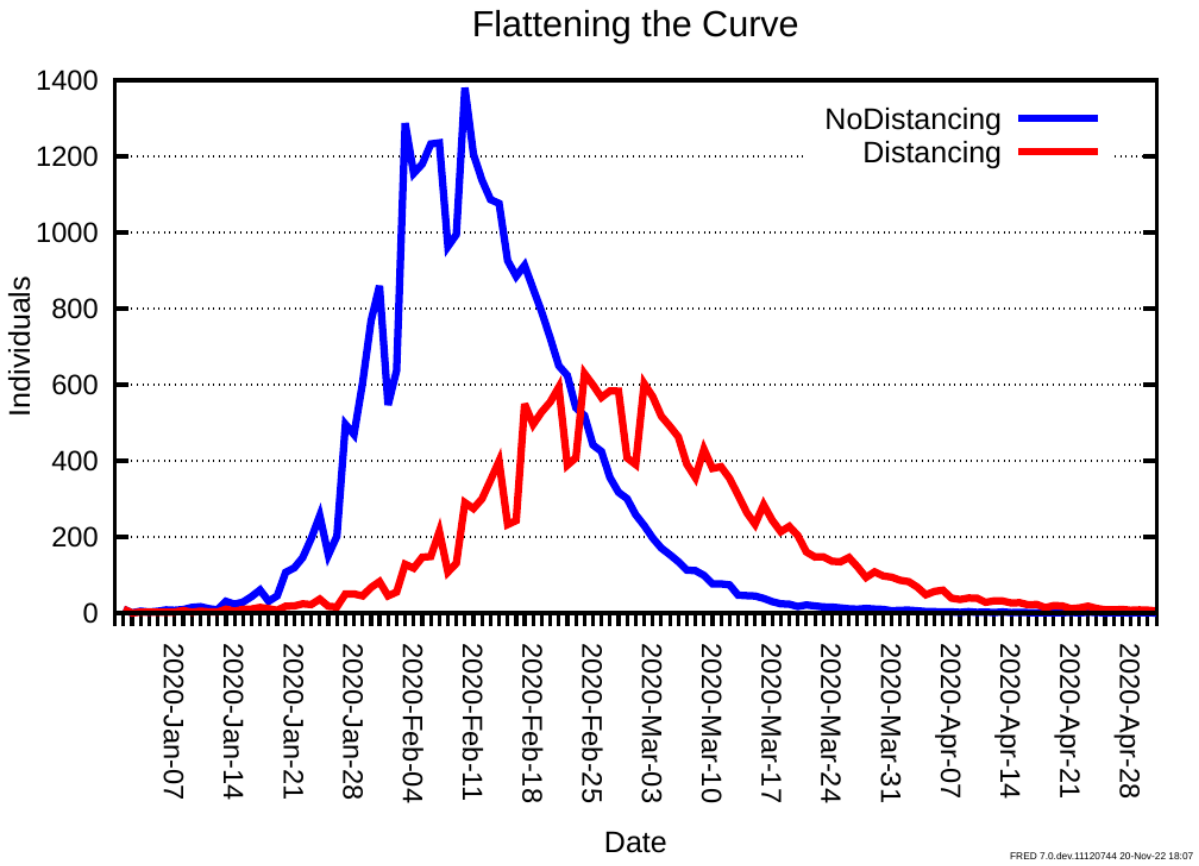
```
$ docker exec -it fred /bin/bash
root@a48b40b88a53:/fred/models# pwd
/fred/models
root@a48b40b88a53:/fred/models# cd FRED-tutorials/flu-with-behavior
root@a48b40b88a53:/fred/models/FRED-tutorials/flu-with-behavior#
```

The configuration, as described in the FRED Local Guide, ensures that the FRED environment variables and command-line tools are accessible.

We can now run the model with default settings and generate some visualizations of its outputs using the `./METHODS` script provided with the model files:

```
root@a48b40b88a53:/fred/models/FRED-tutorials/flu-with-behavior# ./METHODS
```

This generates the file `flatten.pdf` which contains a plot showing two time series that enable us to compare the number of individuals exposed to influenza under two different modeling **scenarios**: one with `NoDistancing` (the original Simple Flu model), and one with `Distancing` (the Flu with Behavior model).



We see that the social distancing behaviors included in the Flu with Behavior model appear to both delay and reduce the magnitude of the peak number of individuals exposed to the virus during the modeled epidemic.

In the following sections we demonstrate an exploratory model workflow in which we modify details of the model code to investigate:

1. The sensitivity of the model outputs to certain modeling assumptions
2. The effect of alternative policy decisions on the emergent dynamics of the epidemic

3.3.2 Vary likelihood that symptomatic individuals decide to stay home

A major modeling assumption in the Flu with Behavior model that is that all agents experiencing symptoms of flu decided to stay home with 50% probability. As this assumption is subject to uncertainty, our exploratory analysis of the model should include a sensitivity analysis of the model to this parameter.

To make varying the probability that symptomatic agents decide to stay home easier, we will convert it into a parameter called `prob_symp_stay_home`. To do this we modify the `stayhome.fred` file to declare `prob_symp_stay_home` as a variable with default value 0.5, and change the definition of the `INFLUENZA.InfectiousSymptomatic` state to use the new variable rather than the original hard coded value

```
variables {
    shared numeric prob_symp_stay_home
    prob_symp_stay_home = 0.5
}
...
state INFLUENZA.InfectiousSymptomatic {
    if (bernoulli(prob_symp_stay_home)==1) then set_state(STAY_HOME,Yes)
}
```

Now modify `main.fred` so that it loads a file called `parameters.fred` in addition to the model files `simpleflu.fred` and `stayhome.fred`

```
...
include simpleflu.fred
include stayhome.fred
include parameters.fred
```

We can now create a script that performs a parameter sweep, in which the model is run multiple times with each run using a different value for `prob_symp_stay_home`. This script could be written in any scripting language of the user's choice. Below we provide an example of such a script written in bash. This runs the model with the probability of agents with symptoms staying home set to 10%, 30%, 50%, and 70%. The script then plots a time series of the number of exposed individuals for each of these parameter values.

