**Agent-Based Modeling with FRED:**

**A Programmer's Guide to FRED**

**by John Grefenstette and Donald Burke**

**[Draft – 18 May 2020 – Do Not Distribute]**

Table of Contents

[Chapter 1: Overview of Agent Based Modeling 1](#_Toc40674543)

[What is FRED? 1](#_Toc40674544)

[Agent-based Models 1](#_Toc40674545)

[Foundational Concepts in FRED 1](#_Toc40674546)

[The FRED Simulation Cycle 2](#_Toc40674547)

[The Modeling Process 3](#_Toc40674548)

[Focus on the purpose of a model 3](#_Toc40674549)

[Intrinsic, endogeneous and exogeneous features of the model 3](#_Toc40674550)

[Sources of data 3](#_Toc40674551)

[Building a conceptual model 3](#_Toc40674552)

[Implementing the model in FRED 3](#_Toc40674553)

[Model verification 3](#_Toc40674554)

[Calibration 3](#_Toc40674555)

[Sensitivity analysis 3](#_Toc40674556)

[Uncertainty analysis 3](#_Toc40674557)

[Costs and benefits of agent-based models 3](#_Toc40674558)

[Modeling with FRED 3](#_Toc40674559)

[Design Goals of FRED 4](#_Toc40674560)

[Efficiency with Large Populations 4](#_Toc40674561)

[Realistic Population Models 4](#_Toc40674562)

[Track Conditions at Population Level 4](#_Toc40674563)

[Limited Active Set of Individuals 5](#_Toc40674564)

[Extensibility 5](#_Toc40674565)

[Suitability of FRED 5](#_Toc40674566)

[FRED Versions 5](#_Toc40674567)

[Revision Notes 6](#_Toc40674568)

[Chapter 2: Synthetic Populations 12](#_Toc40674569)

[Definition 12](#_Toc40674570)

[Data sources 12](#_Toc40674571)

[Process 12](#_Toc40674572)

[Projections of Future Demographics 12](#_Toc40674573)

[Chapter 3: Declarative Programming in FRED 14](#_Toc40674574)

[A Minimal FRED Program 14](#_Toc40674575)

[Running a FRED Program 14](#_Toc40674576)

[The Structure of a FRED Program 15](#_Toc40674577)

[Use Statements 16](#_Toc40674578)

[Include Statements 16](#_Toc40674579)

[Property Statements 16](#_Toc40674580)

[Condition and States 16](#_Toc40674581)

[A Note for Previous FRED Users 17](#_Toc40674582)

[Chapter 4: Agents and Their Properties 20](#_Toc40674583)

[Conditions 20](#_Toc40674584)

[States 21](#_Toc40674585)

[Meta-Agents 21](#_Toc40674586)

[The Import Agent 21](#_Toc40674587)

[Administrators 22](#_Toc40674588)

[Meta-Agents and State Spaces 23](#_Toc40674589)

[Variables 23](#_Toc40674590)

[Global Variables 23](#_Toc40674591)

[Personal Variables 23](#_Toc40674592)

[Initializing Variables 24](#_Toc40674593)

[Factors 25](#_Toc40674594)

[Factors based on the simulation run 25](#_Toc40674595)

[Factors based on time and dates 25](#_Toc40674596)

[Factors based on the agent's demographics 26](#_Toc40674597)

[Factors based on agent's own state 26](#_Toc40674598)

[Factors based on other agents 26](#_Toc40674599)

[Factors based on the agent's groups 26](#_Toc40674600)

[Factors based on the agent's networks 27](#_Toc40674601)

[Expressions 28](#_Toc40674602)

[List Variables and List-Valued Expressions 29](#_Toc40674603)

[Functions within Expressions 30](#_Toc40674604)

[Predicates 33](#_Toc40674605)

[Comparisons 33](#_Toc40674606)

[Meta-Agent Predicates 33](#_Toc40674607)

[Place Predicates 33](#_Toc40674608)

[Date Predicates 34](#_Toc40674609)

[Exposure Predicates 34](#_Toc40674610)

[Symbolic Values in Predicates 34](#_Toc40674611)

[Negating Predicates 34](#_Toc40674612)

[Chapter 5: Mixing Groups 35](#_Toc40674613)

[Declaring a Mixing Group 35](#_Toc40674614)

[Maximum Place Size 35](#_Toc40674615)

[Group Schedules 35](#_Toc40674616)

[Places 38](#_Toc40674617)

[User-Defined Place Types 38](#_Toc40674618)

[Partitions 40](#_Toc40674619)

[Inviting agents to a place 41](#_Toc40674620)

[Networks 41](#_Toc40674621)

[Joining a Network 42](#_Toc40674622)

[Adding Edges 42](#_Toc40674623)

[Deleting Edges 42](#_Toc40674624)

[Quitting a Network 43](#_Toc40674625)

[An Agent's Degree in a Network 44](#_Toc40674626)

[Chapter 6: Rules for States 45](#_Toc40674627)

[Action Rules 46](#_Toc40674628)

[Demographics Actions 46](#_Toc40674629)

[Actions Affecting an Agent's Groups 46](#_Toc40674630)

[Actions that Change an Agent's State 47](#_Toc40674631)

[Actions that Change an Agent's Variables 47](#_Toc40674632)

[Actions that Produce Outputs 48](#_Toc40674633)

[Administrative Actions 48](#_Toc40674634)

[Import Actions 49](#_Toc40674635)

[Wait Rules 50](#_Toc40674636)

[Useful Distributions for State Durations 50](#_Toc40674637)

[Transition Rules 52](#_Toc40674638)

[Probabilistic Rules 52](#_Toc40674639)

[Default Next State Rules 52](#_Toc40674640)

[Chapter 7: State Transitions 53](#_Toc40674641)

[Probabilistic Rules 53](#_Toc40674642)

[Default Next State Rule and Normalization 54](#_Toc40674643)

[Example: Using Rules to Represent Health Risks 54](#_Toc40674644)

[Chapter 8: Transmission 57](#_Toc40674645)

[Transmissibility 57](#_Toc40674646)

[The Transmissibility of a Condition 57](#_Toc40674647)

[The Transmissibility of an Agent 58](#_Toc40674648)

[Transmission by the Import Agent 58](#_Toc40674649)

[Transmission in Places 60](#_Toc40674650)

[Transmission by Rate 60](#_Toc40674651)

[Transmission by Probability 62](#_Toc40674652)

[Transmission in Networks 64](#_Toc40674653)

[Transmission of Places 66](#_Toc40674654)

[Cross-Condition Transmission 67](#_Toc40674655)

[Chapter 9: FRED Output 69](#_Toc40674656)

[Output Files 69](#_Toc40674657)

[Time Series Output 69](#_Toc40674658)

[Health Records File 72](#_Toc40674659)

[Individual Report Files 74](#_Toc40674660)

[Network Files 74](#_Toc40674661)

[Gelphi for Network Display and Analysis 77](#_Toc40674662)

[The LOG File 78](#_Toc40674663)

[Errors and Warnings 78](#_Toc40674664)

[Plotting and Visualization 78](#_Toc40674665)

[Dormant States 78](#_Toc40674666)

[Chapter 10: FRED Simulation Information Management System 80](#_Toc40674667)

[Running FRED directly 80](#_Toc40674668)

[Example: running FRED 80](#_Toc40674669)

[FRED command line options 81](#_Toc40674670)

[The FRED Compiler 81](#_Toc40674671)

[FRED commands 82](#_Toc40674672)

[Job Control 84](#_Toc40674673)

[Chapter 11: FRED Web Interface 87](#_Toc40674674)

[Chapter 12: The FRED Library 88](#_Toc40674675)

[Learning about the FRED Library 88](#_Toc40674676)

[Using a FRED Library 91](#_Toc40674677)

[Conditions Defined in Modules 91](#_Toc40674678)

[The FRED Community 92](#_Toc40674679)

[Chapter 13: Programming Tips 93](#_Toc40674680)

[S-E-I-R Model 93](#_Toc40674681)

[Changes in Personal Behavior Based on Symptoms 95](#_Toc40674682)

[Measuring Where Transmissions Occur 96](#_Toc40674683)

[Using Group-Related Variables 98](#_Toc40674684)

[Writing Loops 100](#_Toc40674685)

[School Closure by Admin Agents 103](#_Toc40674686)

[School Closure of Individual Schools Trggered by an Epidemic 106](#_Toc40674687)

[Seasonality by Month 107](#_Toc40674688)

[Seasonality by Day of Year 108](#_Toc40674689)

[Using Elevation Data 110](#_Toc40674690)

[Surveys: Taking a Sample of the Population 113](#_Toc40674691)

[Taking a Temperature Survey of the Population 115](#_Toc40674692)

[Selecting the Closest Place 117](#_Toc40674693)

[Mapping a Catchment Area 119](#_Toc40674694)

[Changing the Contact Rates of Places 125](#_Toc40674695)

[Density-Based Transmission in Urban vs Rural Locations 128](#_Toc40674696)

[Small Ad Hoc Gatherings 131](#_Toc40674697)

[Mass Gatherings 136](#_Toc40674698)

[Chapter 14: Applications 140](#_Toc40674699)

[Pandemic Influenza 140](#_Toc40674700)

[Chapter 15: Calibration 144](#_Toc40674701)

[THE FRED Contact Matrix 144](#_Toc40674702)

[Acknowledgments 145](#_Toc40674703)

[Appendix A: The Structure of the FRED SImulation Engine 146](#_Toc40674704)

[Core Module 146](#_Toc40674705)

[Fred 146](#_Toc40674706)

[Global 146](#_Toc40674707)

[Age\_Report 146](#_Toc40674708)

[Utils 146](#_Toc40674709)

[Date 146](#_Toc40674710)

[Events 146](#_Toc40674711)

[Random 146](#_Toc40674712)

[State\_Space 146](#_Toc40674713)

[Parser 146](#_Toc40674714)

[Factor 146](#_Toc40674715)

[Expression 146](#_Toc40674716)

[Predicate 146](#_Toc40674717)

[Clause 146](#_Toc40674718)

[Rule 146](#_Toc40674719)

[Geospatial Module 146](#_Toc40674720)

[Geo 147](#_Toc40674721)

[Abstract\_Grid 147](#_Toc40674722)

[Abstract\_Patch 147](#_Toc40674723)

[Admin\_Division 147](#_Toc40674724)

[State 147](#_Toc40674725)

[County 147](#_Toc40674726)

[Census\_Tract 147](#_Toc40674727)

[Block\_Group 147](#_Toc40674728)

[Neighborhood\_Layer 147](#_Toc40674729)

[Neighborhood\_Patch 147](#_Toc40674730)

[Regional\_Layer 147](#_Toc40674731)

[Regional\_Patch 147](#_Toc40674732)

[Visualization\_Layer 147](#_Toc40674733)

[Visualization\_Patch 147](#_Toc40674734)

[Agent Module 147](#_Toc40674735)

[Person 147](#_Toc40674736)

[Demographics 147](#_Toc40674737)

[Link 147](#_Toc40674738)

[Travel 147](#_Toc40674739)

[Preference 147](#_Toc40674740)

[Mixing Module 147](#_Toc40674741)

[Group\_Type 147](#_Toc40674742)

[Place\_Type 147](#_Toc40674743)

[Network\_Type 147](#_Toc40674744)

[Group 147](#_Toc40674745)

[Place 147](#_Toc40674746)

[Network 147](#_Toc40674747)

[Household 147](#_Toc40674748)

[Hospital 147](#_Toc40674749)

[Epidemic Module 147](#_Toc40674750)

[Condition 148](#_Toc40674751)

[Epidemic 148](#_Toc40674752)

[Natural\_History 148](#_Toc40674753)

[Transmission 148](#_Toc40674754)

[Environmental\_Transmission 148](#_Toc40674755)

[Network\_Transmission 148](#_Toc40674756)

[Proximity\_Transmission 148](#_Toc40674757)

[Appendix B: Errors and Warnings 149](#_Toc40674758)

# Chapter 1: Overview of Agent Based Modeling

## What is FRED?

FRED is a programming language and modeling platform for simulating how a population changes over time. The population might change in many ways, including the number of people (through birth and death) or the age structure of the population (how many people there are in each age group), but the most interesting kinds of population changes concern how many people, and exactly which people, might be in a given condition at a given time. FRED enables the user to explore a wide range of conditions and how they spread within a population over time. Conditions might include diseases, behaviors, attitudes, economic status, or just about any other kind of condition the modeler wants to study.

The name **FRED** stand for a **FR**amework for **E**pidemiological **D**ynamics. FRED is not a model. FRED is a tool for building epidemiological models. A common definition of epidemiology is “the study and analysis of the patterns, causes, and effects of health and disease conditions in defined populations (Wikipedia)”. Indeed, FRED has been used to model the spread of both infectious and noninfectious diseases. And epidemiology also traditionally includes health-related conditions such as obesity, drug use, etc. But there is also the casual and non-technical use of the term: epidemiology encompasses *the study of any pattern that can spread in a population.* For example, one may speak of the epidemic of gun violence, or the epidemic of panic, or the epidemic of social media use. In this usage, the term *epidemic* refers to any rapidly increasing patterns in a population (see examples in Malcolm Gladwell's book, *The Tipping Point*). FRED lets us study the *dynamics*of epidemics by tracking how such patterns in populations vary over time and space.

## Agent-based Models

The key feature of agent-based models is that they include each person in the model, along with social contacts and interactions with the environment. This enables the model to include individual responses and behaviors in the model. Such individual responses can vary according to the individual’s characteristics, including demographics (age, sex, race, etc.), as well as the individual’s interactions with members of various social interaction groups, such as their neighborhood, school or workplace. Because individuals in ABMs are located within a specific geographical space, the models can be used to investigate interactions between individuals and spatially distributed resources such as health care facilities. In summary, agent-based models let us study how interactions among individuals and their environment can result in patterns of population behavior. This approach has been shown particularly useful in understanding or predicting the impact of public policy and programs on population health.

## Foundational Concepts in FRED

FRED incorporates fundamental assumptions about space and time. Agents in FRED live in an abstract version of three-dimensional space. They interact with other agents in places with specific geo-locations specified by latitude and longitude coordinates. Some places correspond to actual places in the real world (such as schools in the U.S. models), and some places are artificially generated but still present the spatial distributions in the real world (such as workplaces in the U.S. models). There is a limited form of support for three dimensions, in that all places in FRED have been assigned an estimated elevation (meters above sea level) based on US Geological Survey data. This allows models to take elevation into account when modeling, for example, the risk of flooding.

For all locations in the United States and selected other countries, FRED can use **synthetic populations** based on realistic estimates derived from census data, land-use surveys, educational and employment databases, so that the FRED population of individuals accurately reflects that statistical distribution of population size, household characistics (including household size, age, income and race), school enrollment, workplace sizes, and commuting patterns within a specific geographical area. FRED’s use of synthetic population make it possible to explore models of specific conditions in specific cities, counties or states.

Agents in FRED have built-in demographic characteristics including age, sex, and race. Agents are grouped into households that reflect the household density in statistically accurate ways as the census block group level. Agents may influence or may be influenced by other agents with whom they come in contact within interaction groups including households, neighborhood, schools, and workplaces. Some interaction groups are built into FRED and other can be defined by the user. Agents in FRED have a rich set of properties that can be used to influence the future states and activities of the agent itself and well as other agents. Like the interaction groups, some properties of agents are built into FRED and others can be defined by the user.

The declarative programming style of FRED provides several advantages for agent-based modeling:

* No traditional computer programming is required.
* You can focus on scientific effort (e.g. data collection, conceptual modeling, experimental design).
* FRED provides a simple workflow environment for you and manages all the data produced by the simulation and associated metadata.

## The FRED Simulation Cycle

FRED provides a fully functioning agent-based simulation system that can be completely customized as needed. FRED is a discrete time model and the time step is one hour. The duration of simulations can be from one day to 100 years. To use FRED, the user creates a FRED model in the FRED progamming language. The FRED platform translates a FRED program into the following simulation process:

1. Select the location for the simulation.
2. Select the start and end dates for the simulation.
   1. The simulation begins at midnight on the start date.
   2. The simulation end at 11pm on the end date.
3. For each hour of the simulation, perform the following for each user-defined Condition:
   1. Identify all agents that need to be updated according to the user-defined rules.
   2. For each identified agent:
      1. Select the agent's next state according to the user-defined rules.
      2. Perform any actions that are associated with the agent's next state.
   3. If the Condition involves interactions among agents, simulate the agent interactions within the defined interaction groups. Interactions may result in some agents changing their states.
4. After each day of simulation, record statistics about the levels of each Condition in the population.

The process described above is called a *run* of the model. Since FRED models are stochastic, it is usually desirable to perform several runs in order to produce meaningful statistics about the performance of the model. A set of runs of the same model is called a FRED *job*. Upon completion of a FRED job, the user can obtain REPORTs of the output, including spreadsheets, plots, or videos that display the location of user-selected events on a Report of the simulation area.

The following chapters will provide details on how the user defines the components of a FRED model, including:

* Agent conditions and states
* Rules for initializing agents
* Rules for state transitions for agents
* Interaction groups
* Interaction parameters such as contact rates and the transmissibility or susceptibility of agents for each condition

Later chapters will also address the FRED Simulation Information Manangement System that supports the reproducibility of computational experiments, and the FRED plotting and visualization features.

## The Modeling Process

To do: This section describes the modeling process in general.

### Focus on the purpose of a model

### Intrinsic, endogeneous and exogeneous features of the model

### Sources of data

### Building a conceptual model

### Implementing the model in FRED

### Model verification

### Calibration

### Sensitivity analysis

### Uncertainty analysis

### Costs and benefits of agent-based models

## Modeling with FRED

FRED is not a model. Rather, FRED is a language and a platform for building models. The model comes from you, the FRED user, the modeler. It should be recognized that building a model is a challenging activity requiring a lot of effort on the modeler’s part. FRED helps the process by making it easier to define and build a model, by efficiently executing simulation runs of the user’s model, and by providing numerous ways to collect data and to visualize the results of the model. But the ultimate quality of the results depends on the success of the modeler in building an appropriate model for purpose at hand, and in communicating the model to FRED for execution. Having said that, we believe that using FRED could save the modeler significant effort by eliminating the need to develop custom simulation software, and that interacting with the FRED user community may also contribute to the user’s successful modeling efforts.

Building a serious model is almost always an iterative process. Working toward building a model with FRED usually includes the following steps:

1. Decide if FRED is suited to your research problem
2. Create a conceptual model
3. Create rules for individuals
4. Create and run a FRED Model
5. Analyze model output and test against other known data
6. Revise model and repeat

## Design Goals of FRED

Why did we decide to create FRED? There are many alternatives to building an agent-based model. Any large software system is designed with specific objectives in mind. This Chapter describes the design goals of FRED. These goals have had a significant impact on why FRED does things the way it does. Understanding the goals helps a potential user of FRED to assess whether the tool is a good fit to the user’s problem.

### Efficiency with Large Populations

FRED is designed to simulate interactions among individuals in a large population. The early applications of FRED were in the field of modeling infectious disease epidemics in specific areas, such as U.S. cities, counties or states. To do so, we needed a system that could represents millions of individuals and their potential interactions. If your problem concerns a small number of individuals, then FRED is probably not the right tool to use.