Note that this script can be created outside of the FRED Local Docker environment, and will still be available within the mounted `models` directory in the container.

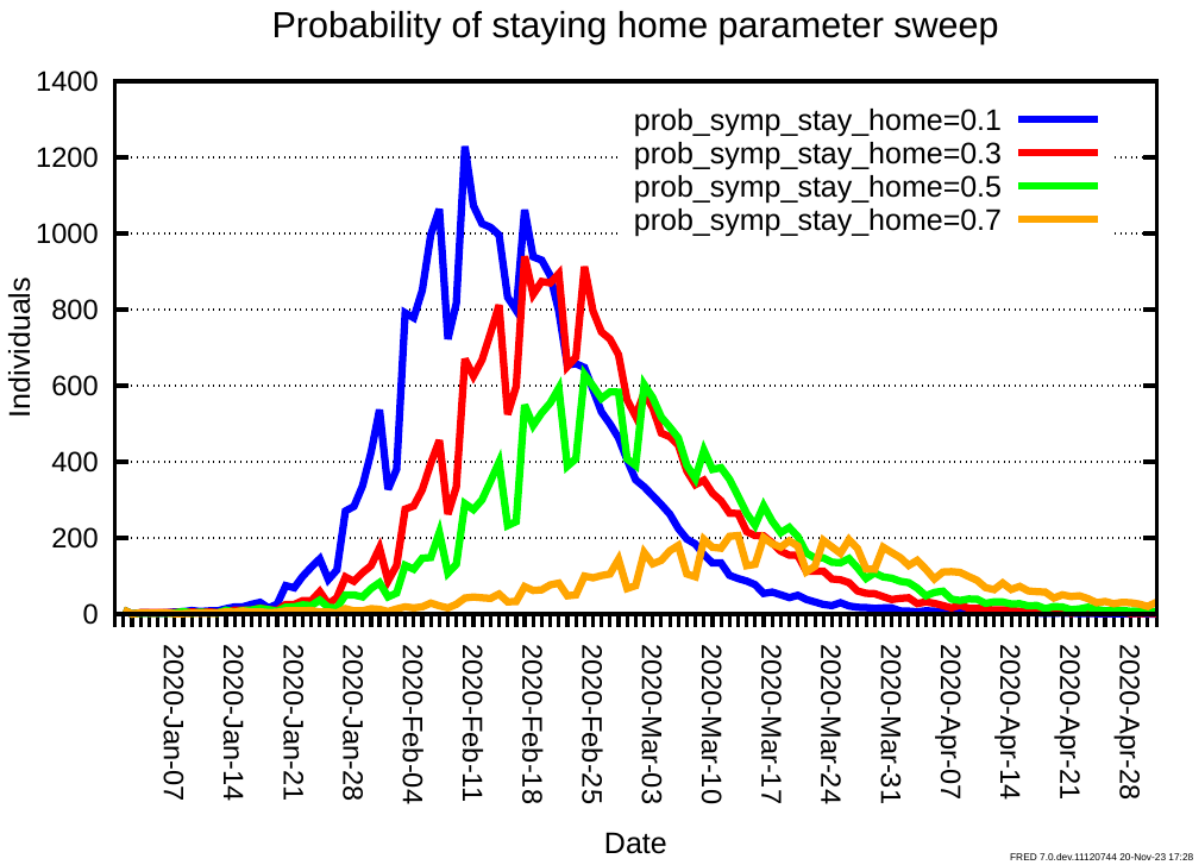
```
#! /usr/bin/env bash

PROBS_STAY_HOME=("0.1" "0.3" "0.5" "0.7")

for P in ${PROBS_STAY_HOME[@]}
do
    # Generate parameters.fred file
    printf "startup {\nprob_symp_stay_home = ${P}\n}\n" > parameters.fred

    # Determine the job name for the run with this parameter
    KEY="stay-home-prob=${P}"
    # Add job name to an accumulating list of job names
    KEY_LIST="${KEY_LIST}${KEY},"
    # Add parameter to an accumulating list of parameters
    PARAM_LIST="${PARAM_LIST}prob_symp_stay_home=${P},"
    PLOT_VARS="${PLOT_VARS}INFLUENZA.newExposed,"

    # Clear any existing results with the current key
```

(continues on next page)

(continued from previous page)

```
    fred_delete -f -k $KEY

    # Run 4 simulations (two in parallel) for the current parameter
    fred_job -k $KEY -p main.fred -n 4 -m 2
done

# Plot the results
fred_plot -o prob-stay-home -k $KEY_LIST \
    -v $PLOT_VARS \  # INFLUENZA.newExposed for every parameter
    -t "Probability of staying home parameter sweep" \
    -l $PARAM_LIST
```

Execute this script `prob-stay-home-sweep.sh` within the Docker environment. This should produce a file called `prob-stay-home.pdf`. Due to the stochastic nature of the simulation, your results should look similar but not exactly the same as the following chart.



As we might have expected, increasing the probability that individuals who become symptomatic decide to stay home decreases the number of individuals who are exposed to flu.

3.3.3 Individuals continue to attend School when sick

In this scenario we investigate the effect of causing agents who experience symptoms of flu to continue to attend school. This might represent the effect of a policy decision to discourage students from social distancing out of concern for the impact it would have on their education.

Reset the probability that symptomatic agents decide to stay home to 50% in `stayhome.fred`:

```
state INFLUENZA.InfectiousSymptomatic {
    if (bernoulli(0.5)==1) then set_state(STAY_HOME,Yes)
}
```

Create a copy of `stayhome.fred` called `workers-stayhome.fred`

```
$ cp stayhome.fred workers-stayhome.fred
```

Modify the actions in the `STAY_HOME.Yes` state in `workers-stayhome.fred` so that agents attend `School` as well as their `Household`

```
condition STAY_HOME {
    start_state = No

    state No {
        attend()
                wait()
                next()
        }

    state Yes {
        skip()
        attend(Household)
        attend(School)
        wait()
        next()
    }
}
```

Create a new `main.fred` that includes `workers-stayhome.fred` rather than the original `stayhome.fred`

```
$ cp main.fred main-workers-stayhome.fred
```