### Realistic Population Models

Because we were motivated to provide decision makers with simulations of epidemics in particular locations, FRED was designed take full advantage of some existing synthetic populations that include both individuals and their mixing groups, including households, neighborhoods, schools, and workplaces. An extensive synthetic population for the United States was developed independently of FRED by RTI Inc., and FRED was initially designed to read this database directly. FRED has been generalized to allow it to read in any suitably formatted population database. FRED does not require all the data fields that are available in the RTI database. At a minimum, FRED requires a list of individuals and a reporting of individuals to geo-located households. However, much of the power of FRED lies in its ability to model interactions among individuals that occur in realistic social groupings, so it is best suited for applications where such data is available.

### Track Conditions at Population Level

The goal of FRED is to track the spread of Conditions within a population. The output includes summary statistics of how many people are in a given state during each day of the simulation. The output can also include Reports showing the location of these individuals and movies that help visualize the spread of Conditions over time and space.

Although FRED’s primary focus is on patterns within a population, FRED can also provide data on specific individual agents in the simulation. FRED can optionally record a kind of *electronic health record* for each individual that includes every change to that individual’s state over time. FRED offers a limited set of tools for searching this EHR dataset, or the user can use it as input to an external analysis tool. In summary, FRED can produce Big Data but FRED itself was not designed to analyze Big Data.

### Limited Active Set of Individuals

FRED works most efficiently if there is a limited set of actively changing individuals. The internal bookkeeping methods in FRED attempt to limit as much computation as possible to the active set of individuals and those individuals who may directly interact with the active set. Other agents are updated on a strictly as-needed basis. For models in which a small subset of a large population is in the active set, FRED can produce dramatic speedups compared to simpler approaches that update each individual agent on each time step. For models in which most individuals may change on each time steps, FRED will still work well, but its performance will reflect that additional computation required.

### Extensibility

FRED permits the modelers to design models that are quite complex. FRED models can track any number of Conditions spreading through a population. The characteristics of each Condition are defined by the user. The evolution of each agent’s state may depend on a wide range of factors, including the agent’s demographics, their social interactions, and the state of other agents in the model. All of these features are defined by the user of FRED, with no software changes required to the core platform.

## Suitability of FRED

There are many tools for building models, including many tools for building agent-based models. Before adopting a tool, users should decide which tool is suited well to their needs. The design goals above provide kinds of a checklist to compare against the needs of a model.

* FRED is well-suited for
  + Large-population models
  + Population-level phenomena arising from individual interactions
  + Phenomena with a straightforward "Natural History"
  + Phenomena with strong social determinants
  + Phenomena with strong spatial characteristics
  + Problems dependent on longitudinal demographic projections
* FRED is less well-suited for
  + Detailed physiological modeling of individuals
  + Problems depending on high-resolution location (e.g. traffic models)

## FRED Versions

This document describes FRED Version 6.

FRED uses a three-part version number (e.g. Version 5.3.0):

* The first number is the major version number, and changes when extensive changes are made to the syntax of the FRED language. Models written for version before the current major version may require manual editing to be accepted in the new FRED syntax.
* The second number is the minor version number and is incremented when new features are added to FRED. Models written for previous minor versions of FRED should still run as expected, but the results may be slightly different during to changes in random number sequences.
* The third number is updated after bug fixes and improvements in stability and/or efficiency. Only models affected by the relevant bugs should be affected by the incremental update.

### Revision Notes

Version 6.5.0:

* Added Reseeding

Version 6.4.0:

* Added date labels to fred\_plot

Version 6.3.7:

* Updates to fred\_slurm\_job

Versions 6.3.4-6.3.6:

* Minor bug fixes

Version 6.3.3:

* Added mass gathering example model: **concert**

Version 6.3.2:

* Support for condition-specific density\_contact\_prob property

Version 6.3.1:

* Added logscale to fred\_plot

Version 6.3.0:

* New calibration methods:
  + fred\_calibrate\_contacts
  + fred\_calibrate\_density
  + fred\_calibrate\_R0

Version 6.2.8:

* Added max\_size property to place types.

Version 6.2.7:

* added fred\_installation\_test
* Added place-specific density\_contact\_prob property.

Version 6.2.6:

* Added place-type specific density transmission.

Version 6.2.5:

* Added set\_state(Cond, State) action.

Version 6.2.4:

* Reduced warning messages about duplicate property settings.

Version 6.2.3:

* Use **my** and **my\_list** to declare personal variables.

Version 6.2.2:

* Improved error checking in Parser

Version 6.2.1:

* Updated example models and documentation

Version 6.2.0:

* Admin agents can set contact factors for individual places.
* Admin agents start in the admin\_start\_state (Excluded by default).
* Accept lower case keywords.
* **Start** and **Excluded** included as pseudostates in all conditions.

Version 6.1.0:

* Support for code blocks for Conditions, Places, and Networks

Version 6.0:

* Support for structured FRED code.

Version 5.18.4:

* It is now a compiler error to omit the wait rule for any state.
* Compiler warnings are issued for state with no transition rules and no default rule. In this case, a self-transition back to the same state is assumed.

Version 5.18.3:

* Generalize import\_census\_tract() to import\_admin\_code().

Version 5.18.2:

* Updated fred\_plot.

Version 5.18.1:

* Bug fixes in fred\_get\_R0\_coefficients

Version 5.18.0:

* Updated job control scripts fred\_job and added fred\_slurm\_job

Version 5.17.2:

* Fixed bug in [initialization of global variables](#_Initializing_Variables).

Version 5.17.1:

* Fixed bug in [initialization of personal variables](#_Initializing_Variables).

Version 5.17.0:

* Fixed bug in [wait(until\_DATE).](#_Wait_Rules)
* Added [mod operator (%)](#_Expressions) to expressions.
* Added [sim\_run factor](#_Factors_based_on_2).
* All variables can now be [initialized via full expressions](#_Initializing_Variables), rather than with constants only.

Version 5.16.0:

* Produce output files of [time-series values of global variables](#_Time_Series_Output) for plotting with fred\_plot.
* New [date-related factors](#_Factors_based_on_2):
  + epi\_week
  + epi\_year
  + sim\_week
  + sim\_month
  + sim\_year

Version 5.15.2:

* Updated documentation.

Version 5.15.1:

* Fixed bug storing admin id's on the global place list.

Version 5.15.0:

* Added [user-defined place locations in the FRED program](#_Defining_Places_in).
* Adopted uniform [place id coding scheme](#_Defining_Places_in).
* Added automatically generated [global lists of administrators](#_Administrators).

Version 5.14.1-2:

* Fixed bugs in reading user-defined place files.

Version 5.14.0:

* Added min\_weight [factors for selecting edges](#_Factors_based_on) in a network.
* Added an [action](#_Actions_Affecting_an) of the form **join(*PlaceType*, Expression)** that causes the agent to join a specific place named by the Expression.
* Fixed bugs having to do with initialization of adminstrative agents.

Version 5.13.2:

* Fixed bugs in the unique identifier sp\_id for generated places.

Version 5.13.1:

* Removed deprecated support for import files.

Version 5.13.0:

* Added deterministic\_contacts option to [Network transmission](#_Transmission_in_Networks).
* Added health\_records\_run property to limit the production of [health records files](#_Health_Records_File) to a single run by default.

Version 5.12.1:

* Efficiency improvements in proximity transmission.

Version 5.12.0:

* Added [**import\_list()**](#_Import_Actions) action. This exposes a list of agents to a Condition. It may be useful to expose the exact same agents at the start of multiple simulation runs.

Version 5.11.1:

* Fixed bug in group id [factors](#_Factors_based_on_1). The group id is now the sp\_id for the group as it appears in the synthetic population file. This is a unique identifier across all counties in the US synthetic population.

Version 5.11.0:

* Meta-agents can modify transmissibility of Conditions using [**set\_trans()**](#_Import_Actions)
* Added [**day\_of\_year**](#_Factors_based_on_2) and [**transmissibility\_of\_Condition**](#_Factors_based_on_3) factors.
* Added [**sin()** and **cos()**](#_Math_Functions) math functions.

Version 5.10.0:

* Added [**record\_location**](#_Health_Records_File) property. If set, then latitude and longitude are REPORT in the health records file.

Version 5.9.0:

* Added rules for [import actions.](#_Import_Actions)
* Added [initialization of variables](#_Initializing_Variables).
* Added [set(Variable, Expression, AgentId)](#_Actions_that_Change)
* Added [latitude and longitude](#_Factors_based_on_1) Factors.
* Added [distance(lat1, lon1, lat2, lon2)](#_Math_Functions) function.

Version 5.8.3:

* Fixed bug in [**value()**](#_Value_Function) function.
* Changed [place\_id factor](#_Factors_based_on_1) to return index of place in synthetic population file.

Version 5.8.2:

* Improved compiler error messages.

Version 5.8.1:

* Fixed bug with gobal list variables.

Version 5.8.0:

* Added [**is\_import\_agent()**](#_Meta-Agent_Predicates)predicate.

Version 5.7.0:

* Added [global variables](#_Global_Variables).
* Added [value(...)](#_Value_Function) function to access the value of an expression for another agent.
* Enabled **if(PredicateList)** clauses in [Action Rules](#_Action_Rules).
* Removed obsolete code.

Version 5.6.0:

* Support for list-valued Expression1 in edge-weight [Action Rules](#_Actions_that_Change):
  + set\_weight(Network, Expression1, Expression2)

Version 5.5.1:

* Updating the documentation of the [select(...) function](#_Select_Function).

Version 5.5.0:

* Support for list-valued expressions in edge [Action Rules](#_Actions_Affecting_an):
  + add\_edge\_to(Network, ListExpression)
  + add\_edge\_from(Network, ListExpression)
  + delete\_edge\_to(Network, ListExpression)
  + delete\_edge\_to(Network, ListExpression)
* Fixed format of VNA header in network output files.

Version 5.4.3:

* Added a compiler Warning if a state has no Wait Rule.

Version 5.4.2:

* Updated documentation to reflect details of [select(...) function](#_Select_Function).
* Add documentation of [administrative Action Rules](#_Administrative_Actions):
  + close(Group)
  + randomize\_network(Network, Expression1, Expression2)

Version 5.4.1:

* Removed debugging messages.

Version 5.4.0:

* Added property "**enable\_var\_records**". If this is set to 1 and **enable\_health\_records** is also set, then the health\_records file contains a line each time an agent changes the value of a personal variable. This is meant to facilitate debugging models.
* Added the [list-valued expression](#_List_Variables_and) **list(...).**

Version 5.3.0:

* Added [modifiable weights](#_Actions_Affecting_an) to edges in networks.
* Added factors based on [network edge weights](#_Factors_based_on).
* Added [list-valued functions](#_List_Variables_and) **pool(...)** and **filter(...).**
* Changed Action Rules to take effect on zero-duration state transitions to the same state.
* Removed concept of logit transition rules. Logistic regression can now be specified by setting transition probability expressions.
* Added new [Action Rules](#_Actions_that_Change):
  + **set\_list(...)**
  + **set\_state(...)**
  + **set\_sus(...)**
  + **set\_trans(...)**
  + **set\_weight(...)**
* Removed Action Rules:
  + **sus(...)**
  + **trans(...)**
  + **mult\_sus(...)**
  + **mult\_trans(...)**
* Changed "fatal()" to "die()"
* Changed "if enter(...)" to "if state(...)"
* Updated Library and Example models.
* Improved compiler error and warning messages.

Version 5.2:

* Added [List Variables](#_Variables) that can store lists of agent id's.
* Added rules for [adding and deleting edges](#_Actions_Affecting_an).
* Added rules for [setting variables](#_Actions_that_Change).
* Added [select(...)](#_Select_Function) function.
* Fixed bugs when joining or quitting places.
* Improved compiler error and warning messages.

Version 5.1:

* Added [network mixing groups](#_Networks) and added Action Rules to the FRED programming language.

Version 4:

* Generalized FRED to a declarative programming model and user-defined Conditions and interaction groups.

Version 3:

* Developed in 2013-2019 by the Pitt Public Health Dynamics Lab. This version introduced the concepts of general health Conditions and supprted both respiratory and vector transmission.

Version 2:

* Developed in 2010-2013 by the Pitt Public health Dynamics Lab, Carnegie Mellon University and the Pittsburgh Supercomputer Center. FRED v2 supported a variety of respiratory infectious diseases.

Version 1:

* Developed in 2009 by the Pitt Public Health Dynamics Lab, the Pittsburgh Supercomputer Center and RTI, Inc. based on the RTI epidemic simulator by Phil Cooley and Shawn Brown. FRED v1 was a special purpose simulator for pandemic influenza.

# Chapter 2: Synthetic Populations

## Definition

A *Synthetic Population* is a data set that represents each person and household in a given location with geospatial accuracy and contains no personally identifiable information.

## Data sources

FRED uses the synthetic population developed by RTI, International.In summary, RTI used a proportional iterative method developed in (Beckman, et al. 1996) to generate an agent population from the US Census Bureau’s Public Use Microdata files (PUMs) and Census aggregated data. See (Wheaton, et al. 2009) for a detailed description. Each agent had a set of socio-demographic characteristics and daily behaviors that included age, sex, employment status, occupation, and household location and membership.

As described on the web site https://www.rti.org/impact/synthpop:

*“Unlike typical sociodemographic data, the RTI U.S. Synthetic Household Population represents households and persons as dots on a Report—matching high-resolution population distributions with the correct mix of households in each census block group.”*

## Process

The RTI synthetic population is based on the U.S. Census Bureau’s Public Use Microdata files (PUMS) and aggregated data from the 2005-2009 American Community Survey (ACS) 5-year sample ([Wheaton, 2012](#_ENREF_36)). This open access database comprises a spatially accurate model of all households, schools, workplaces and group quarters (e.g. prisons, college dorms, military bases and nursing homes) in the United States. Individual agents are defined and assigned to each household, school, and workplace in the database so that the result closely matches the census-based spatial distributions of households and population sizes at the census block group level, as well as commuting patterns across census-tract boundaries. For ABMs that model specific geographic regions in the U.S., this synthetic population provides an excellent source of spatially accurate population information. We have also extended this approach to selected international locations.

## Projections of Future Demographics

Agent-based models (ABMs) are flexible and powerful tools for predicting the effects of infectious and chronic diseases, and for modeling in a host of other widely varied disciplines. Realistic long-term populations are necessary for many simulations, such as development of chronic disease and effects of long-term behavior. FRED includes methods to reproduce external population size estimates. For example, we used population projections created by the Pennsylvania State Data Center (PASDC) to model long term demographic changes for Pennsylvania from 2010 to 2030 in FRED.

To achieve these projections, FRED inputs age-specific mortality and maternity rates and methods for changing the activity profiles for individuals over time, including entering and leaving school or the workplace when appropriate. To achieve the population target numbers from the PASDC, agents are added to or removed from the population on a yearly basis. After a yearly population size update is completed, households are rebalanced, by applying household swapping rules to maintain local population density. New agents are assigned to schools as appropriate to their ages and to workplaces based on location and vacancies. Migration flows are modeled using county to county migration data obtained from the census American Community Survey. Agents are chosen to migrate as complete households when possible, to mimic real migration.

Over the 30-year period of the simulation, FRED closely matches the projections of total Pennsylvania population size at the 5-year intervals for which projections are made available and for the intervening years. FRED matches projections of PA population size by age and maintains consistent household size distributions over time. Agents have continuity of experience through life stages and take on appropriate social roles and activities over their lifetimes. FRED generally maintains consistent school size distributions and workplace size distributions over time. Over the course of the simulation, the proportion of agents who are native to the population is consistent with US Census data for 2010. FRED agents create new households at rates comparable to reported rates.

These methods are generalizable to create realistic populations for any projection and any US location.

# Chapter 3: Declarative Programming in FRED

Declarative programming is a programming paradigm that expresses the logic of the computation without expressing the control flow. (Lloyd, J.W., *Practical Advantages of Declarative Programming*). Within the general class of declarative programming languages, FRED can be classified as a domain-specific language whose domain is Agent-Based Modeling (ABM). The FRED language consists of two primary kinds of statements: *property statement*s and *rule statements*. Property statements define the static features of the model, including the location, the range of dates for the simulation, all concepts of interest, what information should be tracked and reported, and the initial conditions. Rule statements define the dynamic features of the model, including how agents and their environment change state over time. The FRED platform processes the FRED program, sets up the population, applies the initial conditions, simulates the activities and interactions of the agents, and tracks all user-defined conditions within the population. FRED outputs several reports, charts, and visualizations.

The declarative programming style of FRED provides several advantages for agent-based modeling:

* No traditional computer programming experience is required.
* You can focus on scientific effort (e.g. data collection, conceptual modeling, experimental design).
* FRED provides a simple workflow environment for you and manages all the data produced by the simulation and associated metadata.

## A Minimal FRED Program

We generally use the terms **FRED model** and **FRED program** interchangebly. The required elements of a FRED model include the Simulation Location, the Simulation Time Frame, and at least one Condition. A minimal FRED model is shown below:

**# Simulated Location: a small county in Pennsylvania**

**locations = Jefferson\_County\_PA**

**# Simulated Time Frame**

**start\_date = 2020-Jan-01**

**end\_date = 2020-Jan-10**

**# Condition:**

**condition ACTIVE {}**

When this model is run, the population of Jefferson County, PA, comprising 45,318 individuals, is loaded into the simulation and individual agents are assigned to households, neighborhoods, schools and workplaces. The model simulated the agents as they follow their normal activities for a period of 10 days. Every condition includes two pseudostates called *Start* and *Excluded*. All individual agents are assigned to the Start pseudostate in ACTIVE condition. The model declares no transition rules, so by default all agents transition from *Start* to the next state in the condition, which in this case is *Excluded.* All the agents then remain *Excluded*. Clearly, nothing of interest happens during this model.

## Running a FRED Program

It is conventional to use the **.fred** suffix on FRED model files, but it is not required. Suppose that the model above is stored in a file called **minimal.fred.**  Then the command line to execute the model is

**% FRED -p minimal.fred**

Note: Users do not usually invoke the FRED command directly in this way. Instead, command line users run one of the workflow commands such as **fred\_job**, described in Chapter 10. Web users define and run their FRED models via the FRED Web Interface.

The output of the **minimal**.**fred** model includes a spreadsheet that shows the counts of how many agents are in each state of the model on each day:

A screenshot of a cell phone

Description automatically generated

The first three columns give the day counter, the date and the epi-week for each simulation day. This is followed by three columns for each of the states. For example, the *Start* state has columns called ACTIVE.newStart, ACTIVE.Start, and ACTIVE.totStart. The newStart column contains the number of agents who enter that state during the day. The Start column gives the number of agenst in that state and the end of the day. The totStart columns is a running total of all agents who have ever visited that state. The final column ACTIVE.RR gives the *reproductive rate,* or transmission rate for this condition, which is not applicable in the model.

## The Structure of a FRED Program

The syntactic features of the FRED modeling language are:

* FRED program are plain text files.
* Comments are introduced by the # character. All characters after a # are ignored until the end-of-line.
* Lines consisting of only white space are ignored.
* Statements are terminated by an end of line. If the final character of a line is ‘\’, the statement is continued on the next line.
* Multiple statements may occur on the same line, separated by semi-colons.
* Statements include the following types:
  + Use/Include statements
  + Property definition statements
  + Rules
* Property definition statements occurring later in the program override any previous definitions for the same property.
* FRED processes the program file before executing the simulation.
* FRED reports errors if any statements are ill-formed and warnings if statements refer to conditions, places or variables that have not be declared in the program.
* If there are no reported errors, FRED executes the model and reports its results.

## Use Statements

**use FRED::*Model***

This statement includes one of the built-in model into the user program. A program can use multiple built-in models.

**Note:** The models in the FRED models library each define the properties and rules needed for that specific model. It is possible that two models may conflict with one another, so you are advised to read each model before using it in your FRED program.

## Include Statements

**include *filename***

The effect of this statement is to include the specified file at this point in the FRED program. This is useful if the user wants to include one or more sub-models from the user's own files.

## Property Statements

Property statements have the form:

***Property* = *Value***

where ***Property*** is one of the properties defined in later Chapters. If ***Value*** consists of a list, elements of the list are separated by spaces.

Unless noted otherwise, if multiple statements define the same property, the property definition occurring latest in the program will be used to set the value of the property.

## Condition and States

FRED keeps track for each agent of that agent’s current State in each Condition defined in the model. For each State, rules control how agents change during a FRED simulation. There are three categories of rules in FRED: Action Rules, Wait Rules, and Transition Rules. Action Rules control how the agent’s own state or own variable change when the agent enters a state. Wait Rules control how long an agent stays in a given state. Transition Rules control how an agent moves from one state to another.

## A Note for Previous FRED Users

The FRED language has undergone rapid evolution in the past year, and previous users will note that the version 6 has adopted a cleaner and, hopefully, clearer programming style. This has been accomplished mainly though the FRED parsing process which turns the new langauge syntax into lower level statements.

The suggested overall structure of a FRED program is

# Global property statements

start\_date = ...

end\_date = ...

locations = ...

# variable declarations and initializations

my x ...

global\_var Y ...

...

# condition and state definitions

condition Cond1 { ... }

state Cond1.State1 { ... }

state Cond1.State2 { ... }

...

condition Cond2 { ... }

state Cond2.State1 { ... }

When the preprocessor encounters a condition declaration, it translates the properties within the brackets to individual property statements as follows:

condition COND { property = value }

is translated to

COND.property = value

Any number of properties can appear within brackets, separated by semi-colons or newlines. For example, the code block:

condition INF {

transmission\_mode = proximity

exposed\_state = E

states = S E I R

}

is equivalent to

Condition INF

INF.transmission\_mode = proximity

INF.exposed\_state = E

INF.states = SEIR

Property statement in brackets are cumulative, so that the above set of properties could also be expressed as:

condition INF { transmission\_mode = proximity }

condition INF { exposed\_state = E; states = S E I R }

You can even mix and match the bracket statements above with the dot notation, e.g.,

INF.transmissibility = 1.0

add the transmissibility to the INF condition definition, whereever it appears in the FRED program.

These same rules apply to Places and Networks, e.g.,

place Household { contacts = 0.234 }

is equivalent to

Place Household

Household.contacts = 0.234

For rules, the FRED preprocessor translates code block to individual rules, as in:

state INF.E {

set\_sus(0)

if (age < 20) then wait(3)

if (20 <= age) then wait(4)

next(Is) with prob(0.67)

default(Ia)

}

is equivalent to

if state(INF,E) then set\_sus(0)

if state(INF,E) and(age < 20) then set wait(3)

if state(INF,E) and (20 <= age) then set next(Is) with prob(0.67)

if state(INF,E) then set default(Ia)

As before later code block can add to the rules for a given state, using either the bracket or the dot notation.

Finally, the FRED preprocessor accepts initial upper or lower case for the keywords **condition, place, network,** and **state.** Lower case is preferred for easing of typing and is adopted in this document.

# Chapter 4: Agents and Their Properties

FRED is a framework for agent-based modeling. There are two kinds of agents in FRED: ordinary agents and meta-agents.

An ordinary agent represents an individual person. The ordinary agents in FRED represent individual people. We will refer to ordinary agents as either individuals or simply, agents. Agents have demographic characteristics including age, sex, and race. Agents interact with other agents in *mixing groups*, such as households, neighborhoods, schools, and workplaces. Some mixing groups are built into FRED and others can be defined by the user.

A meta-agent represents an abstract agency that can affect the course of the model. For example, the user can define a meta-agent that represents the school administration for a school district, and this agent can be responsible for making decisions such as when to close the schools in an emergency. Other meta-agents can start disease outbreaks by infecting selected individuals from a source of infection that is not modeled explicitly in the simulation.

For both meta-agents and ordinary agents, their behavior is controlled by defining conditions, states and rules.

## Conditions

Agents in FRED have a rich set of properties that can be used to influence the future states and activities of the agent itself and well as other agents. Like the interaction groups, some properties of agents are built into FRED and others can be defined by the user. The reminder of this chapter describes the properties of agents.

To build a model in FRED, you first need to decide what **Conditions** you want to track within the population. Conditions might include diseases, economic Conditions such as poverty, or behaviors such as drug use or vaccine uptake. Models must include at least one Condition but can include as many Conditions as needed. Conditions consists of **states** and **rules**. Some rules and recommendations:

* Each agent is in exactly one state for each Condition.
* By convention, we usually use all caps for Condition names and initial caps for state names, for example, the INFLUENZA Condition might have states called Susceptible, Infectious, and Recovered.
* Each Condition must have a unique name, but state names may appear in more than one Condition. The “full name” of a state includes its Condition name, and is written CONDITION.State, for example: INFLUENZA.Susceptible
* Condition and State names must contain only alphanumeric characters with no blanks, hyphens, underscores or other punctuation or special characters.
* Each Condition include two special *pseudostates* by default: **Start** and **Excluded**.
* All ordinary agents begin in the **Start** pseudostate. Think of **Start** as the starting-line for a race, in which some runners are assigned to different starting positions. **Start** usually contains rules that immediately send each agent to an appropriate initial state. Meta-agents begin in other states by default, as explained later. If no rules are provided for **Start**, then all ordinary agent proceed immediately to the first actual State defined in the Condition, or to **Excluded** if no actual States are defined.
* The **Excluded** pseudostate is for any agents for which the Condition does not apply. For example, if there is a Condition called PREGNANCY, the the **Start** state should include a rule that transitions all males from **Start** to **Excluded**.

You declare a Condition by the Condition block that gives the properties of the Condition. For example, if your model is focuses on the interact of influenza and pregnancy, you might declare:

**condition INFLUENZA { …}**

**condition PREGNANCY { … }**

The block of code within the brackets define the properties of the Condition, as shown below.

## States

Each Condition includes one or more states. The states of a Condition are declared in the condition block with a property statement of the form:

**states = *State1 … StateN***

As an example, suppose the model includes the level of symptoms that a person has. The model might be defined as follows:

**condition SYMPTOMS {**

**states = None Mild Moderate Severe LifeThreatening**

**}**

Note that it is not necessary to declare the **Start** and **Exclude** pseudostates. State names do not need to be unique across Conditions. FRED uses the notation **Cond.State** to refer to the state **State** within the Condition **Cond**.

For each Condition in the model, each agent is always in exactly one of the states defined for that Condition. We will see in later sections how to set the initial state for each agent, and how to define rules that dertermine how agents change states over time.

Like Condition names, State names may contain any alphanumeric character, but no other characters are allowed.

## Meta-Agents

FRED meta-agents represent other agencies that may affect the model. There are two kinds of meta-agents in FRED: **administrators** and the **Import Agent.** The following sections provide further details on meta-agents.

### The Import Agent

The **Import Agent** is responsible for exposing individual agents to specific Conditions due to some cause that is not explicitly represented within the FRED model. For example, one way that an outbreak of a disease can begin is that an individual might contract the disease while traveling abroad or an unmodeled visitor infects someone in the population. FRED achieves such effects through the **Import Agent**, a meta-agent that can alter the state of ordinary individuals in population in the model.

The Import Agent starts in the specified **import\_start\_state** for each Condition, and then changes states via transition rules, just like ordinary agents. If not import\_start\_state is specified for a Condition, the meta-agent is not active for that Condition. If the Import Agent executes a rule that has special actions described below, it can cause the exposure of ordinary agents. When this happens, we say that an "imported case" of the Condition has occurred.

### Administrators

An **administrator** is a meta-agent associated with a mixing group (i.e. a place or a network). Each mixing group has a normal schedule of operations that defined that days and time that agents can interact within that group. The administrator can override the normal schedule by closing the given mixing group based on the current situation in the model. Examples would be school administrators who decide when to close a school in the case of a health emergency, hospital administrators who may decide to open a new community health center, or law enforcement agencies that may intervene to shut down an illegal drug market. Adminsitrators can also control certain aspects of how agents behave within their mixing group.

The user declares the use of Administrators for a mixing group by the property statement:

***GroupType*.has\_administrator = 1**

For example, if a model includes school closures controlled by administartors, the program should include:

**School.has\_administrator = 1**

The effect is that FRED will generate a meta-agent administrator (or admin agent) for each school in the model.

The admin agent can close its associated place by entering a state with the action:

**close(*GroupType*)**

For an example of school closures by admin agents, see Chapter 19.

The admin agent can also change the number of contacts per hour that occur within its associated place. This is accomplished by executing the action:

**set\_contacts(*Expression*)**

Contacts are discussed in detail in Chapter 8.

If a Place Type has an administrator, the FRED automatically generates a global list variable called ***PlaceType*List**. This is assigned a list of the administrators for the places in the PlaceType, in the order in which individual places are generated. For example, suppose the user defines a new PlaceType called **Pharmacy**, and generates two pharmacies as follows:

**place Pharmacy {**

**has\_administrator = 1**

**add = 942003001 40.451164 -79.999803 230.1**

**add = 942003002 40.626449 -79.723195 240.5**

**}**

Then a global list variable called **PharmacyList** is created automatically and will contain the agent id's for the two pharmacies generated above (See Chapter 7).

### Meta-Agents and State Spaces

There is usually no reason to have individual agents and meta-agents in the same state. While FRED does not force the separation between states for individual agents and states for meta-agents, it is recommended that the modeler partition the states for a given Condition so that the states reached by individual agents and meta-agents do not overlap.

## Variables

There are two kinds of variables in FRED: ordinary variables and list variables. Ordinary variables take on single numerical values and list variables contains lists of values. All values are real numbers.

### Global Variables

Global variables are variables that are effectively shared by all agents. Any agent can change the value of any global variable, and the new value is accessible to all agents. Global variables are declared by statements of the form:

**global *VarName1 ... VarNameN***

**global\_list *ListVarName1 ... ListVarNameN***

### Personal Variables

Agents may include any number of personal variables. Each agent has its own instance of each personal variable. Personal variables are declared with statements of the form:

**my *VarName1 ... VarNameN***

For example, the following statement declares that each agent with have three personal variables called **temp, bmi,** and **bp.** These variables may change on an individual level, and affect the agent's state in any condition.

**my temp bmi bp**

Personal list variables are declared with statement of the form:

**my\_list *ListVarName1 ... ListVarNameN***

### Initializing Variables

Ordinary global and personal variables can be initialized by statements of the form:

***Variable* = *Expression***

For example, if the program includes the following statements,

**my temperature**

**temperature = 98.6**

then each agent will have its own version of the **temperature** variable, and they will all have the initial value **98.6.**

If the variable is initialized via an expression, the expression is evaluated for each agent. For example,

**temperature = 98.6 \* normal**

will asign to each agent a distinct temperature drawn from a normal distribution with mean 98.6 and standard deviation of 1.0.

Suppose we want to have another variable equal to the agent's age squared. We could initialize this variable as follows:

**my ageSquared**

**ageSquared = age\*age**

Since the initialization expression is evaluated separately for each agent, the variable will be set to each age's own age squared.

The program can also initialize global variables:

**global isRaining**

**isRaining = 1**

In this case, there is a single global variable called **isRaining** and it had the initial value of 1.

List variables can be initialized to a list of values by using a list-valued expression, for example:

**global\_list listOfPrimes**

**listOfPrimes = list(1, 2, 3, 5, 7, 11, 13, 17, 19)**

**my\_list FeatureList**

**FeatureList = list(id, age, sex, race)**

In the latter example, each agent has a FeatureList list variable that is initialized to that agent’s id number (starting with 0), age (as an integer), sex (0 = M, 1 = F), and race code as defined in synthetic population.

Some notes:

* Variable initialization statements may appear anywhere in the program, but it is good practice to place the initialization statement near the top of the program for readability.
* If there are multiple initialization statement for a given variable, the last occurring statement takes effect.
* Initialization of global variables takes place before any agent is created, and so the initial values are available when initializing personal variable for agents.
* Initialization of personal variables takes place before any transition rules are applied.
* If no explicit initial value is given for a varibale, then the variable is initialized to 0.
* If no explicit initial value is given for a list variable, then the list variable is initialized to the empty list.

## Factors

Factors comprise the universe of facts that the agents might "know" about the world, including the current time and date, the demographic properties of the agent, the current and previous states of the agent, the states of others in the agent's social groups, and facts based on the agent's places and networks. The value of a given factor generally varies from agent to agent and often varies over time for a single agent. As described in Chapter 5, any of these factors can appear in rules that determine both the initial state of an agent and how an agent changes state. Factors include:

### Factors based on the simulation run

* **sim\_run** – the simulation run number, starting with 1.

### Factors based on time and dates

* **sim\_day** – the simulation day, starting with day 0.
* **sim\_week** -- the simulation day divided by 7, truncated to an integer
* **sim\_mon** -- the simulation day divided by 30, truncated to an integer
* **sim\_year** -- the simulation day divided by 365, truncated to an integer
* **epi\_week** -- the epidemic week of the year, with possible values 1..53
* **epi\_year** -- the 4-digit epidemic year, e.g., 2020
* **day\_of\_week** – day of the week, coded: Sun, Mon, ,,, Sat
* **day\_of\_month** – day of the month, with values 1..31
* **day\_of\_year** -- index of day of year, with Jan-01 having value 1 and Dec-31 having value 365 (in non-leap year).
* **month** – integer representing the month where Jan=1, Feb=2, ..., Dec=12
* **year** – four digit year
* **date** – YYYY-MMM-DD, eg, 2020-Jan-05
* **hour** -- 0..23

### Factors based on the agent's demographics

* **id** -- the agent's unique integer id
* **birth\_year** -- the year of the agent's birth
* **age\_in\_days** – integer number of days
* **age\_in\_weeks** – integer: age\_in\_days / 7
* **age\_in\_months** – integer: age\_in\_days / 30
* **age\_in\_years** – real value, e.g. 33.3
* **age** – integer part of age\_in\_years, e.g. 33
* **sex** – male or female
* **race** – as defined by the synthetic population
* **profile** – as defined by the synthetic population
* **household\_relationship** – as defined by the synthetic population
* **number\_of\_children** – integer number of births during the simulation

### Factors based on agent's own state

* **current\_state\_in\_Condition** - integer index of current State
* **time\_since\_entering\_Condition.State** – in hours
* **susceptibility\_to\_Condition** – real value >= 0.0
* **transmissibility\_for\_Condition** – real value >= 0.0
* **transmissibility\_of\_Condition** – same as **transmissibility\_for\_Condition** for ordinary agents. For meta-agents, the transmissibility of the named Condition.
* **transmissions\_of\_Condition** -- number of transmissions by the agent
* ***Var*** – current value of the variable named *Var*. Note: Var may either be a global variable or a personal variable.
* **list\_size\_of\_ListVar** -- number of elements in the given List Variable. Note: ListVar may either be a global variable or a personal variable.
* **id\_of\_source\_of\_Condition** -- the id of the agent who transmitted the Condition to this agent

### Factors based on other agents

* **incidence\_count\_of\_Condition.State[\_in\_GroupType][\_excluding\_me]**
* **incidence\_percent\_of\_Condition.State[\_in\_GroupType][\_excluding\_me]**
* **current\_count\_of\_Condition.State[\_in\_GroupType][\_excluding\_me]**
* **current\_percent\_of\_Condition.State[\_in\_GroupType][\_excluding\_me]**
* **total\_count\_of\_Condition.State[\_in\_GroupType][\_excluding\_me]**
* **total\_percent\_of\_Condition.State[\_in\_GroupType][\_excluding\_me**]
* **sum\_of\_Variable\_in\_GroupType** -- sum of the personal Variables of member in the agent's GroupType
* **ave\_of\_Variable\_in\_GroupType** -- ave of the personal Variables of member in the agent's GroupType

### Factors based on the agent's groups

* **GroupType** -- the group sp\_id as provided in the synthetic population of the agent's mixing GroupType group, or -1 if no such group exists
* **admin\_of\_GroupType** -- the agent id of the administrative agent associated with the agent's GroupType group or -1 if no sich group exists
* **income\_of\_GroupType** -- mean household income of members of the agent's GroupType group
* **size\_of\_GroupType** -- number of members of the agent's GroupType group
* **elevation\_of\_PlaceType** -- meters above sea level
* **adi\_national\_rank\_of\_PlaceType** -- National ADR rank of block group containg agent's PlaceType (1..100)
* **adi\_state\_rank\_of\_PlaceType** -- State ADR ranking of block group containing agent's PlaceType (1..10)
* **block\_group\_of\_PlaceType** -- FIPS code
* **census\_tract\_of\_PlaceType** -- FIPS code
* **county\_of\_PlaceType** -- FIPS code
* **state\_of\_PlaceType** -- FIPS code
* **latitude\_of\_PlaceType** -- latitude of agent's place
* **longitude\_of\_PlaceType** -- longitude of agent's place

### Factors based on the agent's networks

* **in\_degree\_of\_*Network*** -- agent's in-degree in given network
* **out\_degree\_of\_*Network*** -- agent's out-degree in given network
* **degree\_of\_*Network*** -- agent's degree in given undirected network
* **id\_of\_last\_inward\_edge\_in\_*Network*** -- id of last agent connected to the agent
* **id\_of\_last\_outward\_edge\_in\_*Network*** -- id of last agent connected from the agent
* **id\_of\_max\_weight\_inward\_edge\_in\_*Network*** -- id of agent with the maximum incoming edge weight to the agent
* **id\_of\_max\_weight\_outward\_edge\_in\_*Network***-- id of agent with the maximum outward edge weight from the agent
* **id\_of\_min\_weight\_inward\_edge\_in\_*Network*** -- id of agent with the minimum incoming edge weight to the agent
* **id\_of\_min\_weight\_outward\_edge\_in\_*Network***-- id of agent with the minimum outward edge weight from the agent

## Expressions

FRED program statement can include numerical expressions that are defined over an agent’s Factors. Expression are defined by the following formal grammar:

***Expression* = *simple\_factor* | real number | *func1*(Expression1) | *func2*(Expression1,Expression2)**

***func1* = log | exp | abs | sin | cos**

***func2* = add | sub | mult | div | mod | dist | equal | min | max | uniform | normal | lognormal | select | value**

***simple\_factor* = *sex* | *household\_relationship* | *race* | *profile* | *day\_of\_week* |  *month***

***sex* = male | female**

***household\_relationship* = householder | spouse | child | sibling | parent | grandchild | in\_law | other\_relative | boarder | housemate | partner | foster\_child | other\_non\_relative | institutionalized\_group\_quarters\_pop | noninstitutionalized\_group\_quarters\_pop**

***race* = unknown\_race | white | african\_american | american\_indian | alaska\_native| tribal | asian | hawaiian\_native | other\_race | multiple\_race**

***profile* = infant | preschool | student | teacher | worker | weekend\_worker | unemployed | retired | prisoner | college\_student| military | nursing\_home\_resident**

***day\_of\_week* = Sun | Mon | Tue | Wed | Thu | Fri | Sat**

***month* = Jan | Feb | Mar | Apr | May | Jun | Jul | Aug | Sep | Oct | Nov | Dec**

Notes:

1. The numerical operators + -, \*, /, and % are stated in infix notation, where A+B is a synonym for add(A,B), etc.
2. The **div** function returns 0 if Expr2 == 0, and the **log** returns 1e-100 if Expr <= 0.0.
3. The **household\_relationship** factor is defined as the relations between the person and the householder. The values are derived from the census data used in the Public Use Microdata to create the synthetic population (U.S. only).
4. The **race** factor values are derived from the census data used in the Public Use Microdata to create the synthetic population (U.S. only).
5. The **profile** factor values are based on FRED’s built-in assignment of activity profiles. An agent’s profile may change over time, based on the agent’s age and stage of life (see Ages and Stages).