Modify `main-workers-stayhome.fred` so it includes `workers-stayhome.fred` instead of `stayhome.fred`

```
...
include simpleflu.fred
include workers-stayhome.fred
```

Write a bash script to run the simulations and plot the results, call this `workers-stayhome.sh`

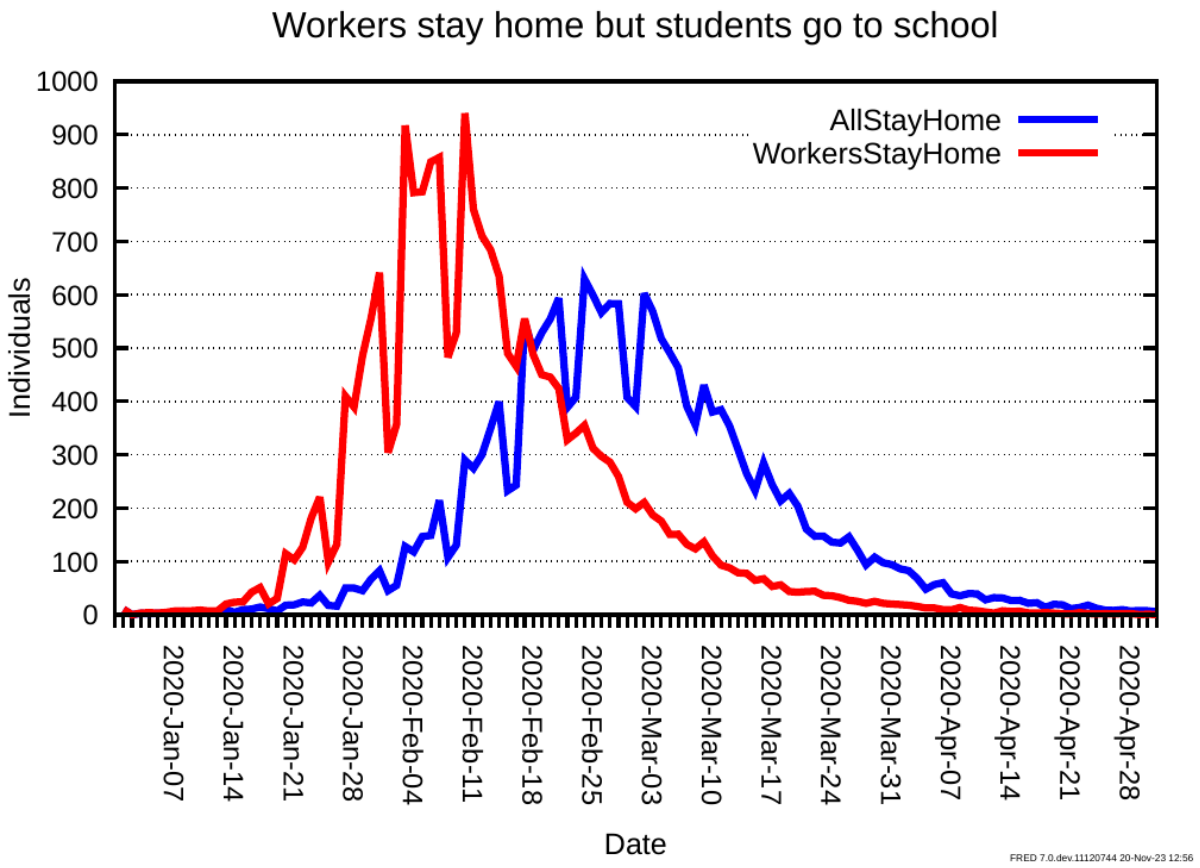
```
#! /usr/bin/env bash

fred_delete -f -k flu-with-behavior
fred_delete -f -k workers-stayhome

fred_job -k flu-with-behavior -p main.fred -n 4 -m 2
```

(continues on next page)

(continued from previous page)

```
fred_job -k workers-stayhome -p main-workers-stayhome.fred -n 4 -m 2

fred_plot -o workers-stayhome -k flu-with-behavior,workers-stayhome, \
    -v INFLUENZA.newExposed,INFLUENZA.newExposed \
    -t "Workers stay home but students go to school"  \
    -l AllStayHome,WorkersStayHome, --clean
```

Running this script in Docker produces output similar to the following.



We see that having only workers stay home and encouraging students to go to school is less effective at reducing the number of people exposed to the flu than if everyone considers staying home when they experience symptoms. Specifically, allowing everyone to stay home if they choose to do so upon becoming symptomatic reduces the number of exposures and delays the peak daily number of exposures compared to if students are forced to attend school.

3.3.4 Individuals continue to attend Work when sick

Here we consider the case where agents are allowed to stay home from school when they experience symptoms, but agents who go to work are not allowed to stay home. This is effectively the inverse of the previous scenario, and consequently many of the changes to the model files are analogous. This scenario might represent the effect of a policy decision to discourage workers from social distancing out of concern for the impact it would have on the economy.

Create a copy of `stayhome.fred` called `students-stayhome.fred`

```
$ cp stayhome.fred students-stayhome.fred
```

Modify the actions in the `STAY_HOME.Yes` state in `students-stayhome.fred` so that agents attend their `Workplace` as well as their `Household`

```
condition STAY_HOME {
    start_state = No

    state No {
        attend()
                wait()
                next()
        }

    state Yes {
        skip()
        attend(Household)
        attend(Workplace)
        wait()
        next()
    }
}
```

Create a new `main.fred` that includes `students-stayhome.fred` rather than the original `stayhome.fred`

```
cp main.fred main-students-stayhome.fred
```

Modify `main-workers-stayhome.fred` so it includes `students-stayhome.fred` instead of `stayhome.fred`

```
...
include simpleflu.fred
include students-stayhome.fred
```

Write a bash script to run the simulations and plot the results, call this `students-stayhome.fred`

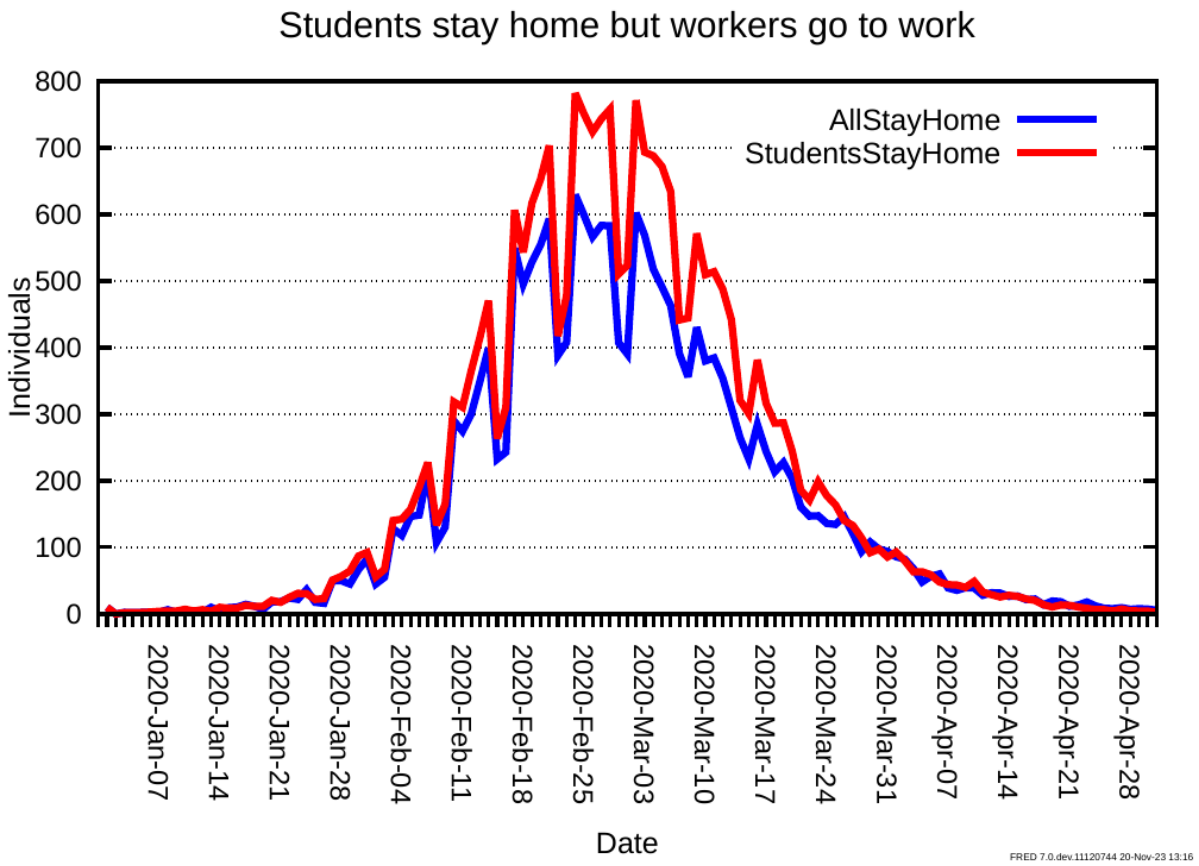
```
#! /usr/bin/env bash

fred_delete -f -k flu-with-behavior
fred_delete -f -k students-stayhome

fred_job -k flu-with-behavior -p main.fred -n 4 -m 2
fred_job -k students-stayhome -p main-students-stayhome.fred -n 4 -m 2

fred_plot -o students-stayhome -k flu-with-behavior,students-stayhome, \
          -v INFLUENZA.newExposed,INFLUENZA.newExposed \
```

(continues on next page)

(continued from previous page)

```
        -t "Students stay home but workers go to work"  \
        -l AllStayHome,StudentsStayHome, --clean
```

Running this script produces the following output



Like the scenario where students were forced to attend school but workers could choose to stay home, when students are allowed to stay home but workers must go to work, the number of agents exposed to the flu is higher than if everyone is allowed to stay home. However, unlike in the previous scenario, allowing workers as well as students stay home doesn't appear to cause a delay in the peak daily number of agents exposed to the flu compared to if only students are allowed to stay home.

3.4 Summary

3.4.1 Additional language features introduced

In addition to the features of the FRED language used in the Simple Flu model, this model also demonstrates the following features:

- The `skip` and `attend` actions are used in `stayhome.fred` to control the Places agents attend, depending on their conditions.
- State rule shadowing is used in `stayhome.fred` to modify the the rules for a state defined previously in the program.

- `stayhome.fred` contains an example of a conditional action rule that is only executed if a predicate is true.
- Use of variables to specify parameters in place of hard-coded data.

3.4.2 Modeling workflow concepts introduced

- Used a custom bash script to perform a parameter sweep

CHAPTER

FOUR

SCHOOL CLOSURE MODEL

4.1 Overview

This model studies the effects of school closures in Jefferson County, PA. The `main.fred` file includes three components:

- Influenza condition
- `school.fred`: a condition for group agents to close the school according to case numbers in their school or county as well as normally scheduled holiday closure dates. This condition also keeps track of students whose schools are closed.
- `parameters.fred` : a file written into by the `METHODS` script to modify the school closure policies and other variables.

4.1.1 Normal Closures

The schools are given a school schedule that closes the schools during summer, winter, and spring break. There are states for each break: `WinterBreak`, `SpringBreak`, and`SummerBreak`. In these states, school administrators (represented by `group agents`) close their schools and wait until the end of the break period. They pass into these states from the `CheckCalendar` state:

```
state CheckCalendar {
    wait(0)
    if (date_range(Dec-20,Jan-02)) then next(WinterBreak)
    if (date_range(Mar-10,Mar-15)) then next(SpringBreak)
    if (date_range(Jun-15,Aug-25)) then next(SummerBreak)
    default(Open)
}
```

Group agents go to the break states by checking the current date with `date_range()`. The group agents only pass into this check state if their school is not already affected by flu closures. No flu closures will take place if `school_closure_policy` is set equal to `NO_CLOSURE`.

4.1.2 School Closure due to Flu

The schools are also setup to actively check the number of cases of flu either in their school or in their county and close schools if the number passes a threshold set by one of the variables: `local_closure_trigger` or `global_closure_trigger`. After deciding to close due to the flu, the admin goes to the `Close` state, and the school remains closed for a time period set by the variable `days_closed`.