Expressions are evaluated with respect to a specific agent. For example, if the expression

**age \* 2.5**

occurs in a rule, it evaluates to 2.5 time the age of the agent evaluating that rule.

Expression can be arbitraily complex, such as

**age \* 2.5 + income\_of\_Household / 5.0**

As shown above, white space is permitted within an expression.

### List Variables and List-Valued Expressions

FRED includes List Variables, which are agent-specific lists of values. List Variables are useful for storing lists of identifiers of other agents, for example, a list of the agent's friends.

List-valued Expressions are expressions that evaluate to lists of numerical values. They are useful for manipulating List Variables.

List-valued Expression are defined by the following formal grammar:

***ListExpression* = list(*ExpressionList*) | pool(*GroupTypeList*) | filter(*ListExpression,PredicateList*)**

**ExpressionList = *Expression* | *Expression,ExpressionList***

**GroupTypeList = *GroupType* | *GroupType,GroupTypeList***

**PredicateList = *Predicate* | *Predicate,PredicateList***

The functions that return List Values include:

* **list(ExpressionList):** returns a list of values returned by the list of Expressions. For example, if the agent's id is 12345 and its age is 42, then

**list(id, age)**

returns the list (12345, 42).

* **pool(GroupType\_List):** returns a list of all agent id's who share one or more of the Groups with the agent. For example,

**pool(Household, Classroom)**

returns a list of all the agents who are in the agent's Household or in the agent's Classroom, or both.

* **filter(ListExpression,PredicateList):**  returns a list of all agent id's that are in the List Expression who also satisfy all the Predicates in the Predicates List. For example,

**filter(pool(Classroom), age\_in\_years < other:age\_in\_years, sex==other:sex)**

returns a list of all the agents who are in the agent's Classroom and have the same sex as the agent and are older than that the agent. In this expression, the factor **sex** refers to the sex of the agent evaluating the expression, and the factor **other:sex** refers to the sex of the agent being evaluated by the filter. Any of the factors listed in Chapter 4 can be prepended with **other:** to refer to the agent being evaluated by the filter.

### Functions within Expressions

#### Math Functions

FRED supports a number of math functions, including:

* **log(arg)** -- logarithm using base e
* **exp(arg)** -- exponential function using base e
* **abs(arg)** -- absolute value
* **sin(arg)** -- sine function; arg interpreted as radians.
* **cos(arg)** -- sine function; arg interpreted as radians.
* **distance(lat1, lon1, lat2, lon2)** -- returns the distance in km between the points <lat1,lon1> and <lat2,lon2>.
* **dist(arg1, arg2)** -- returns the distance in km between the places whose id's are represented by the arguments
* **equal(arg1, arg2)** -- returns 1 if the arguments are equal, 0 otherwise
* **min(arg1, arg2)** -- returns the smallest of the two values
* **max(arg1, arg2)** -- returns the largest of the two values
* **uniform(lower, upper)** -- returns a uniform random number in range [lower, upper)
* **normal(mean, stdev)** -- returns a draw from a normal distributions with given mean and standard deviation
* **lognormal(median, dispersion)** -- returns a draw from a lognormal distribution with given median and dispersion

#### Select Function

The following function returns a selected item from a list:

* **select(*ListExpression*, *Expression*)**
* If the Expression does not have the form **pref(...),** then it is evaluated and the resulting value is treated as an index that points to an member of the List. For example, if the ListExpression contains the list (10, 20, 30, 40) then the value of function

**select(ListExpression, 2)**

would be 30 (since the first element in the list has index of 0).

* If the second argument has the form **pref(),** the expression is interpreted as a **preference function** consisting of a list of expressions that will be evaluated for each candidate in the ListVariable, resulting in a **preference score** for each eligible candidate. Positive values for the expression make the candidate more likely to be selected and negative values for the expression make the candidate less likely to be selected. If the expression evaluates to zero it makes no contribution to the candidate's preference score.

Let's suppose that an agent has constructed a ListVariable called *friendCandidates* of potential friends, and that the agent prefers to select friends who are older than the agent. We could use a selection function like:

**select(friendCandidates, pref(other:age - age))**

The formula for computing the overall preference score for a given candidate is:

**preference\_score(candidate) = (1 +  (positive-valued expressions)) /**

**(1 + | (negative-valued expressions)| )**

In the example, candidates who are the same age as the agent will get a preference score of 1, candidates that are older than the given agent will get preferences scores greater than 1 and candidates younger than the agent will have preference score less than 1.

As a more complex example, suppose the agent prefers freinds of similar age and also prefers friends who attend the same school. The following function specifies these two preferences, with a greater weight attached to the same-school term:

**select(friendCandidates, pref(-2\*abs(other:age-age)), 14\*equal(place\_id\_of\_School,other:place\_id\_of\_School)))**

The first preference expression penalizes candidates based on the difference between their age and the agent's own age. The second expression contributes a positive score to candidates who attend the same school as the selecting agent. The factor **place\_id\_of\_School** returns the numerical id for the agent's place and the factor **other:place\_id\_of\_School** returns the id number for the candidate's School. The **equal** function returns 1 if its arguments are equal and 0 otherwise (also known as the Kronecker delta function). The resulting preference score for a candidate would be:

**preference\_score(candidate) = 15 / (1 + 2\*(difference in age))** for candidates in the agent's school

**= 1 / (1 + 2\*(difference in age))** for candidates not in the agent's school

Suppose there were two candidates in the agent's school that differ in age by 0 and 1, respectively, from the agent's own age, and one candidate from another school with age difference of 2 years. Then these candidates would be assigned preference scores of 15, 5, and 0.2, respectively, according to the formula above.

After each candidate's preference score is computed, a probability distribution is defined by dividing each preference score by the total of all preference scores. A person is then selected from the eligible candidates using the normalized preference probabilities.

Continuing the example above, the preference scores sum to 20.2 so they would yield probabilities of 15/20.2 (≈ 0.743), 5/20.2 (≈ 0.247), and 0.2/20.2 (≈ 0.01), respectively, for the three candidates. The selection is then made based on these probabilities.

#### Value Function

The following function allows an agent to access values associated with other agents:

* **value(Expression1, Expression2):** the first argument is first evaluated to produce the id of an agent. The function then returns the value of the second argument, evaluated with respect to the other agent identified by the first argument.
  + If the first argument is that name of a Group Type (such as School), then the other agent is the administrative agent for the original agent's group of that type (such as the admin agent for the original agent's school). If there is no such admin agent, the result of the function is 0.

## Predicates

Predicates are true/false assertions that are evaluated with respect to the agent evaluating a given rule.

### Comparisons

We often want to Condition an agent's behavior on characterisics such as the agent's age. We can check the age of an agent through a Predicate such as **(age < 19).** Predicates can use any of the numeric comparisons:

**==** (is equal to)

**!=** (is not equal to)

**<** (is less than)

**<=** (is less than or equal to)

**>** (is greater than)

**>=** (is greater than or equal to)

In addition to the two-way comparisions above, there is a three-way predicate than tests whether the first argument has a value between the second and third argument, inclusively:

* **range(Expression,lower,upper**) -- true if the value of the Expression is at least **lower** and no greater than **upper.** For example, the following is true if the agent’s current age is at least 15 and no greater than 20:

**range(age,15,20)**

### Meta-Agent Predicates

The following Predicates test for the agent's status as a meta-agent:

* **is\_import\_agent()** -- true if the agent is the Import Agent.
* **admin(*PlaceType*)** – true if the agent is the administrator of a place of the given type.

### Place Predicates

***Place Predicates*** are defined for relationships between an agent and the places the agent frequents. The following are the currently defined place-Predicates:

* **at(*PlaceType*)** -- true if the agent is currently at the specified type of place, e.g. at(School) is true during times when the agent is actual present at the agent’s school
* **member(*PlaceType*)** – true if the agent belongs to a place specified type, e.g. member(Workplace) could be used as another was of testing whether the agent is employed.
* **host(*PlaceType*)** – true if the agent is the host of a place of the given type.
* **open(*PlaceType*)** – true if the place type is currently open

### Date Predicates

***Date Predicates*** are true-or-false statement about the current simulation date:

* **date(MMM-DD)** -- true if the current simulation date matches the pattern MMM-DD (e.g., Apr-03).
* **date\_range(MMM-DD,MMM-DD)** -- true if the current simulation date falls within the given calendar range.

For example **date\_range(Dec-01,Feb-03)** is true if the current date is later than or equal to December 1st or earlier than equal to February 3rd.

### Exposure Predicates

***Exposure Predicates*** concern where the agent may have been exposed to a given Condition:

* **exposed\_in(*Condition*, *GroupType*)** -- true if the agent was exposed to the given Condition in the given Group
* **exposed\_externally(*Condition*)** -- true if the agent was exposed to the given Condition by an importation event.

### Symbolic Values in Predicates

When testing the current state of an agent, FRED recognizes the user-defined symbolic state names. For example, if the user defined a Condition INCOME with a state named Rich, then the following is a valid Predicate:

**current\_state\_in\_INCOME == Rich**

The symbolic values listed above may occur in equality-testing Predicates. For example, the following are valid Predicates:

**sex == female**

**white != race**

**household\_relationship == householder**

**profile == student**

### Negating Predicates

Predicates may be preceeded by **not**, which negates the remainder of the Predicate. For example,

**not (age < 17)**

is equivalent to

(**16 <= age)** .

# Chapter 5: Mixing Groups

FRED Agents interact with each other in **Mixing Groups.** Examples of mixing groups include households, schools, workplaces, or friendship networks. Some mixing groups represent specific locations (household, schools) and some do not (friendship networks). This chapter deals with **Places**, which are mixing groups that have a specific geo-location, and **Networks,** which represent relationships among agents but are not associated with a specific location.

## Declaring a Mixing Group

To declare a mixing group in FRED, use one of the following statement blocks:

**place *PlaceType* { … }**

**network *NetworkType* { … }**

The code within the brackets defined the properties of groups of the name type.

## Maximum Place Size

Each place type has a maximum snumber of individuals per place. The default value is

**max\_size = 999999999**

## Group Schedules

For an agent to interact with others in a given group at a given time, three requirements must be satisfied:

* The agent must be a *member* of the group
* The group must be *open for business*
* The agent must be *present* in the group.

As an example of how to think about this, consider students in schools. In order to attend a school on a given day, (1) the student must be enrolled in the school, (2) the school must be open that day, and (3) the student must not be absent that day.

Agents can become a member of a group in a few ways:

* The agent can be assigned to a given place by the synthetic population files.
* The agent can join a group as the result of a rule
* The agent can be transmitted an invitation to a group by a group host.

These methods will be described in later sections.

Once an agent is member of a group, the agent is considered to be *at* the group during any time that the group is *open*. The times during which a group is open is modelled in two ways:

* The FRED program can declare the usual open times as a set of properties, or
* the group administrator (a meta-agent) can close the group under the control of rules.

The following properties define the times and days that a group is usually open. These properties apply to all instances of this group type (e.g. all schools).

**starts\_at\_hour\_H\_on\_DAY = N**

In this statement H may take on any integer from 0 to 23, representing a 24-hour clock. DAY is one the values {Sun, Mon, Tue, Wed, Thu, Fri, Sat}. The right-hand side value N is a non-negative integer and means that the group is open for N consecutive hours starting at the indicated hour. The default value of this property is N=0.

As a convenience, groups can also include property statements of the form

**starts\_at\_hour\_H\_on\_weekends = N**

which is equivalent to the following statements:

**starts\_at\_hour\_H\_on\_Sat = N**

**starts\_at\_hour\_H\_on\_Sun = N**

Similarly, the property statement:

**starts\_at\_hour\_H\_on\_weekdays = N**

is equivalent to the following statements:

**starts\_at\_hour\_H\_on\_Mon = N**

**starts\_at\_hour\_H\_on\_Tue = N**

**starts\_at\_hour\_H\_on\_Wed = N**

**starts\_at\_hour\_H\_on\_Thu = N**

**starts\_at\_hour\_H\_on\_Fri = N**

The on\_weekdays and on\_weekends properties override any properties defined for specific days of the week.

The following statement are included in FRED programs by default (in **config.fred**):

**place Household {**

**starts\_at\_hour\_6\_on\_weekdays = 2**

**starts\_at\_hour\_6\_on\_weekends = 2**

**}**

**place Neighborhood {**

**starts\_at\_hour\_16\_on\_weekdays = 2**

**starts\_at\_hour\_16\_on\_weekends = 3**

**}**

**place School {**

**starts\_at\_hour\_13\_on\_weekdays = 2**

**}**

**place Classroom {**

**starts\_at\_hour\_9\_on\_weekdays = 2**

**}**

**place Workplace {**

**starts\_at\_hour\_13\_on\_weekdays = 2**

**starts\_at\_hour\_13\_on\_weekends = 2**

**}**

**place Office {**

**starts\_at\_hour\_9\_on\_weekdays = 2**

**starts\_at\_hour\_9\_on\_weekends = 2**

**}**

This defines a daily schedule for agents. For example, an agent that is a student has the following schedule on weekdays:

* 6am-8am: interact with members of the household
* 9am-11am: interact with members of the same classroom
* 1pm-3pm: interact with members of the same school
* 4pm-6pm: interact with members of the same neighborhood

The weekend schedule is the same except that school activities do not occur and the neighborhood activities are expanded to a three hour period (50% more than on weekdays).

The default schedule above is provided because it has been useful in some early FRED models of infectious disease. It can be adjusted as needed for other FRED models. In particular, if one or more of the built-in place types are not needed in a model, they can be disabled by setting the open hours to zero. For example, to disable interactions in School, include the property in a FRED program:

**place School {**

**starts\_at\_hour\_13\_on\_weekdays = 0**

**}**

This statement will override the default property, and school interactions will be disabled for all agents.

By default, an agent is present in any of its associated groups whenever that group is open. However, agents may decide to be absent by entering specific states.

To handle dynamic group closures (e.g. temporary school closures), you may assign an "administrator" agent to each group.

**has\_administrator = 0/1 (default = 0)**

The administrator can close a group by entering a state with the "close\_admin\_group" property.

## Places

Places are Mixing Groups that have a specific geo-location (latitde, longitude, elevation), as well as other properties. The user can declare any number place types in FRED. Place types are declared by the property block:

**place *PlaceType* { … }**

The default configuration file includes these place types:

**place Household { … }**

**place Neighborhood { … }**

**place School { … }**

**place Classroom { … }**

**place Workplace { … }**

**place Office { … }**

**place Hospital { … }**

Other place types can be added by the user.

For the built-in place types Household, School, Workplace and Hospital, the FRED synthetic population includes files that list each place, its location, its elevation (above sea level), and links to individuals in the population. The remaining built-in places are associated with agents by FRED as described under **Partitions** below.

By default, all of the initial agents are assigned membership in the default place type according to the input files in the synthetic population. Not all agents are members of all place types, for example, students are members of their school but usually not members of a workplace. Membership in user-defined places is controlled by rules, as explained later.

### User-Defined Place Types

The user can define a new place type by adding a statement block to the FRED program:

**place *PlaceType* { … }**

where *PlaceType* is the name of the class of places. The convention is to use a captalized, singular name such as School or Workplace. For example, if the model includes pharmacies, the program would include the statement:

**place Pharmacy { … }**

There are three ways to set the location of user-defined places:

* Defining places in the FRED program
* Place creation by the Import agent
* Reading from a file within the synthetic population
* Defining Places in the FRED Program

#### Defining Places in the FRED Program

The FRED program can specify new place locations with the property statement within the Place type code block:

**add = *PlaceId Latitude Longitude Elevation***

For example,

**place Pharmacy {**

**add = 942003001 40.451164 -79.999803 230.1**

**add = 942003002 40.626449 -79.723195 240.5**

**}**

The *PlaceId* is an integer value with up to 18 digits. Place id's must be unique among all places in a given FRED model. In order to avoid conflicts with places defined by the FRED synthetic population, the first digit in user defined places should be **9** as in the above example. It's a good idea to use the place id to code meaningful information. In the example above, we picked the place id to code the fact that this is the first pharmacy within the area of FIPS code 42003 (Allegheny County, PA). The FRED synthetic population uses the following conventions for the first digit of place id's:

|  |  |
| --- | --- |
| First Digit: | Place Type: |
| 1-2 | Households |
| 3 | Reserved |
| 4 | Schools and Group Quarters |
| 5 | Workplaces |
| 6 | Hospitals |
| 7 | User-defined Places with place id = 0 |
| 8 | Reserved |
| 9 | User-defined Places |

If the user-supplied place id is 0, then FRED assigned a unique place id with an initial digit of 7. For example, if the user does not care about the specific id's assigned to the pharmacies, the program can contain statement such as:

**add = 0 40.451164 -79.999803 230.1**

**add = 0 40.626449 -79.723195 240.5**

The remaining numbers on each line give the latitude, the longitude, and the elevation in meters above sea-level for each place. If these are unknown, they can be replaced by 0. Of course, in that case the information can not be meaningfully used within the model.

#### Places Created by the Import Agent

New places are generated by the Import Agent when it transmits a Condition that transmits places. See the Section on [Transmission of Places](#_Transmission_of_Places). In this case, the location of the newly generated place depends on the **base\_type** place of the exposed agent as explained in Chapter 8. The place id of the new place is generated by FRED and begins with the digit 7. The latitude, longitude, and elevation of the new place is inherited from the base\_type location for the agent.

#### Places Defined in Files in the Synthetic Population

For command line users of FRED with access to a local copy of the synthetic population, it is possible to create location files for user-defined places. The name of the file must be *placetype*.txt where *placetype* is the lowercase version of the *PlaceType*. For example, of the Place Type is Pharmacy, the file should be named **pharmacy.**txt. The file must be placed in the appropriate county level directory. The format of the file is shown in the following example:  
  
sp\_id lat lon elevation  
942003001 40.451164 -79.999803 230.1

942003002 40.626449 -79.723195 240.5

...

The file must have a column title line of “sp\_id”, “lat”, “lon” and “elevation.” Each record is on its own line with values being separated by a single tab. As with place locations defined in the FRED program, if the value in the first column is 0, FRED will assign a unique place id with an initial digit of 7.