Global Flu Closure

The `GLOBAL_CLOSURE`, `LOCAL_CLOSURE`, and `NO_CLOSURE` variables are set to arbitrary but unique integers to get around the inability to assign strings to variables. The global closure option is selected by setting `school_closure_policy = GLOBAL_CLOSURE`. This variable passes group agents from the `CheckEpidemic` state in the `SCHOOL` condition to the `CheckGlobalEpidemic` state:

```
state CheckGlobalEpidemic {
    wait(0)
    if (global_closure_trigger <= current_count(INF.Is)) then next(Close)
    default(CheckCalendar)
}
```

This state checks the county flu count against the global threshold, `local_closure_trigger`. If the threshold is reached, then all group agents go to `Close` state.

Local Flu Closure

The local closure option is selected by setting `school_closure_policy = LOCAL_CLOSURE`. This variable passes group agents from the `CheckEpidemic` state in the `SCHOOL` condition to the `CheckLocalEpidemic` state:

```
state CheckLocalEpidemic {
    wait(0)
    if (local_closure_trigger <= current_count(INF.Is, School)) then next(Close)
    default(CheckCalendar)
}
```

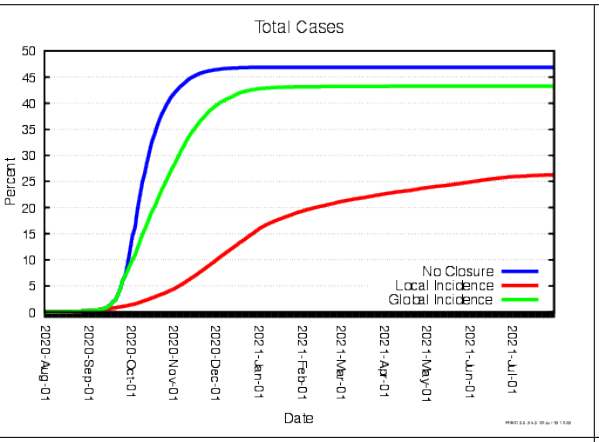
In this state, each admin checks if the number of infected agents in their own school is greater than the threshold set by `local_closure_trigger`. If the number is greater or equal, then the admin goes to the `Close` state and closes the school, otherwise they pass to the `check_calendar` state.

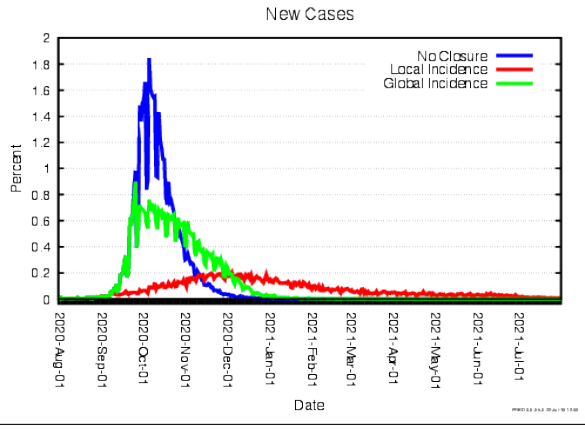
Student Tracking

The `StudentSchoolOpen` and `StudentSchoolOpen` states are included in the `SCHOOL` condition to keep track of how many students have their school closed/open over the course of time. Students are filtered into `StudentSchoolOpen` from the `Start` state with the conditional `if (is_member(School) & age < 18)`. The students then switch between the two states by checking if their school has been closed by an admin using the `is_temporarily_closed(<group>)` predicate. ##Results

4.1.3 Plotting Flu Cases

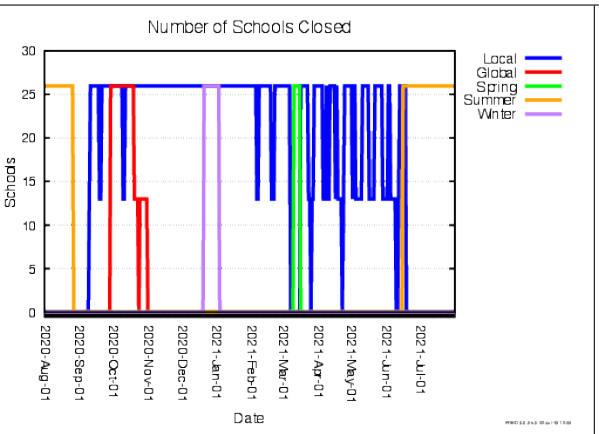
The following plots show the effectiveness of the different policies in limiting the number of total and daily infections as a percent of the county population.

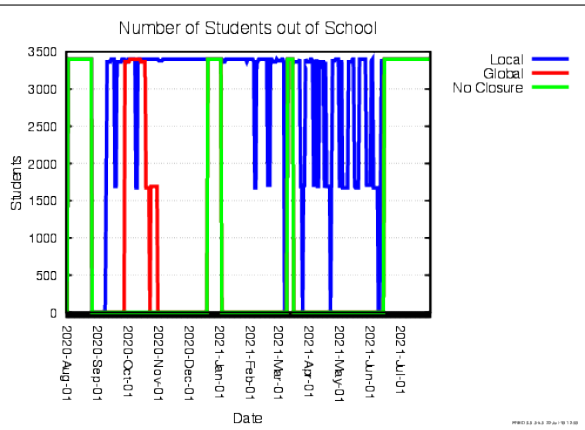




4.1.4 Plotting School Closures

The plots below show the number of schools closed and number of students out of school over time. Because the threshold for school closure is the same regardless of school size, the larger schools are more likely to close under the local closure policy. This results in a higher percentage of students out of school than percentage of schools closed for the local policy.





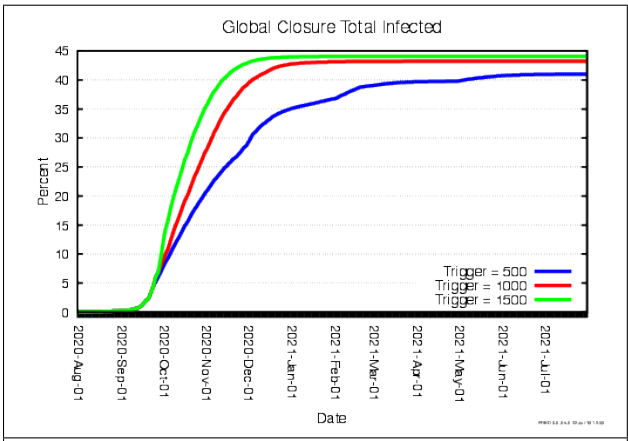
4.2 Modifying Closure Variables

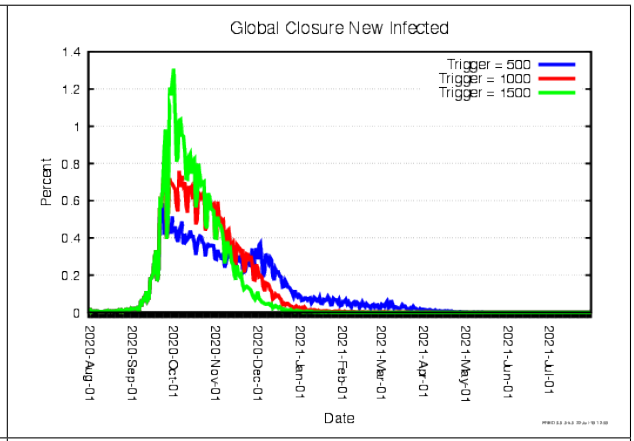
FRED variables are modified in the METHODS script for this model, which overwrites various combinations the `school_closure_policy`, `days_closed`, `global_closure_trigger`, and `local_closure_trigger` variables. For each combination of interest, the changes are written into the `parameters.fred` file and then `fred_job` is called to execute the model with the modified parameters. This produces a range of results as captured in the following figures.

In each figure, the modified variable values are shown in the legend for the figure. The other variables not represented in a figure use the following default values:

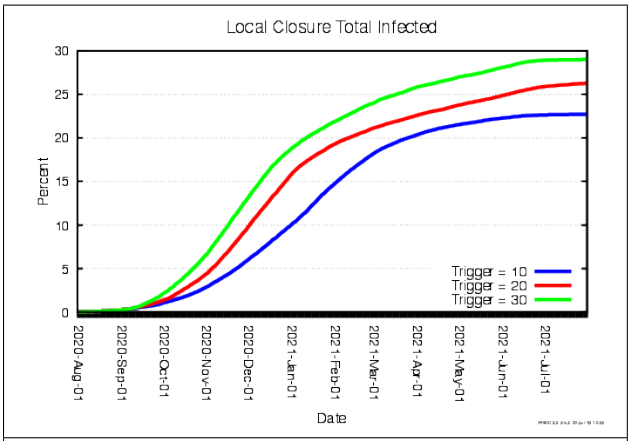
- `global_closure_trigger = 1000`
- `local_closure_trigger = 20`
- `days_closed = 28`

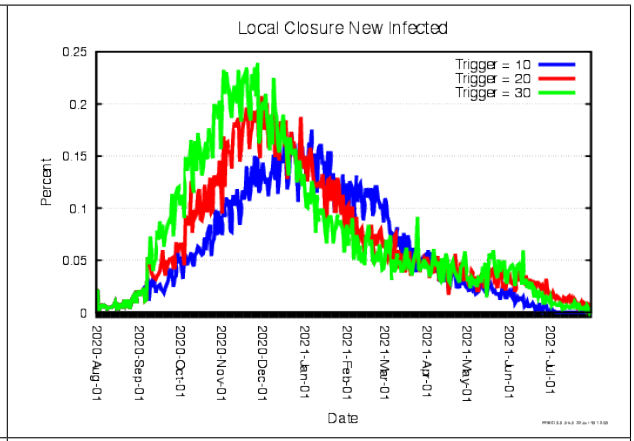
4.2.1 Changing the `global_closure_trigger` Variable



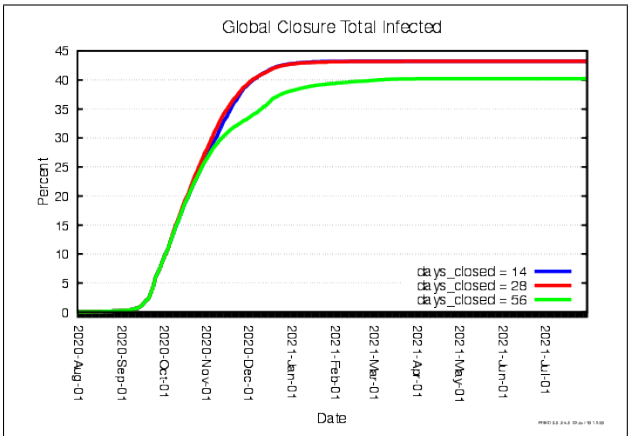


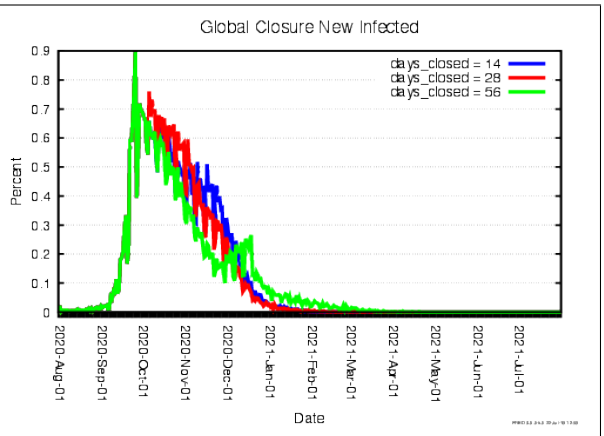
4.2.2 Changing the `local_closure_trigger` Variable