Partitions  
  
You can can partition any Place\_Type into sub-places (e.g., the default property settings in FRED partition Schools into Classrooms).  The following statements in the place type code block control the partition process:

**partition = *PlaceType2***

**partition\_size = *Number***

**partition\_basis = none | age | random**

**partition\_min\_age = *Number***

**partition\_max\_age = *Number***

The default property settings include the following statements:

**place School {**

**partition = Classroom**

**partition\_size = 40**

**partition\_basis = age**

**partition\_max\_age = 21**

**}**

The above properties divide the agents in a given school into classrooms by age. The maximum size of any classroom is 40, and the maximum age for membership in a classroom is 21.

**place Workplace {**

**partition = Office**

**partition\_size = 50**

**partition\_basis = random**

**}**

The above properties divide the agents in a given workplace into offices of maximum size 50. The assignment is made by random selection.

The creation of partitions and the assignment of agents to partitions is performed prior to the start of the simulation and uses a different random number seed for each run. That is, if FRED is run 10 times, the set of classmates for a given student will vary for each run. However, the assignment of students to schools is based on the synthetic population files and is the same from run to run. The same applies to workplaces and offices. Partition membership cannot be assigned through synthetic population files.

Inviting agents to a place  
  
One agent can "invite" another agent to a place, for example, a host might invite people to a party by transmitting the party location. This is discussed further in Chapter 8.

## Networks

**Networks** are mixing groups that do not have a specific geo-location. Networks differ fundamentally from Places in that all the people who share a given Place at a given time are assumed to interact with each other uniformly (except for a possible bias to interact with people of a similar age), whereas people interact in a given Network only with other people to whom they are explicitly linked. Networks in FRED can be used to represent network relationships such as friendships, sexual partners, buyers/sellers, health care providers/patients, and other relationships.

The user can declare any number of Network types in FRED, and each Network type has a set of properties that determine how agents interact within that type of Network. Network types are declared by the property statement:

**network *Network* { ... }**

where the code block between brackets include the properties of the Network type. Each Network type is associated with exactly one Network, which has the same name as the Network type. Networks in FRED may contain several connected components; that is, there may not be a path between every pair of individuals in a Network. In general, individuals are linked to a subset of the other members of the Network. Networks may be undirected or directed. There are no built-in Network types.

For an agent to interact with others in a given network at a given time, three requirements must be satisfied:

* The agents must both be *members* of the network, and
* The network must be *open*. Like Places, network might have an administrator, a meta-agent who can decide to temporariliy close the network, and
* The agent must be *active* in the network.

The properties that define the times and days that a network is open are the same as for Places. By default, an agent is active in any of its associated networks whenever the network is open. However, agents may decide to be absent from a network by entering specific states, as explained later.

### Joining a Network

An agent can **join** a network without creating links to other agents. An agent joins a network through Action of the form:

**join(*Network*)**

The effect of this action is that the agent will join the named Network, unless the agent already belongs to the Network. In that case, the action has no effect. This rule does not create any links to other agents in the network.

### Adding Edges

Agents add edges as a result of the following Rule types, explained in Chapter 5:

**add\_edge\_to(*Network*,*Expression*)**

**add\_edge\_from(*Network*,*Expression*)**

If an agent adds an edge using one of these rules, the agent first joins the Network if not already a member.

### Deleting Edges

Edges between agents in a network can be deleted through actions of the form:

**delete\_edge\_to(*Network*,*Expression*)**

**delete\_edge\_from(*Network*,*Expression*)**

#### Example: Defining a Friendship Network

Suppose we want to include a friendship network in a model. We could declare a Network called Friends as follows:

**network Friends { ... }**

We can to declare a Condition that controls the social relationships of agents:

**condition SOCIAL {**

**states = Start Join Choose Add Delete None Excluded**

**}**

The program will use a list variable to hold each agent's candidates for friendship, and an ordinary variable to store the id of the next selected candidate:

**list candidates**

**my newfriend**

The first set of Rules include some agents between 15 and 17 years old in the Network at time 0:

**state SOCIAL.Start {**

**if (range(age,15,17)) then next(Join) with prob(0.01)**

**default(None)**

**}**

**state SOCIAL.Join {**

**join\_network(Friends)**

**wait(1)**

**default(Choose)**

**}**

It is important to wait before entering the Choose state, since we want all the agents to finished joining the network before we choose a new friend. After waiting 1 hour, the agent enters a state in which a new friend is chosen. First, we set the list of candidates to all those agents who are either in the Friends network already or who share a Classroom with the agent, and who are within one year of the agent's age. Next, the agent selects a new friend from the candidate list, giving preference to candidates who are close in age to the agent.

**state SOCIAL.Choose {**

**set(candidates,filter(pool(Classroom,Friends), abs(age-other:age)<2))**

**set(newfriend,select(candidates,pref(-10\*abs(age-other:age))))**

**wait(0)**

**next(Add)**

**}**

The agent then proceeds to add an edge to the new friend:

**state SOCIAL.Add {**

**add\_edge\_to(Friends,newfriend)**

**}**

### Quitting a Network

An agent may quit its membership in a network with an action of the form:

**quit(*Network*)**

The effect of this action is that the agent will delete all its edges in the given Network and end its membership in the Network. If the agent is not a member of the Network, the action has no effect.

### An Agent's Degree in a Network

If the network is undirected, the number of agents to whom an agent is linked is called the **degree** of the agent in the network. If the network is directed, the number of outward edges from an agent is called the **out\_degree** of the agent and the number of inward edges is called the **in\_degree** of the agent. In undirected networks, the **degree**, the **in\_degree**, and the **out\_degree** are all the same value

# Chapter 6: Rules for States

For each State in a given Condition, the modeler should answer three questions:

1. What happens when an agent enters this State? That is, what does this State mean in terms of the agent's own status or how the agent interacts with others?
2. How long does the agent stay in this State?
3. What State does the agent go to next?

The answers to these questions are expressed in Rules that control how agents change during a FRED simulation. There are three major categories of Rules in FRED that corresponds to the questions above:

1. Action Rules
2. Wait Rules, and
3. Transition Rules.

Each State in FRED must include a block of code that declares the Rules associated with the State:

**state *Condition*.*State* {**

***Action Rules***

***Wait Rules***

***Transition Rules***

**}**

For convenience, FRED interprets an empty code block like

**state *Condition*.*State* { }**

as

**state *Condition*.*State* {**

**action()**

**wait()**

**next()**

**}**

This means that the given state has no actions, waits indefinitely and has no transitions to other states. This configuration is common among terminal states and is always assumed for the **Excluded** pseudostate.

Action Rules control how the agent’s own internal status changes when the agent enters a state. For example, entering a state can change the agent's susceptibility to the current Condition:

**set\_sus(INF, 1.0)**

Wait Rules control how long an agent stays in a given state, expressed in hours. For example, an infected agent might spend two days in a state:

**wait(48)**

Transition Rules control how an agent moves from one state to another:

**next(Infectious)**

All rules can be qualified by a list of clauses called ***Predicate***, for example:

**[ if (*PredicateList*) then ] next(*State*)**

The part in brackets is optional. If present, then the rest of the rule only applies if all the tests in the ***PredicateList*** are true for the agent. We now discuss each type of Rule in more detail.

## Action Rules

Entering a State can cause changes to an agent; these changes are called **Actions**. Actions are defined through Rules of the form:

**[if (*PredicateList*) then ] *Action*(*args*)**

Actions include the following:

### Demographics Actions

The following actions effect the life and death of agents:

* **give\_birth() --** Create a new agent who is the offspring of the current agent.
* **die() --** The agent dies.

### Actions Affecting an Agent's Groups

Interactions between agents occur with mixing Groups, include Places and Networks.The following actions determine which other agents the agent interacts with.

* **join(*GroupType*)** -- The agent will select and join a group of the given type, *GroupType*. If the agent already belongs to group of the given type, the action has no effect.
* **join(*PlaceType*, Expression)** -- The agent will join a specified place of the given type. The specific place that the agent joins is the value of the Expression, treated as an sp\_id (unique lidentifier) for the place. If the agent already belongs to place of the given type, the agent first quits that place before joining the specified place. If the place already has maximum size, the action has no effect.
* **quit(*GroupType*)** -- If the agent belongs to a group of the given type, the agent will leave that group; otherwise, the action has no effect.
* **absent(*GroupList*) --** The agent does not attend any of the listed groups while in this state.If the list is empty, the agent does not attend any group.
* **present(*GroupList*) --** This action cancels any previously define absences for the listed groups.If the list is empty, the agent resumes it normal schedule of group activities.
* **add\_edge\_to(*Network*,*Expression*) --** An edge in the given Network is added from the agent to the agent whose id is the value of the Expression. Both agents join the Network if they are not already members. If the Expression does not evaluate to valid agent id, the action has no effect.If the Expression is a list-valued expression (that is, a list of agent id's), then an edge is added to each agent in the list.For example, the followng action would create an edge in the network HH to all members of the agent household:

**add\_edge\_to(HH, pool(Household))**

* **add\_edge\_from(*Network*,*Expression*) --** An edge in the given Network is added to the agent from the agent whose id is the value of the Expression. Both agents join the Network if they are not already members. If the Expression does not evaluate to valid agent id, the action has no effect.If the Expression is a list-valued expression (that is, a list of agent id's), then an edge is added from each agent in the list.
* **delete\_edge\_to(*Network*,*Expression*) --** If an edge exists in the given Network from the agent to the agent whose id is the value of the Expression, then the edge is deleted. Otherwise, the action has no effect.If the Expression is a list-valued expression (that is, a list of agent id's), then an edge to any agent in the list is deleted.
* **delete\_edge\_from(*Network*,*Expression*) --** If an edge exists in the given Network to the agent from the agent whose id is the value of the Expression, then the edge is deleted. Otherwise, the action has no effect.If the Expression is a list-valued expression (that is, a list of agent id's), then an edge from any agent in the list is deleted.

### Actions that Change an Agent's State

Entering a state may cause the agent to change from one state to another state in another Condition. As an example, an agent that enters a state representing receiving immunity from a vaccine may have the effect of changing from a susceptible state to a non-susceptible state for one or more disease Conditions.

* **set\_state(*Condition2*,*State2,State3*)** -- If the agent is currently in***Condition2*.*State2****,* then the agent’s current state in ***Condition2***becomes***State3****.*
* **set\_state(*Condition2*,*State2*)** -- The agent’s current state in ***Condition2***becomes***State2***, regardless of the previous state of the agent in ***Condition2.***

### Actions that Change an Agent's Variables

An agent can have any number of variables that keep track of values of interest. The following Actions change the agent's variables when the agent enters the given state:

* **set\_sus(Condition,*Expression*) --** Change the agent's susceptibility to the named Condition to the value of the expression.
* **set\_trans(Condition,*Expression*) --** Change the agent's transmissibility for the named Condition to the value of the expression if the agent is an ordinary agent. If the agent is a meta-agent, the transmissibility of the named Condition is changed to the value of the expression.
* **set(*Variable,Expression*) --** Set the agent’s ***Variable*** to the value of the expression.
* **set(*Variable,Expression1,Expression2*) --** Expression2 is an expression that returns a **target agent** id. The target agent’s ***Variable*** is set to the value of the ***Expression1***, which is evaluated with respect to both the agent executing the rule and the target agent. For example,

**set(AgeFactor, age+2\*other:age, MyFriend)**

If this action is executed by agent N and the value of agent N's **MyFriend** variable is the id of agent M, then the **AgeFactor** variable of agent M is set to the sum of agent N's age and 2 times agent M's age.

* **set\_weight(*Network,Expression1,Expression2*) --**If the agent is a member of the given Network, set the weight of selected edges.If Expression1 is a list-valued expression that returns a list of agent id's, then each edge between the agent executing the rule and an agent in the list is set to the value of Expression2, which may include factors defined over either or both agents. For example, the following would set the weight of any edge between the agent and other members of its household in the given Network to the difference between the agent's own age and the age of the other household member:

**set\_weight(Network, pool(Household), (age - other:age))**

If Expression 1 is a single-valued expression and the agent has an edge to the agent whose id is the value of *Expression1*, then set the weight of the edge from the agent to the other agent to the value of *Expression2,* which may include factors defined over either or both agents.

* **set\_list(*ListVariable,* *ListExpression*) --** Set the agent’s ***ListVariable*** to a list returned by the given List-valued Expression.

### Actions that Produce Outputs

The following action control values that the Agent reports in the simulation output.

* **report(*Factor*)** -- The agent will report the value of the given *Factor*. For example, consider a model with a Condition called PARTY that describes whether the agent is invited to a PlaceType Party. Suppose on the the states is called Attend. Then the action **report(size\_of\_Party)** would cause the agent to report the size of the agent’s current Party. The result is a time-series of values, but the values are only updated when the agent enters the state **PARTY.Attend**.

## Administrative Actions

Some actions can only be performed by [***administrative agents***](#_Administrators). If the following actions are invoked by ordinary agents, they have no effect.

* **close(*Group*) --** The Group associate with this administrative agent is closed as long as the agent remains in the given state. See the examples in Chapter 19.
* **set\_contacts(*Expression*) –** The contact rate for the Group associated with this administrative is multiplied by the value of the given *Expression*. The effect is to change the number of contacts among the agents attending the Group. This can change the rate of transmissions that occur within the Group.
* **randomize\_network(*Network, Expression1, Expression2*) --** The Network administered by the agent is converted to a random network. The new mean degree of the network is determined by the value of the first expression, and the maximum degree allowed is determined by the value of ther second expression.

## Import Actions

The following action has a special effect when performed by the [*Import Agent*](#_The_Import_Agent).

* **set\_trans(Condition,*Expression*) --** If the agent is a meta-agent, the transmissibility of the named Condition is changed to the value of the expression.

Some actions can only be performed by the [*Import Agent*](#_The_Import_Agent). If the following actions are invoked by other agents, they have no effect.

* **import\_count(Expression) --** The expression is evaluated and the Import Agent selects the specified number of susceptible agents for exposure.
* **import\_per\_capita(Expression) --** The expression is evaluated and the Import Agent selects the specified fraction of susceptible agents for exposure.
* **import\_ages(lower, upper) --** The exposed agents all have age at least lower and less than or equal to upper.
* **import\_list(ListExpression) --** The agents with id's on the list will be exposed exposed. **Note:** if there is an import\_list rule for a given state, it overrides any other import rules for that state.
* **import\_location(Expression1, Expression2, Expression3) --** The exposed agents all have households near a specific latitude and longitude. The latitude is the value of the first expression, the longitude is the value of the second expression, and the radius is the value of the third expression.
* **import\_admin\_code(N) --** The exposed agents all have households within the region defined by administrative code N, where N is an 11-digit FIPS code for a census tract, and 5-digit FIPS code for a county, or a 2-digit FIPS code for a state (U.S. only).
* **count\_all\_import\_attempts() --** This action changes the number of individuals are who selected for exposure if the **import\_count()** action is in effect.

These actions are explained in more detail in [Chapter 8](#_Chapter_8:_Transmission).

## Wait Rules

Whenever an agent enters a State, FRED determines how long the agent should wait in that State before the Transition Rules are applied to select the next state. The wait time for each State is defined through Wait Rules that have the following two forms:

**[ if (PredicateList) then ] wait(*WaitExpression*)**

The protion in brackets is optional. One form of WaitExpression specifies a duration (**in hours**) for how long to wait until transition decisions are made. Some examples:

**wait(1) --** Wait 1 hour

**wait(0) --** Apply Transition Rules immediately. Transient states with zero wait times are often useful, since they can cause effects that change the agent's other Conditions, or they can serve as decision points that separate the state trajectories of agents based on their properties.

**wait() --** Wait indefinitely.

Any expression is permitted, so if the wait time should be drawn from a normal distribution with median 5.5 days and standard deviation of 2.0 days, you can say:

**wait(24 \* normal(5.5, 2.0) )**

State durations are often defined using statistical distributions, including those shown in the Table below:

### Useful Distributions for State Durations

|  |  |  |
| --- | --- | --- |
| Distribution | Parameters | Use |
|  | number | Activities with a fixed duration, e.g. attending a class. |
| normal | mean, std | Activities produced by many small effects acting additively and independently, e.g., human gestation period |
| lognormal | median, dispersion | Duration produced by multiplication of many effects, e.g., duration of illness, length of marriage before divorce |
| geometric | mean  (mean = 1.0/p) | time before success of Bernoulli trials with prob p.  e.g., time before winning a lottery |
| uniform | min, max | e.g., value of a dice toss |

The second form of WaitExpression is useful when the state should last until a certain time or date:

**wait(until\_3pm)**

* Wait until the next occurrence of 3pm.
* If it is now 3pm, this would wait 24 hours.

**wait(until\_Fri\_at\_3pm)**

* Wait until the next occurrence of Friday at 3pm.
* If it is now 3pm on Friday, this would wait one week.

**wait(until\_Today\_at\_9am)**

* Wait until 9am today.
* If it is already 9am or later, the agent will wait forever.

**wait(until\_Tomorrow\_at\_3pm)**

* Wait until 3pm the following day.

**wait(until\_10\_days\_from\_now\_at\_3pm)**

* Wait until 3pm 10 days in the future.

**wait(until\_Apr-03\_at\_2pm)**

* Wait until 3pm on the next April 3rd.
* If today is April 3 but already 2pm, wait until 2pm on April 3 of next year.

**wait(until\_2021-Apr-03\_at\_2pm)**

* Wait until 3pm on April 3, 2021. If it is already past 2pm on April 3, 2021, then wait forever.

If no **am** or **pm** is given, then the hour is interpreted as a 24-hour clock, with **at­\_0** meaning midnight.

Each state must have at least one unconditional wait rule, or a compiler error will result and the program will not execute. If multiple wait rules are list for a given state, the first one that applies is executed. For example:

**state INF.Symptoms {**

**set(sick,1)**

**if (age > 60) then wait(24\*7)**

**if (age <= 60) then wait(24\*5)**

**wait()**

**next(Recovered)**

**}**

In the example above, we assign a duration of Symptoms that depends on the age of the agent: 7 days for agents over 60 and 5 days for everyone else. Even though the final **wait()** will never be reached in this case, it is required in order that FRED knows what to do in case the conditional statements all fail. In general, the FRED compiler is unable to determine whether at least one of the conditions will be true for all agents.

## Transition Rules

Transition Rules control the next state that the agent will assume. There are two kinds of Transition Rules:

* Probabilistic Rules
* Default Rules

### Probabilistic Rules

**[ if (*Predicate1,...,PredicateN*) then ] next(*State*) [ with prob(*Expression*) ]**

This rule means that if the agent satisfies all the Predicates specified, then the probability[[1]](#footnote-1) of the agent entering the state in the **next** clause is the value of ***Expression***. Expressions are defined over numerical properties of the agent, such as its age. If the **with prob(*Expression***)is omitted, it is equivalent to **with prob(1).**

An example of a transition rule in a FRED program might be:

**if (age > 16) then next(Dropout) with prob(0.2)**

The rule says, “if an agent is more than 16 years old, then the probability of the agent’s becoming a Dropout is 20%.”