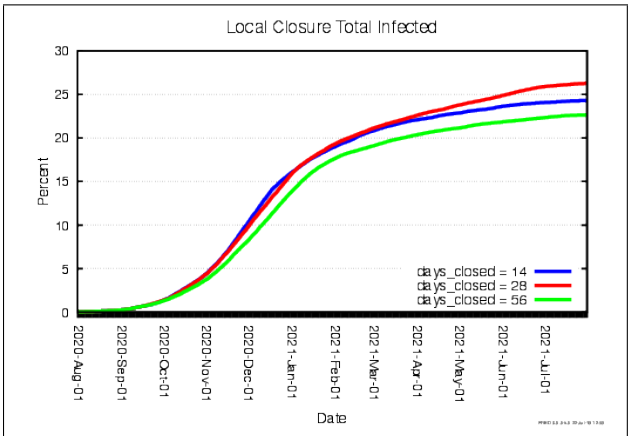


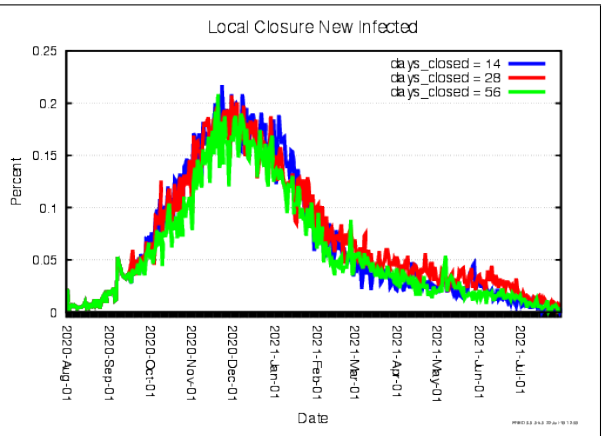
4.2.3 Changing the `days_closed` Variable Under Global Closure





4.2.4 Changing the `days_closed` Variable Under Local Closure





CHAPTER

FIVE

FLU WITH VACCINE MODEL

5.1 Introduction

This example builds on the the Flu with Behavior model (see `../flu-with-behavior`) by adding an `INFLUENZA_VACCINE` condition that allows agents to be exposed to a vaccine that protects the agent from influenza with a probability equal to `vaccine_effectiveness` by setting `INFLUENZA.sus = 0`.

5.2 Review of code implementing the model

The code that implements the Flu with Vaccine model is contained in five `.fred` files:

- `main.fred`
- `simpleflu.fred`
- `stayhome.fred`
- `vaccine.fred`
- `parameters.fred`

5.2.1 `main.fred`

This `main.fred` contains four basic components. The first two will be familiar from the Simple Flu and Flu with Behavior models. The simulation block defines the dates and physical location of the simulation. A series of `include` statement refer to other `.fred` files to reference conditions, states, and variables necessary to run the simulation.

The Flu with Vaccine `main.fred` also directly defines a global variable.

```
variables {
    shared numeric flu_delay
    flu_delay = 90
}
```

In this case, it is the number of days to wait before vaccinating the first agents. The next code chunk modifies the `INFLUENZA` condition (which was previously defined in `simpleflu.fred`)

```
condition INFLUENZA {
    meta_start_state = ImportDelay

    state ImportDelay {
```

(continues on next page)

(continued from previous page)

```
        wait(24 * flu_delay)
        next(Import)
    }
}
```

Specifically, it directs the meta-agent to start in the `ImportDelay` state and transition to `Import` after the delay. Looking back to `simpleflu.fred`, we can see that `meta_start_state = Import`. Calling `include simpleflu.fred` prior to this new `condition` block in the `main.fred` script causes the settings in the initial file to be overwritten. Now, rather than initializing in `Import` and immediately infecting some of the agents with `INFLUENZA`, the meta-agent will begin in `ImportDelay` and wait the parameterized delay before advancing to `Import` and assigning infections.

5.2.2 simpleflu.fred

`simpleflu.fred` is identical to the file of the same name in the Simple Flu model. It specifies the state update rules for the `INFLUENZA` Condition that models agents' status with respect to influenza infection. Specifically, each agent is in one of the following five states:

1. Susceptible to infection with influenza
2. Exposed to influenza
3. Infectious and symptomatic with influenza
4. Infectious and asymptomatic with influenza
5. Recovered following infection with influenza and assumed to be immune

Refer to the tutorial on the Simple Flu Model for a complete explanation of the code in `simpleflu.fred`. For the purpose of this tutorial, consider the following code snippet which specifies the State rules for the `InfectiousSymptomatic` and `Recovered` States belonging to the `INFLUENZA` condition:

```
    state InfectiousSymptomatic {
        INFLUENZA.trans = 1
        wait(24* lognormal(5.0,1.5))
        next(Recovered)
    }

    ...

    state Recovered {
        INFLUENZA.trans = 0
        wait()
        next()
    }
```

Here `INFLUENZA.trans = 1` and `INFLUENZA.trans = 0` are action rules executed by agents entering the `INFLUENZA.InfectiousSymptomatic` state to change their transmissibility of influenza to 1, and agents entering the `INFLUENZA.Recovered` state to change their transmissibility of influenza to 0 (i.e. they are non-infectious). This change is accomplished by multiplying condition transmissiblity by the agent's transmissibility, but in this case `INFLUENZA` transmissibility is already equal to 1. The wait rule `wait(24* lognormal(5.0,1.5))` causes each agent that becomes infectious and symptomatic to remain so for a number of days determined by sampling from a lognormal distribution with median=5.0 and dispersion=1.5. Finally the transition rule `next(Recovered)` causes agents to transition, deterministically, to the `Recovered` state once their period of infection has elapsed. Once in `Recovered`, transmissibility is set to 0 and the agent waits indefinitely (`wait()`). The `next()` transition has no effect, but is required to have any following rule.

5.2.3 stayhome.fred

`stayhome.fred` is identical to the file of the same name in the Flu with Behavior model. This file defines new behavior for agents: while symptomatic with an influenza infection (specifically, while in the `InfectedSymptomatic` state of the `INFLUENZA` condition), agents will probabilistically stay at home all day rather than attending work or school.

5.2.4 vaccine.fred

`vaccine.fred` defines the variables, condition, and states (for both agents and the meta-agent) required to simulate basic vaccination behavior in our population.