### Default Next State Rules

Within each state’s code block, there may be one transition rule of the form:

**default(*State*)**

Note that this rule does not have an if clause. This rule says that the default next state is the given **State**, if none of the other next states are selected by the other transitin rules. Specifically, if the next state probabilities for all other next states sum to a value p < 1.0, then the transition probability from the current state to ***State*** is set to (1-*p*). If more than one default transition rule is present for a given state, the last one in the program file will apply.For each State, the built-in default Transition Rule is:

**State *Condition*.*State* { default(*State*) }**

That is, an agent will stay in the same state if there is no other Transition Rule. If no explicit transition rules or default rules are included in the FRED model file, the compiler will issue a warning to remind the user that the state will transition back to itself by default.

# Chapter 7: State Transitions

This Chapter describes how Transition Rules are used to decide what state an agent goes to next.

Many systems are described by a state-transition matrix that defines the probability for moving from the state that labels a row in the matrix to any of the other states, with the individual probabilities represented by the columns in the matrix. For example, in the matrix below, there is a probability of 0.5 of moving from state A to either state B or state C.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  | Next State | | |
|  |  | A | B | C |
|  | A | 0.0 | 0.5 | 0.5 |
| Current State | B | 0.1 | 0.8 | 0.1 |
|  | C | 0.0 | 0.0 | 1.0 |

In FRED, only the portion of the state-transition matrix required for each agent is computed dynamically, so that the transition probabilities can vary from agent to agent and can also vary over time for an individual agent. When an agent reaches the state transition duration for its current state, the agent’s next state is computed as follows:

1. Compute the transition probabilities for this agent in the row of current state.
2. Select next state using the probability distribution for the row of the current state.

For example, given the transition matrix shown above for a given agent in state B, the agent would transition to state A with a probability of 0.1, transition to state C with a probability of 0.1 and remain in state B with a probability of 0.8.

## Probabilistic Rules

Suppose we have an agent in state C.i for some Condition C that has states n States. For each possible next state j, FRED computes the probability p(i,j) of the transition from i to j as follows:

1. Find all qualifying rules for the given agent from State i to State j. These are the rules that specify the current state C.i and the next state j and for which the agent meets all the Predicates in the rule.
2. For each qualifying Rule, compute the probability term for the given agent. If the probability term is omitted, the probability for this Rule is 1.0.
3. Use the maximum probability computed for any qualifying Rule as the tentative probability p(i,j) for the agent to transition from state i to state j.

## Default Next State Rule and Normalization

After repeating this process for each possible next state, we have a probability vector of the form:

**p(i,1), p(i,2), … p(i,n)**

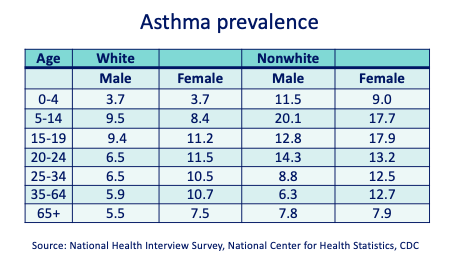
Let *Total* be the sum of the values in the vector. *Total* may not equal 1.0, so the values need to be normalized to create a valid probability distribution. But first, FRED applies the Default next state Rule for state i. Let state *k* be the default next state for state i. The effect of the Default Next State is:

If *Total* < 1.0, then add (1.0 – Total) to p(i,k). Otherwise, divide all the values in the vector by *Total.*

After applying the rule above, the values in the modified vector **<p(i,1), p(i,2), … p(i,n)>** sum to 1.0. The next state is selected by choosing randomly with this vector as the probability weight for the states.

## Example: Using Rules to Represent Health Risks

This Section shows how to assign to each individual in the population a risk of having various Conditions. For many health Conditions, we can find tables that show the prevalence of a given health Condition based on demographic factors such as age, race and sex. For example, the following table shows the prevalence of asthma in the United States, based on data from the CDC:



Suppose we are creating a FRED model of asthma in which all individuals begin the the Start state. At the start of the simulation, we want each individual to move to either the AtRisk state (meaning that they are subject to asthma attacks) or to the Negative state (meaning they do not suffer from asthma attacks), according to data in the Table above.

This can be accomplished by having one transition rule per cell in the Table, as shown in the following Condition:

**condition ASTHMA {**

**states = AtRisk Acute Recovered Negative Import**

**transmission\_mode = environmental**

**exposed\_state = Acute**

**}**

**state ASTHMA.Start {**

**if (sex==male, race==white, range(age, 0, 4)) then next(AtRisk) with prob(0.037)**

**if (sex==male, race==white, range(age, 5, 14)) then next(AtRisk) with prob(0.095)**

**if (sex==male, race==white, range(age, 15, 19)) then next(AtRisk) with prob(0.094)**

**if (sex==male, race==white, range(age, 20, 24)) then next(AtRisk) with prob(0.065)**

**if (sex==male, race==white, range(age, 25, 34)) then next(AtRisk) with prob(0.065)**

**if (sex==male, race==white, range(age, 35, 64)) then next(AtRisk) with prob(0.059)**

**if (sex==male, race==white, 65<=age) then next(AtRisk) with prob(0.055)**

**if (sex==female, race==white, range(age, 0, 4)) then next(AtRisk) with prob(0.037)**

**if (sex==female, race==white, range(age, 5, 14)) then next(AtRisk) with prob(0.084)**

**if (sex==female, race==white, range(age, 15, 19)) then next(AtRisk) with prob(0.112)**

**if (sex==female, race==white, range(age, 20, 24)) then next(AtRisk) with prob(0.115)**

**if (sex==female, race==white, range(age, 25, 34)) then next(AtRisk) with prob(0.105)**

**if (sex==female, race==white, range(age, 35, 64)) then next(AtRisk) with prob(0.107)**

**if (sex==female, race==white, 65<=age) then next(AtRisk) with prob(0.075)**

**if (sex==male, race!=white, range(age, 0, 4)) then next(AtRisk) with prob(0.115)**

**if (sex==male, race!=white, range(age, 5, 14)) then next(AtRisk) with prob(0.201)**

**if (sex==male, race!=white, range(age, 15, 19)) then next(AtRisk) with prob(0.128)**

**if (sex==male, race!=white, range(age, 20, 24)) then next(AtRisk) with prob(0.143)**

**if (sex==male, race!=white, range(age, 25, 34)) then next(AtRisk) with prob(0.088)**

**if (sex==male, race!=white, range(age, 35, 64)) then next(AtRisk) with prob(0.063)**

**if (sex==male, race!=white, 65<=age) then next(AtRisk) with prob(0.078)**

**if (sex==female, race!=white, range(age, 0, 4)) then next(AtRisk) with prob(0.09)**

**if (sex==female, race!=white, range(age, 5, 14)) then next(AtRisk) with prob(0.177)**

**if (sex==female, race!=white, range(age, 15, 19)) then next(AtRisk) with prob(0.179)**

**if (sex==female, race!=white, range(age, 20, 24)) then next(AtRisk) with prob(0.132)**

**if (sex==female, race!=white, range(age, 25, 34)) then next(AtRisk) with prob(0.125)**

**if (sex==female, race!=white, range(age, 35, 64)) then next(AtRisk) with prob(0.127)**

**if (sex==female, race!=white, 65<=age) then next(AtRisk) with prob(0.079)**

**default(Negative)**

**}**

**state ASTHMA.AtRisk {**

**set\_sus(ASTHMA, 1.0)**

**wait()**

**next()**

**}**

**state ASTHMA.Negative {**

**set\_sus(ASTHMA, 0.0)**

**wait()**

**next()**

**}**

Each rule contains a Predicate that matches the demographic factors of each cell in the table, and then assigns the same probability of being at risk as the corresponding cell in the table. It is worth noting that any individual in the population matches only one of the probability rules, because the Predicates are mutually exclusive. For example, a 5-year old white male only matches the second rule, so his transition probability from Start to AtRisk is 0.095. The effect of the default rule is to set the transition probability from Start to Negative of this same individual to 1.0 - 0.095 = 0.905. So, we expect that about 9.5% of 5-year old white males in the population will be in the AtRisk state after the transition. Of course, the results will vary from run to run, since each individual makes a probabilistic transition.

# Chapter 8: Transmission

One way that agents interact in FRED is through a process called ***transmission****.* Transmission means that one agent causes another agent to enter a certain state within a Condition. The first agent is said to be ***transmissible*** for the Condition, and the second agent is said to be ***susceptible*** to the Condition. We also say that the first agent ***exposes*** the second agent to the Condition. Transmission can also be caused by the action of the Import meta-agent.

## Transmissibility

There are three forms of *transmissibility* that serve three different functions in FRED models:

* the transmiisibility of a Condition
* the transmissibility of each agent
* the transmissible actions of the Import Agent.

Each of these is decribed below.

The Transmissibility of a Condition

To make a Condition transmissible, the FRED program must declare a ***transmissibility*** property for the Condition:

**transmissibility = *X***  
  
The value *X* (set to 0 by default) is the level of contagiousness for the Condition, and is used to determine the probability that an encounter between a transmissible agent and susceptible agent results in a transmission event. The higher the value of X, the greater the likelihood of contagion from one agent to another. A value of X = 1.0 corresponds to the transmissibility of a “standard” influenza (R0 = 1.2). The Section on calibration discusses how to set transmissibility for other Conditions if the R0 is known.

Transmissible Conditions must also have a ***transmission mode*** property:

**transmission\_mode = none | respiratory | proximity | vector | network | environmental**

The default is “**none**”.

Each transmissible Condition has a distinguished state called the ***exposed state***. When an agent is exposed to a transmissible Condition, the agent immediately goes into the state specified by the property:

**exposed\_state = State**

For example:

**condition INFLUENZA {**

**exposed\_state = Exposed**

**}**

tells FRED that if an agent who is infectious with **INFLUENZA** transmits the disease to another agent, the second agent will enter the state called **Exposed**.

The Transmissibility of an Agent

Each agent has an individual level of *transmissibility* for each Condition that may change depending on the state of the agent.  An agent's transmissibility is controlled by the action:  
  
**set\_trans(COND, Y)**  
  
This action allows an agent's individual transmissibility to change, for example, when the agent moves from a non-infectious state of a transmissible disease to an infectious state.  
  
Each agent also has an individual level of *susceptibility* to each Condition, and this level may change depending on the state of the agent.  An agent's susceptibility is controlled by the action:  
  
**set\_sus(COND, Z)**

Normally, when a transmissible agent with individual transmissibility Y interacts with a susceptible agent with individual susceptibility Z, the probability of a transmission event is proportional to X\*Y\*Z, where X is the transmissibility of the Condition.  The probability also depends on other factors like the contact likelihood in the place where they meet and other factors described below.

## Transmission by the Import Agent

The Import Agent is a special meta-agent that can affect various aspects of the population. Among other powers, the Import Agent can alter the transmissibility of a Condition by a rule containing the following action:

* **set\_trans(Condition,*Expression*) --** If the agent is a meta-agent, the transmissibility of the named Condition is changed to the value of the expression.

Furthermore, the Import\_Agent can cause the transmission of a Condition to a susceptible individual.  This is called ***importing an*** exposure. There are several ways to control the timing and the number of imported exposures via actions executed by the Import Agent. If any other agent attempts to execute these CTIONS, it will have no effect. The actions that an Import Agent can perform include:

* **import\_count(Expression) --** The expression is evaluated and the Import Agent selects the specified number of susceptible agents for exposure. For example, suppose we have a rule that says:

**state COND.Import {**

**import\_count(100)**

**}**

Then, when the Import Agent enters the Import state, it will expose up to 100 agents to Condition COND. If there are fewer than 100 susceptible agents, then the number exposed will be the number of susceptible agents. If there are at least 100 susceptible agents, then agents will be selected agents and attempt to expose them until 100 agents are successfully exposed. The probability of a successful exposure is proportional to the agent's susceptibility to the Condition. That is, if an agent's susceptibility to the Condition is 0.2, then if that agent is selected from exposure, the probability of success is 0.2. Therefore, agents with lower susceptibility have a smaller chance of exposure than agents with a higher level of susceptibility.

* **import\_per\_capita(Expression) --** The expression is evaluated and the Import Agent selects the specified fraction of susceptible agents for exposure. For example,

**import\_per\_capita(0.25)**

This action says the Import Agent will attempt to expose 25% of the susceptible agents to Condition COND. Again, agents with a greater level of susceptibility will be more likely to be exposed that agents with lower susceptibility.

* **import\_ages(lower, upper) --** The exposed agents all have age at least lower and less than or equal to upper. Example

**import\_ages(10,20)**

With this action the Import Agent will only expose susceptible agents whose age is in the range 10-20.

* **import\_list(ListExpression) --** The agents with id's on the list will be exposed. Example

**import\_list(list(123,456,789))**

This will expose agents 123, 456, and 789. This may be useful when you want to expose the exact same agents in multiple simulation runs. **Note:** if there is an import\_list rule for a given state, it overrides any other import rules for that state.

* **import\_location(Expression1, Expression2, Expression3) --** The exposed agents all have households near a specific latitude and longitude. The latitude is the value of the first expression, the longitude is the value of the second expression, and the radius is the value of the third expression.For example,

**import\_location(40.0, -80.0, 10.0)**

This says the Import Agent will only expose susceptible agents that live within 10 km of latitude 40 and longitude -80.

* **import\_admin\_code(N) --** The exposed agents all have households within the region defined by the FIPS code N.For example,

**import\_admin\_code(42003000112)**

This says the Import Agent will only expose susceptible agents that live within census tracts 42003000112.

* **count\_all\_import\_attempts() --** This action changes the number of individuals are who selected for exposure if the **import\_count()** action is in effect, explained below.

What if the Import Agent enters a state with more than one import action rules? In this case the following applies:

* **import\_list** take precedence over all other import actions**.**
* **import\_per\_capita** take precedence over **import\_count.**
* **import\_admin\_code** takes precedence over **import\_location.**
* **import\_ages** always restricts the age of imported cases.

Note that the actions of the Import Agent are not meant to model the interaction of an unknown transmissible person with another susceptible person. Rather, the Import Agent produces a scenario in which some number of agents have been exposed, regardless of how unlikely that scenario might be.

## Transmission in Places

All transmissions from one individual to another requires that the two individuals must share a Mixing Group, which includes Places and Networks. Places in FRED are mixing groups where agents interact equally with all other agents who share the Place. Networks are mixing groups in which an agent interacts only with other agents that have explicit links to the first agent. Transmission works similarly in Places and Networks, but there are some differences as explained below.

When transmissible and susceptible agents are present in the same place, a transmissible agent may transmit a Condition to susceptible agent. Transmission is a stochastic event and depends on several factors including:

* the transmissibility of the Condition
* the transmission mode for the place (by *rate* or by *probability*)
* the hourly contact rate or contact probability for the place
* the duration of the interactions in the place
* random selections of which agents interacts during a given meeting
* the transmissibility of the transmissible agent
* the susceptibility of the susceptible agent
* the similarity of ages if age biases are specific for the given place.

There two possible transmission modes for each type of place: by *rate* or by *probability*. Deepnding on the mode, different algorithms are used to determine the number of transmissions that occur in a given place. The next sections describe each mode.

### Transmission by Rate

*Transmission by Rate* is the default for places in FRED. In this transmission mode. each transmissible agent contacts a number of other individuals during each hour that the place operates. Each contact may result in a transmission.

The property that determines the hourly contact rate is:

**contacts = *Number***

The default number of hourly contacts is 0.0.

The number of *transmission contacts per transmissible agent t* in a Place during its open hours on a given day is given by the formula:

**contacts = (transmissibility of Condition) \* (transmissibility of agentt)**

**\*(hourly contact rate of Place)\*(duration of the meeting in the Place)**

The duration of the meeting in the Place is set by the Place Schedule (see [Group Schedule](#_Group_Schedules)). The hourly contact rate is the same for all places of a given PlaceType, and is set by the property:

**contacts = *Number***

The default number of hourly contacts is 0.0.

Each transmissible agent *t* makes the number of contacts computed above. For each such contact event, the agent selects another agent *s* to contact. The probability that agent *t* successfully exposes the selected agent *s* is:

**probability of transmission = (susceptibility of agent *s*) \* (same-age-bias)**

The final term in the equation above is a factor the reflects the tendency of agents of similar age to interact with a given Place, and is controlled by the property:

**same\_age\_bias = *Number***

There is also a a global property that enables transmission in Places to take age-bias into account, and it is enabled by default:

**enable\_transmission\_bias = 1**

If the Place as a positive age-bias, the bias is computed by the formula:

**same-age-bias(*t, s*) = exp(-(same\_age\_bias of Place)\*(|age of *t* - age of *s*|))**

That is, the probability of transmission decreases with larger difference in age between the interacting agents.

There several important details in this process:

* Agents do not contact themselves in this process. Each contact is assumed to be between two distinct agents.
* The agent *s* selected to contact may be selected more than once. That is, if an agent in a given place is assumed to have *n* contacts per day, it is not assumed that these contacts are necessarily with *n* distinct other agents.
* The agent *s* is selected among all current members of the Place, not just those currently attending the place. In other words, it is possible that agent *s* is absent when selected. The assumption here is that the number of contacts per agent per place is the nominal number of contacts if everyone is present. If many agents are absent (due to staying home sick, for example), then the number of contacts is effectively reduced proportionally.
* The selection of agent *s* is independent of the susceptibililty, that is, the selected agent is not necessarily susceptible.

The following hourly contact properties are included in FRED programs by default:

**place Household {**

**same\_age\_bias = 0.05**

**contacts = 0.17105**

**}**

**place Neighborhood {**

**same\_age\_bias = 0.1**

**contacts = 0.37475**

**}**

**place School { contacts = 0.20795 }**

**place Classroom { contacts = 0.4159 }**

**place Workplace { contacts = 0.03395 }**

**place Office { contacts = 0.0679 }**

The above values were obtained through the calibration process discussed in Chapter 12 and to reflect the age bias observed in published contact matrices (see Chapter 10).

## Transmission by Probability

The second transmission mode in places is *transmission by probability*. In this mode, each transmissible agent has a fixed probability of contacts any other agent within any given hour that the place is in operation. The major difference between transmission by probability and transmission by rate is the number of contacts per transmissible agents. In transmission by rate, the number of contacts is indepedent of the number of individuals in the place. In transmission by probability, the number of contacts increases proportionally to the number of individuals in the place. In other words, transmission by rate is *indepdendent of density* and transmision by probability is *dependent on density.* The choice of appropriate transmission mode varies by model and by condition. For example, if a condition is transmitted through close contacts between individuals, then a density independent mode may be appropriate because an individual can only have a limited number of close contacts within a given time period. On the other hand, if a condition is freely transmitted to everyone an individual comes into slight contact with (for example, a highly contagious respiratory disease like measles), then a density-dependent transmission mode may be more appropriate because an infectious individual may transmit the condition to a large number of people in a crowded place.

To select the transmission by probability mode, the definition of the place type includes the following properties:

**place *PlaceType* {**

**enable\_density\_transmission\_for\_*COND* = 1**

**density\_contact\_prob\_for\_*COND* = *p***

**}**

For example,

**place Neighborhood {**

**enable\_density\_transmission\_for\_INF = 1**

**density\_contact\_prob\_for\_INF = 0.0001**

**}**

The effect is to use transmission by probability for the INF condition with the probability = 0.0001 of contact for the INF condition occuring between any two individuals in a neighborhood in a given hour of time in the neighborhood.

There is another form for the density contact probability:

**place *PlaceType* {**

**density\_contact\_prob = *p***

**}**

This form leaves off a condition name and the result is that the same contact probability is applied to any condition and place type for which density transmission is enabled. That is, in this case the density contact probability is considered to be an intrinsic property of the type of place, independent of the condition. The rationale for this approach is that the condition's transmissibility is taken into account separately as explained below. However, they may be models in which it desired to specify distinct density contact probablities for different conditions, so both forms are provided.

When using transmission by probability, the probability that a transmissible individual has a transmissible contact with each individual in a Place in a given hour is given by the formula:

**prob of contact = (transmissibility of Condition) \* (density\_contact\_prob of the Place)**

The value of **prob\_contact** is clipped at a maximum value of 1.0. It follows that an individual's probability of having a transmissible contact during a time block of **h** hours is set to:

**prob of individual exposure = 1.0 - (1.0 - prob of contact) ^ (number of transmissibles \* h)**

The number of potential transmission evens is then:

**number of potential transmission events = (prob of individual exposure) \* (number of individuals)**

For each such potential transmission event, a transmissible individual ***t*** and a potential host ***s*** are selected at random. If the agent ***s*** is present in the place, the probability that the transmission event succeeds is:

**probability of transmission = (transmissibility of agent *t*) \* (susceptibility of agent *s*) \* (same-age-bias)**

where the same-age-bias operates as explained previously.

See Chapter 13 for an example of density transmission in neighborhoods.

## Transmission in Networks

When agents are present in the same network, one agent may transmit a Condition to another agent to whom it is linked, according to the contact rate associated with that network type. Any Condition can be transmissible through networks by setting the following property:

**condition COND { transmission\_mode = network }**

A given network-transmissible Condition can by transmitted in a given type of Network by setting the property:

**network NET {** **can\_transmit\_*Condition* = 1 }**

For example, if a model includes both a dating network and a marriage network (with edges between spouses), a sexually transmitted disease Condition called STD might be transmitted in both networks by including the properties:

**network Dating { can\_transmit\_STD = 1 }**

**network Marriage { can\_transmit\_STD = 1 }**

The transmissibility of each network-transmissible Condition is controlled by Network transmission contact parameters:

**network NET { contact\_count\_for\_Condition = N** **}**(default N = 0)

**network NET { contact\_rate\_for\_Condition = R }** (default R = 0)

where **contact\_count** is the number of contacts between a transmissible individual and that agent's outgoing edges per each hour that the Network activity is simulated, and **contact\_rate** is the fraction of outgoing edge contacted per hour. If both are specified, **contact\_rate** overrides **contact\_count**.

The folowing property concerns how contacts for potential transmission are chosen, and will be explained below.

**network NET {deterministic\_contacts\_for\_*Condition* = 0/1 }** (default 1)

When transmissible agent has an edge to a susceptible agent in a Network, the transmissible agent may transmit a Condition to the susceptible agent. As in Places, transmission in a Network is a stochastic event and depends on several factors including:

* the transmissibility of the Condition
* the hourly contact rate for the Network
* random selections of which agents interact during a given hour
* the transmissibility of the transmissible agent
* the susceptibility of the susceptible agent.

The number of *transmission contacts per agent* in a Network depends on whether the user has specified contact counts or a contact rate of the network. If the Network uses a contact rate, the number of contacts for each transmissible agent is:

**contacts = (transmissibility of Condition)\*(transmissibility of agent)**

**\*(duration of the meeting in the Network)**

**\*(Network contact rate)\*(number of edges from the agent in the Network)**

If the Network uses a contact count, the number of contacts for each transmissible agent is:

**contacts = (transmissibility of Condition)\*(transmissibility of agent)**

**\*(duration of the meeting in the Network)\*(Network contact count)**

The duration of the meeting in the Network is set by the Network Schedule (see [Group Schedule](#_Group_Schedules)).

Each transmissible agent *t* makes the number of contacts computed above. For each such contact event, the agent selects another agent *s* that has an edge from the first agent. The probability that agent *t* successfully exposes the selected agent *s* is:

**probabaility of transmission = susceptibility of agent *s***

The property that determines the hourly contact rate is:

**contacts = *Number***

This property represents a fraction of the number of people linked to the given agent that will be contacted during each hour that the network is open. The actual number of contacts attempted will be:

**Contact count = (contacts \* out\_degree) \* current\_network\_open\_duration**

The default number of hourly contacts is 0.0. The **current\_network\_open\_duration** refers to the number of hours that the network is open starting at the current hour. For example, if we want to model weekly interactions within a network NetA, we would set the properties:

**network NetA {**

**is\_open\_at\_hour\_0\_on\_Sunday = 1**

**}**

This means that FRED only considers interactions in NetA once a week (at midnight on Sunday, but these values are somewhat arbitrary.)

You can also specify the maximum number of contacts per hour

**contact\_count = *Number***

The actual number of contacts attempted will be:

**Contact count = contact\_count \* current\_network\_open\_duration**

There several important details in this process:

* Agents do not contact themselves in this process. Each contact is assumed to be between two distinct agents.
* The agent *s* selected to contact may be selected more than once.
* If the **deterministic\_contacts** property is set for the Network, then contacts are selected without replacement, meaning that each edge is used once before any edge is used a second time. If the **determinisitic\_contacts** property is not set, then contacts are selected from the edges with replacement, meaning that some edges may be selected repeatedly and other edges not selected at all.
* The agent *s* is selected among all agents to whom the transmissible agent is linked, not just those currently attending the Network. In other words, it is possible that agent *s* is absent when selected. The assumption here is that the number of contacts per agent per Network is the nominal number of contacts if everyone is present. If many agents are temporarily absent from the Network, then the number of contacts is effectively reduced proportionally.
* The selected of agent *s* is independent of the susceptibililty, that is, the selected agent is not necessarily susceptible.
* There is no age-bias option in network transmission.

## Transmission of Places

FRED supports a form of transmission in which one agent transmits a place to another agent. One could think of this as sending an invitation to attend the first agent's place to the second agent.

To enable a Condition to transmit a place, use a property statement of the form

**condition COND { place\_type\_to\_transmit = *PlaceType* }**

If agent A is transmissible for this Condition and exposes the susceptible agent B, then agent B will join the Place of agent A. For example:

**condition PARTY {**

**place\_type\_to\_transmit = PartyHouse**

**transmission\_mode = proximity**

**transmissibility = 2.0**

**}**

**state PARTY.Host {**

**set\_trans(PARTY, 1.0)**

**}**

**state PARTY.PossibleGuest {**

**set\_sus(PARTY, 1.0)**

**}**

If agent A is in state PARTY.Host and agent B is in state **PARTY.PossibleGuest** and agents A and B share a proximity mixing group (if they attend the same school for example), then A may expose B to the PARTY Condition, causing agent B to join the PartyHouse where agent A is a member. If B is already a member of another PartyHouse, agent B would first end its membership in its PartyHouse before joining A's PartyHouse. If the place is already at its maximum size, then agent B does not join the place.

If a Condition transmits a place, then if the Import Agent exposed an ordinary agent to the Condition, a new place is generated based on the exposed agent's own location, and the exposed agent joins the new place as its first member. To extend the previous example, supoose the Import Agent reaches a state called GenerateParties in the PARTY Condition in which it exposes 10 susceptible agents via the rule:

**state PARTY. GenerateParties {**

**import\_count(10)**

**}**

This causes the Import Agent to select 10 susceptible agents at random and to generate a new PartyHouse place (at the household location of the exposed agent, by default). Each of the 10 exposed agents would become members of their respective PartyHouse places. The following property changes the location of the generated places:

**base\_type = *PlaceType2* [ default = Household ]**

where *PlayType2* is the type of place to base the generated place on. For example, if we want to generate each Party House based on the location of the exposed agent's workplace, we could use the property:

**place PartyHouse { base\_type = Workplace }**

## Cross-Condition Transmission

Sometimes it is desirable to model situation where a transmission event in one Condition results in the exposed agent becoming exposed to a second Condition. This property is set on a state by state basis:

**condition COND { State.condition\_to\_transmit = COND2 }**

where the default value of COND2 is COND1.

A typical example might be a model in which we have two Conditions, one for Violence and one for Injury. Suppose we have the following states for these two Conditions:

**condition VIOLENCE {**

**states = NonViolent Threatening Attack**

**Attack.condition\_to\_transmit = INJURY**

**transmission\_mode = proximity**

**}**

**condition INJURY {**

**states = None Injured Recovered**

**exposed\_state = Injured**

**}**

Suppose an agent A is transmissible for **VIOLENCE** and agent B is susceptible to **VIOLENCE**. If agent A is in the **Attack** state and transmits **VIOLENCE** to agent B, then agent B would become exposed to **INJURY** and enter the **Injured** state. Agent B would not change its state in the **VIOLENCE** Condition as a result of this event.

# Chapter 9: FRED Output

## Output Files

The FRED Core simulation produces a number of output files that include data generated from each simulation run, including:

* daily and weekly time-series statistics for every state defined in the model
* daily and weekly time-series statistics for every global variable defined in the model
* health records showing each individual agent's changes over time
* detailed records for selected individuals
* network files for each network defined in the model
* a log file reporting internal messages from the FRED software
* error and warnings

The user controls how much detail is included in most of these files using options that will be explained in the sections below.

If the user is accessing FRED through the FRED Web Interface, these files are all stored in the FRED Server and can be downloaded through the Web Interface to the user's local computer.

If the user is running the command line version of FRED using the **fred\_job** command, these files are stored in the directory **~/FRED/RESULTS/JOB/<ID>** where **<ID>** is the unique id number associated with the job. This directory also contains the meta-data for the job such as the time of the job and the FRED Version used. The specific output files for run **N** of the job appear in **~/FRED/RESULTS/JOB/<ID>/OUT/RUN<N>.** When the user is developing a model with the command line version of FRED, it is typical to run tests in the user's working directory using the **fred\_run** or **run\_fred** commands (both identical). In this case, the files are contained in the subdirectory **OUT/RUN<N>.**

## Time Series Output

FRED output includes the daily time-series for every state defined in the model. The user can access these files either through the [FRED Web interface](#_Chapter_15:_FRED) or by using the command line interface discussed here.

We illustrate the FRED output files using the following FRED program call test.fred:

**use FRED::Influenza**

**start\_date = 2020-Jan-01**

**end\_date = 2020-Jan-05**

**locations = Jefferson\_County\_PA**

Suppose we run a job consisting of 4 runs using this program:

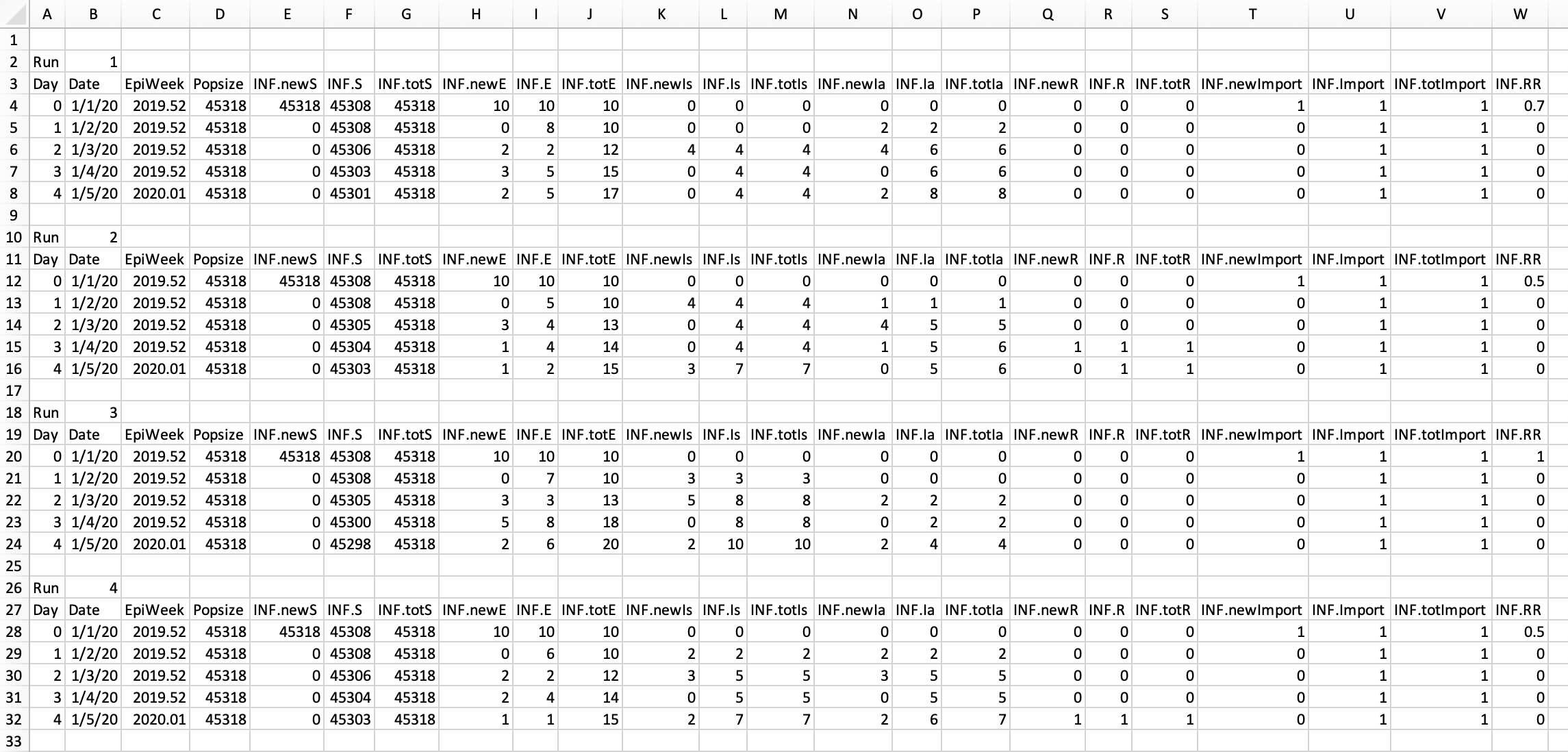
% fred\_job -k test -p test.fred -n 4

This command requests the FRED Simulation Information Management System (FRED SIMS) to create a new job with **key =** **test** and execute the program **test.fred** four times. FRED SIMS stores all the files associated with this job in a directory dedicated to this job.

One output file is a comma-separated-value file (csv file) that can be opened by standard spreadsheet applications. To access the file use the fred\_csv command:

% fred\_csv -k test > test.csv

The file **test.csv** contains the daily values of all the variables in the model (in this case, the model in the FRED Library module FRED::Influenza). This file can be opened with your favorite spreadsheet program and it looks like this:



The file contains one line of data for each day of each run. The columns show the name of the states defined in the model, as well as simulation day, date, epi-week and population size.

The FRED::Influenza model includes the Condition INF and its states: S, E, Ia, Is, R and Import. For each state, FRED records three values for each day:

* the prevalance (that is, number of individuals in the population who are currently in the given state) labeled with the state name (e.g. INF.S above);
* the cumulative count (that is, the total number of individuals who have ever been in the given state) labeled COND.totSTATE (e.g., INF.totS above), and
* the daily incidence (that is, the number of individuals entering this state on the given day) labeled COND.newState (e.g., INF.newS above).

There are two variations of the fred\_csv command. If you only want the data for a given run, you can pick the run number with the -n option:

% fred\_csv -k test -n 3

Run,3

Day,Date,EpiWeek,Popsize,INF.newS,INF.S,INF.totS,INF.newE,INF.E,INF.totE,INF.newIs,INF.Is,INF.totIs,INF.newIa,INF.Ia,INF.totIa,INF.newR,INF.R,INF.totR,INF.newImport,INF.Import,INF.totImport,INF.RR

0,2020-01,01,2019.52,45318,45318,45308,45318,10,10,10,0,0,0,0,0,0,0,0,0,1,1,1,1.000000

1,2020-01-02,2019.52,45318,0,45308,45318,0,7,10,3,3,3,0,0,0,0,0,0,0,1,1,0.000000

2,2020-01-03,2019.52,45318,0,45305,45318,3,3,13,5,8,8,2,2,2,0,0,0,0,1,1,0.000000

3,2020-01-04,2019.52,45318,0,45300,45318,5,8,18,0,8,8,0,2,2,0,0,0,0,1,1,0.000000

4,2020-01-05,2020.01,45318,0,45298,45318,2,6,20,2,10,10,2,4,4,0,0,0,0,1,1,0.000000

If you want the combined data for a single column, you can specify the column header using the -v option:

% fred\_csv -k test -v INF.totE > totE.csv

% open totE.csv

A screenshot of a cell phone

Description automatically generated

This table includes statistics about the specified state for each simulation day over all the runs in the job. For each day, the row includes the minimum value, the first quartile, the median, the third quartile, the maximum value, the mean and standard deviation, and the individual values in each run.

FRED also produces time-series data for all global variables defined in the FRED program. For example, suppose the model is keeping track of the number of drug prescriptions over time, and includes a global variable:

Global Prescriptions

Prescriptions = 0

This variable might be updated by each agent who received a prescription:

State(HealthCare, ReceivedPrescription) {

set(Prescriptions, Prescription+1)

}

The time series for a global variable named *X* is stored under the file name **FRED.*X***, so the plot for the Prescription variables can by produced by the command:

% fred\_plot -k <job\_name> -v FRED.Prescriptions

Note: FRED does not currently prodcues output files for global list variables.

## Health Records File

If the property

**enable\_health\_records = 1**

is set, the FRED creates a **health records file** that contains individual level information that reflects every time an agent changes state.

The health records file may be quite large, so by default the file is created only for the first run of a job that includes multiple simulation runs. The user can control which run has a health records file by setting the property:

**health\_records\_run = N**

to get the file for run N (N = 1 by default). If N = -1, health record files are produced for all runs. **Warning:** *This may results in a large amount of data.*

The program controls which Conditions to include in the health records file by property statements of the form:

**COND.enable\_health\_records = 1**

By default, no Condition is included in the file.

If the following statement appears in the FRED program:

**record\_location = 1**

then FRED includes the latitude and longitude of the agent on each line in the **health records file**.

As an example, suppose you run the example Pandemic Influenza program in Chapter 14 with the additional program statements:

**enable\_health\_records = 1**

**INF.enable\_health\_records = 1**

The first line tells FRED to create a health records file (for run 1), and the second line tells FRED to include a record each time an agent changes state in its INF Condition.

This example shows the command-line use of FRED, but you can also obtain the resulting files through the Web Interface.