The primary change to this simulation, and the entirety of this file, is to create an `INFLUENZA_VACCINE` condition with states that influence both this condition and the `INFLUENZA` condition. Agents begin in `Start` and advance to `Considering` with the probability `willing_to_consider`, defined in the `variables` block of the current file. Any agent that does not advance to `Considering` goes instead to the `Excluded` state for `INFLUENZA_VACCINE` and stays there permanently. These agents will not gain `INFLUENZA` immunity except by contracting the illness.

```
state Considering {
    INFLUENZA_VACCINE.sus = 1
    wait()
    next()
}
```

Agents that move to `Considering` have their susceptibility to `INFLUENZA_VACCINE` set to 1 and then wait to have the vaccine transmitted to the by the meta-agent.

```
state Decide {
    INFLUENZA_VACCINE.sus = 0
    wait(24)
    next(Taker)
}

state Taker {
    wait(24 * days_until_effective)
    next(Immune) with prob(vaccine_effectiveness)
    default(Failed)
}
```

Once an agent in `Considering` is exposed by the meta-agent, they move to the `Decide` state. These agents change their susceptibility to the vaccine to zero, wait 24 hours, and then "take" the vaccine, moving to the `Taker` state. Here, agents wait until the vaccine becomes effective, defined in the `variables` block as `days_until_effective`, and then move to either `Immune` with the `vaccine_effectiveness` probability or `Failed` otherwise. Agents in the `Immune` state modify their `INFLUENZA` susceptibility to zero and can no longer be infected with that condition. The `Failed` state does not modify `INFLUENZA` susceptibility, mimicking a vaccination failure. Agents in both of these states remain there indefinitely.

```
meta_start_state = ImportStart
start_state = Start
transmission_mode = proximity
transmissibility = 1
exposed_state = Decide

...
```

(continues on next page)

(continued from previous page)

```
state ImportStart {
    wait(24 * vaccine_delay)
    next(ImportVaccine)
}

state ImportVaccine {
    import_per_capita(initial_vaccines)
    wait()
    next()
}
```

The remaining two states apply only to the meta-agent. At the top of the `INFLUENZA_VACCINE` condition, the meta-agent is started in `ImportStart`. We also define that the condition will be transmitted via proximity and that agents exposed to the condition will advance to the `Decide` state.

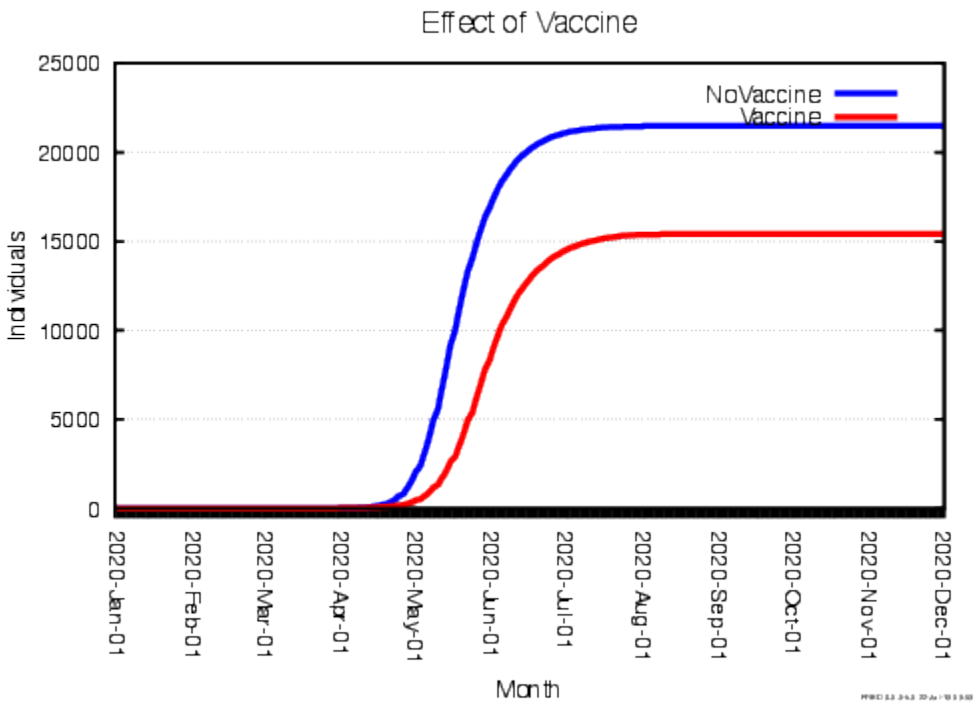
When the meta-agent begins in `ImportStart`, it waits the number of days defined by `vaccine_delay` (plus 1 hour) and then advances to `ImportVaccine`. During each time step, the meta-agent changes state before the ordinary agents, so it is common to have the meta-agent wait 1 time step at the start, so that ordinary agent have time to execute the rules in their start state. In this case, some ordinary agents end up on the `Considering` state at time `0`, where they become susceptible to vaccination. In this `ImportVaccine` state, the meta-agent exposes some proportion of the `Considering` agents to the vaccine (moving them to `Decide`). This proportion is defined in the `variables` block as `initial_vaccines`, though we will alter this value in the `parameters.fred` file.

5.2.5 `parameters.fred`

A `parameters.fred` file is also introduced in this model. This is largely a convenience for programmatically modifying the simulation parameters with a bash script. `parameters.fred` is the last file `included` in `main.fred`, allowing the contained values to overwrite any that were specified in earlier-imported files. The `METHODS` file in this directory defines two different FRED jobs by sequentially creating two different `parameters.fred` files and running `fred_job` with each of those files.

5.3 Sample Model Outputs

This model is run using the `METHODS` file in a FRED Local container. Running this file produces two sets of outputs that differ only in the proportion of the population that is originally vaccinated, 0% or 30%. Ideally, there would be a testing step in development to make sure that vaccines are being transmitted and imparting immunity (or reducing susceptibility) in the way that is expected. We can see that initially vaccinating 30% of the population reduces the cumulative infections substantially, providing evidence that the `INFLUENZA_VACCINE` condition is functioning as expected.



5.4 Summary

In addition to the features from previous tutorial modules, this model also includes:

- A second, transmissible `condition` can alter the behavior of the original `condition` by directly changing susceptibility.
- The use of ordered imports overwrites earlier parameter values.
- An example of modifying parameters in successive simulations by rewriting `parameters.fred`.