Suppose the FRED program is in a file called **inf.fred.** First run a FRED job with the key (or job name) **test:**

% fred\_job -k test -p inf.fred

After running the job, the health records file includes records like the following:

**HEALTH RECORD: 2020-01-01 12am day 0 person 39790 age 20 is an IMPORTED EXPOSURE to INF**

**HEALTH RECORD: 2020-01-01 12am day 0 person 39790 age 20 sex F race 1 household H-11047265 school NONE income 49200 CONDITION INF CHANGES from S to E**

**HEALTH RECORD: 2020-01-01 12am day 0 person 43509 age 66 is an IMPORTED EXPOSURE to INF**

**HEALTH RECORD: 2020-01-01 12am day 0 person 43509 age 66 sex M race 1 household H-11046512 school NONE income 40000 CONDITION INF CHANGES from S to E**

You can get the whole file with the command:

% fred\_get\_records -k test > records.txt

This copies the health\_records file to a local file called **records.txt.**

It is often useful to see all the records for a given individual. The following command returns all the records for person 39790:

% fred\_get\_records -p 39790

**HEALTH RECORD: 2020-01-01 12am day 0 person 39790 age 20 is an IMPORTED EXPOSURE to INF**

**HEALTH RECORD: 2020-01-01 12am day 0 person 39790 age 20 sex F race 1 household H-11047265**

**school NONE income 49200 CONDITION INF CHANGES from S to E**

**HEALTH RECORD: 2020-01-03 5am day 2 person 39790 age 20 sex F race 1 household H-11047265**

**school NONE income 49200 CONDITION INF CHANGES from E to Is**

**HEALTH RECORD: 2020-01-03 5am day 2 person 39790 ENTERING state INF.Is MODIFIES state StayHome.No**

**to StayHome.Symptoms**

**HEALTH RECORD: 2020-01-12 9am day 11 person 39790 age 20 sex F race 1 household H-11047265**

**school NONE income 49200 CONDITION INF CHANGES from Is to R**

**HEALTH RECORD: 2020-01-12 9am day 11 person 39790 ENTERING state INF.R MODIFIES state StayHome.Yes to StayHome.No**

This showing the complete path of this individual through the states defined in the model.

To see all the records showing every agent that changes into a given state:

% fred\_get\_records -s Is

**fred\_get\_records -k test -s Is | head -10**

**HEALTH RECORD: 2020-01-02 7am day 1 person 18142 age 1 sex F race 1 household H-11045344**

**school NONE income 1000 CONDITION INF CHANGES from E to Is**

**HEALTH RECORD: 2020-01-02 12pm day 1 person 43509 age 66 sex M race 1 household H-11046512**

**school NONE income 40000 CONDITION INF CHANGES from E to Is**

**HEALTH RECORD: 2020-01-03 12am day 2 person 38779 age 28 sex F race 1 household H-11051097**

**school NONE income 78000 CONDITION INF CHANGES from E to Is**

**HEALTH RECORD: 2020-01-03 12am day 2 person 45113 age 18 sex F race -1 household GH-450004650-016 school NONE income 38900 CONDITION INF CHANGES from E to Is**

**HEALTH RECORD: 2020-01-03 1am day 2 person 35833 age 67 sex F race 1 household H-11059471**

**school NONE income 49400 CONDITION INF CHANGES from E to Is**

**...**

This shows the times and dates that any individuals changed from state E to state Is in this run.

To see all the records that match a given pattern in the file, using the -g option (meaning **grep**):

% fred\_get\_records -g 'IMPORTED'

**HEALTH RECORD: 2020-01-01 12am day 0 person 39790 age 20 is an IMPORTED EXPOSURE to INF**

**HEALTH RECORD: 2020-01-01 12am day 0 person 43509 age 66 is an IMPORTED EXPOSURE to INF**

**HEALTH RECORD: 2020-01-01 12am day 0 person 38779 age 28 is an IMPORTED EXPOSURE to INF**

**HEALTH RECORD: 2020-01-01 12am day 0 person 16183 age 28 is an IMPORTED EXPOSURE to INF**

**HEALTH RECORD: 2020-01-01 12am day 0 person 45113 age 18 is an IMPORTED EXPOSURE to INF**

**HEALTH RECORD: 2020-01-01 12am day 0 person 18354 age 23 is an IMPORTED EXPOSURE to INF**

**HEALTH RECORD: 2020-01-01 12am day 0 person 18142 age 1 is an IMPORTED EXPOSURE to INF**

**HEALTH RECORD: 2020-01-01 12am day 0 person 11651 age 30 is an IMPORTED EXPOSURE to INF**

**HEALTH RECORD: 2020-01-01 12am day 0 person 39085 age 9 is an IMPORTED EXPOSURE to INF**

**HEALTH RECORD: 2020-01-01 12am day 0 person 35833 age 67 is an IMPORTED EXPOSURE to INF**

This output verifies that 10 individuals were exposed to INF by the Import Agent at the start of the simulation run.

## Individual Report Files

Coming Soon.

## Network Files

If the model includes networks, the FRED program will produce output files for a given network if the program includes a property statement of the form:

***Network****.***print\_interval = N**

If N=0 (the default), no output file is produced for the given Network. If N > 0, output files are produced after every N days of the simulation. If N > 0, network files are always produced for first and the last days of the simulation.

For each day that a network is output, two files are produced:

One file is called **<*Network*>-<*Day*>.vna** (for example, **MyNet-1.vna)** and is in the **VNA** format. This text file contains lines of the form:

**\*node data**

**ID age sex race**

**39790 20 F 1**

**43509 66 M 1**

**...**

**\*tie data**

**from to weight**

**43509 43508 1.000000**

**43509 35391 1.000000**

**...**

The first section contains one line for each agent in the network, giving the agent's id, age, sex, and race. The second section has one line for each edge in the network, showing the agent, the agent it is connected to, and the weight of the edge.

The second is called **<*Network*>-<*Day*>.txt** (for example, **MyNet-1.txt)**. This text file contains FRED program statement of the form:

**INFtrans.add\_edge = 43509 43508 1.000000**

**INFtrans.add\_edge = 43509 35391 1.000000**

These lines can be included verbatim in a FRED program to define the edges in the network.

As an example, suppose you run the example Pandemic Influenza program in Chapter 14 with the additional program statements:

**INF.transmission\_network = INFtrans**

**INFtrans.print\_interval = 5**

The first line above tells FRED to generate a network called INFtrans which contains one node for each infectious agent and one edge between the infectious agent all those agents exposed by the first agent. The second line tells FRED to output network files every 5 days. (Another option would be to pick a larger number, say 1000. In that case, since the simulation only lasts 120 days, FRED will produce network files for only the first day and the last day of the simulation.)

After running the job, we get the following files for day 0. The show the 10 exposures from the Import Agent (represented by id -1 here):

**INFtrans-0.vna:**

**\*node data**

**ID age sex race**

**39790 20 F 1**

**-1 0 M -1**

**43509 66 M 1**

**38779 28 F 1**

**16183 28 M 1**

**45113 18 F -1**

**18354 23 M 1**

**18142 1 F 1**

**11651 30 F 1**

**39085 9 M 1**

**35833 67 F 1**

**\*tie data**

**from to weight**

**-1 39790 1.000000**

**-1 43509 1.000000**

**-1 38779 1.000000**

**-1 16183 1.000000**

**-1 45113 1.000000**

**-1 18354 1.000000**

**-1 18142 1.000000**

**-1 11651 1.000000**

**-1 39085 1.000000**

**-1 35833 1.000000**

**INFtrans-0.txt:**

**INFtrans.add\_edge = -1 39790 1.000000**

**INFtrans.add\_edge = -1 43509 1.000000**

**INFtrans.add\_edge = -1 38779 1.000000**

**INFtrans.add\_edge = -1 16183 1.000000**

**INFtrans.add\_edge = -1 45113 1.000000**

**INFtrans.add\_edge = -1 18354 1.000000**

**INFtrans.add\_edge = -1 18142 1.000000**

**INFtrans.add\_edge = -1 11651 1.000000**

**INFtrans.add\_edge = -1 39085 1.000000**

**INFtrans.add\_edge = -1 35833 1.000000**

After 5 days we get the following files, whick show which agents have exposed other agents within the first 5 days of the simulation:

**INFtrans-5.vna:**

**\*node data**

**ID age sex race**

**39790 20 F 1**

**-1 0 M -1**

**43509 66 M 1**

**38779 28 F 1**

**16183 28 M 1**

**45113 18 F -1**

**18354 24 M 1**

**18142 1 F 1**

**11651 30 F 1**

**39085 9 M 1**

**35833 67 F 1**

**43508 61 F 1**

**38548 30 F 1**

**35391 70 M 1**

**33993 36 M 1**

**16424 27 M 1**

**4798 29 M 1**

**39789 56 F 1**

**28027 34 F 1**

**41925 35 M 1**

**28636 70 F 1**

**\*tie data**

**from to weight**

**39790 39789 1.000000**

**-1 39790 1.000000**

**-1 43509 1.000000**

**-1 38779 1.000000**

**-1 16183 1.000000**

**-1 45113 1.000000**

**-1 18354 1.000000**

**-1 18142 1.000000**

**-1 11651 1.000000**

**-1 39085 1.000000**

**-1 35833 1.000000**

**43509 43508 1.000000**

**43509 35391 1.000000**

**43509 28636 1.000000**

**38779 16424 1.000000**

**38779 28027 1.000000**

**16183 38548 1.000000**

**11651 33993 1.000000**

**11651 4798 1.000000**

**43508 41925 1.000000**

**INFtrans-5.txt:**

**INFtrans.add\_edge = 39790 39789 1.000000**

**INFtrans.add\_edge = -1 39790 1.000000**

**INFtrans.add\_edge = -1 43509 1.000000**

**INFtrans.add\_edge = -1 38779 1.000000**

**INFtrans.add\_edge = -1 16183 1.000000**

**INFtrans.add\_edge = -1 45113 1.000000**

**INFtrans.add\_edge = -1 18354 1.000000**

**INFtrans.add\_edge = -1 18142 1.000000**

**INFtrans.add\_edge = -1 11651 1.000000**

**INFtrans.add\_edge = -1 39085 1.000000**

**INFtrans.add\_edge = -1 35833 1.000000**

**INFtrans.add\_edge = 43509 43508 1.000000**

**INFtrans.add\_edge = 43509 35391 1.000000**

**INFtrans.add\_edge = 43509 28636 1.000000**

**INFtrans.add\_edge = 38779 16424 1.000000**

**INFtrans.add\_edge = 38779 28027 1.000000**

**INFtrans.add\_edge = 16183 38548 1.000000**

**INFtrans.add\_edge = 11651 33993 1.000000**

**INFtrans.add\_edge = 11651 4798 1.000000**

**INFtrans.add\_edge = 43508 41925 1.000000**

### Gelphi for Network Display and Analysis

The VNA files can be used to create network diagram using a number of tools, including Gelphi (https://gephi.org). An example of a display of the day 5 network using Gelphi is shown below:

A close up of a necklace

Description automatically generated

The Gelphi system lets you interrogate each node and edge to see its' id and other properties and provides numerous other network analysis tools.

## The LOG File

The FRED LOG file contains internal messages and progress reports from the FRED Core software. It is often useful to examine the LOG when debugging the FRED software itself. The LOG file is not intended for the casual user.

## Errors and Warnings

The FRED Compiler analyzed the FRED program and reports any Errors or Warnings. Errors are fatal and prevent the programs from running. Examples of Errors include:

* Missing simulation location
* Missing simulation start and end dates
* Missing wait rules for states
* Syntax errors in rules, such as mis-matched parantheses
* Unrecognized expressions or actions

Warnings are for the user's information and do not represent fatal problems with the program. Examples of warnings include:

* Redundant property statements. If a property is set more than once, the last one in the program takes effect, but the previous statements are reported in a warning.
* Redundant rules, such as more than one wait rule for a given state. The last one in the program takes precedence.
* References to undefined Conditions or States. These may represent typos, or they may refer to Conditions or States that the user has excluded from the current model.

To see errors and warnings at the command-line level, use the command:

**% fred\_compile -p <fred-program>**

On the Web Interface, if you submit a program that generates errors or warnings, they are displayed on the job submission page. If any errors are reported, the job will not be executed.

A complete list of error and warning messages is in Appendix B.

## Plotting and Visualization

The previous section shows how to generate FRED output files for further analysis, such as creating plots of the time-series values associated with the state in the model. For the convenience of the user, FRED provides a built-in plotting facility. Plots can be requested on the FRED Web Project page, but here we describe how to generate plots using the command line interface.

### Dormant States

Dormant states are absorbing states that FRED no longer needs to track when visualization (Reporting) is requested. The declaration is:

**COND.State.is\_dormant = 1**

This statement tells FRED not to record the location of an agent once it enters the given state. It doesn't not affect the simulation results, but it saves significant computation time and data storage when visualization is being performed.

# Chapter 10: FRED Simulation Information Management System

FRED includes a command-line interace to the FRED Simulation Information Management System (FRED SIMS) that manages the workflow and data associated with a user’s FRED project. Using FRED SIMS helps to ensure that FRED projects are reproducible.

This Chapter describes commands associated with FRED SIMS. For users of FRED Web, this all occurs behind the scenes. For command-line FRED users, the commands described here are used to maintain your local database of FRED experiments and their results.

**NOTE:** Throughout this document, “%” refers to the Unix system prompt and is not part of the typed command.

## Running FRED directly

Usually you will want to use one of the job control commands described below for running a large number of simulations with FRED, but here we show how to run FRED directly from the command line. The FRED program takes several optional command line arguments, including the path to and name of the FRED progran file:

**% FRED -p program\_file\_name**

If the argument is omitted the file “**model.fred**” in the current working directory is assumed. The program file must exist or an error occurs.

By default, FRED will create a directory called OUT and write a report to a file called OUT/RUN1/out1.csv. Log messages will be printed on the screen, but can be saved to a file:

**% FRED -p program\_file\_name > LOG.txt**

### Example: running FRED

Use a text editor to create a file called **model.fred** that contains the lines:

**##########################**

**# File: model.fred**

**# Include a simple model of pandemic influenza**

**use FRED::Influenza**

**# Simulated Location**

**locations = Jefferson\_County\_PA**

**# Simulated Time Frame**

**start\_date = 2020-Jan-01**

**end\_date = 2020-Apr-30**

**############################**

This program tells FRED to simulate an influenza epidemic in Jefferson County, PA for given time period. The Influenza model will be read from the file

**FRED/library/Influenza/Influenza.fred**

To run FRED, use the command

**% FRED > LOG.txt**

Open the file OUT/RUN1/out.csv in a text editor or spreadsheet program to examine it. The file LOG.txt contains log messages that show FRED in action. The LOG file is often helpful if errors occur.

## 

## FRED command line options

The FRED program takes several optional command line options:

**% FRED [-p program\_file] [-r run\_number] [-d directory] [-c ]**

All the arguments are optional.

If the optional arguments are given, FRED reads the user-provided program\_file, runs a single simulation with number run\_number, and writes its output files to the given directory. The output directory is relative to the current working directory.

If **-p** is ommitted, program\_file defaults to **model.fred**.

If **-r** is ommitted, run\_number defaults to 1.

If the **-d** is omitted, the output directory defaults to **OUT**.

The **-c** option is used by the FRED compiler as explained in the next section.

## The FRED Compiler

If the **-c** argument is provided on the FRED command line, FRED compiles the program and prints errors and warnings, respectively, but no simulation is performed. The preferred way to invoke the compiler is by the following command:

**% fred\_compile -p program**

This runs the compiler on the named program and prints errors and warnings.

The compiler performs the following checks:

* The following required elements must be defined or an error is flagged:
  + The Simulated Location
  + The Simulated Time Frame
  + At least one Condition must be defined
* Any Property statement that appears in the program but is not used is flagged as a warning. Ignored property statements can occur for a number of reasons, but the most common reason is that the property is misspelled. Properties may also be ignored if they are overridden by another property statement later in the program. In any case, the user should examine the warnings carefully and correct the problem if necessary.
* Any ill-formed Rule statement results in an error. The user must correct the error before proceeding.
* Any ignored Rules (because they are overridden or do not apply to the selected Conditions) are reported as warnings.

The user should always call fred\_compile after making changes to a FRED program. The compiler attempts to provide helpful messages whenever possible.

The FRED compiler is also called whenever the command run\_fred or fred\_job is invoked. If errors are found, the user must fix the errors before the simulation will run.

## FRED commands

FRED provides several **commands** that facilitate performing several common simulation tasks. FRED commands generally have the form fred\_xxx.

#### fred\_help

Use fred\_help to list all the available FRED commands:

% fred\_help  
This is the fred\_help command.  
  
To get help for specific topics, try:  
 fred\_help topic  
  
To get help for any specific command, try:  
 <fred\_command> --help  
  
Some important fred commands are:  
  
## Documentation  
 fred\_help -- this program  
 fred\_param <string> -- search for parameters that contain given string  
  
## Setting up simulation work flows  
 fred\_set  
 fred\_calibrate  
 fred\_get\_fips  
 fred\_make\_params  
 fred\_make\_qsub  
 fred\_make\_tasks  
 fred\_make\_rt  
 fred\_rt  
  
## Job control  
 run\_fred  
  
 fred\_delete  
 fred\_job  
 fred\_jobs  
 fred\_sa  
  
## Retrieving information from a FRED job  
 fred\_find  
 fred\_log  
 fred\_status  
 fred\_AR  
 fred\_CAR  
 fred\_R0  
 fred\_cd  
 fred\_csv  
 fred\_density  
 fred\_get\_places  
 fred\_id  
 fred\_peak  
 fred\_plot  
 fred\_tail

fred\_clear\_all\_results  
  
## Making Reports and movies  
 fred\_make\_Report  
 fred\_make\_Reports  
 fred\_make\_movie

#### run\_fred

The run\_fred command performs multiple realizations (runs) in a local directory. Each run uses a distinct seed for the random number generator, so the results will vary from run to run. The format is:

**% run\_fred -p program -d directory -s start\_run -n end\_run**

The order of the arguments doesn’t matter, and all arguments have default values:

p="model.fred"  
d=""  
s=1  
n=1

So

**% run\_fred -n 3**

is the same as:

**% FRED -p model.fred -r 1 -d OUT > OUT/RUN1/LOG**

**% FRED -p model.fred -r 2 -d OUT > OUT/RUN2/LOG**

**% FRED -p model.fred -r 3 -d OUT > OUT/RUN3/LOG**

The random seed for each run is set based on the both the seed value in the params file and on the run number, so a collection of FRED runs can be executed in any order with the same results.

*The run\_fred command automatically runs the FRED compiler and reports any errors or warnings. If any error occurs, the program is terminated.*

**NOTE:** You will not normally use run\_fred directly; instead, you will use the fred\_job command (described next) which uses run\_fred and also creates additional files that help you manage your work. But it is a good idea to use run\_fred when initially developing or debugging models. If you have errors, check the LOG files in the OUT directory.

## Job Control

#### fred\_job

A FRED **job** is a set of simulations that use the same parameters, except for the initial seed for the random number generator. Running multiple replications permits statistical analysis. To start a FRED job, use the command

fred\_job [options]

where options include:

-h -- print help  
 -k key -- assign job the specified key  
 -p program -- use named program file  
 -n number\_of\_runs -- number of runs in the job

*The fred\_job command automatically runs the FRED compiler and reports any errors or warnings. If any error occurs, the program is terminated.*

All the data associated with a job is stored in a database called $FRED\_HOME/RESULTS. You can manage the database using these commands:

fred\_jobs -- lists all jobs in the RESULTS database  
 fred\_clear\_all\_results -- delete all jobs in the RESULTS database  
 fred\_delete -k key -- delete the job with given key

**Example using fred\_job**

The fred\_job command is used to create a FRED **job**, consisting of one or more FRED simulations that use the same parameters. FRED jobs are identified by a **key** that the user selects to name the job. The following command will create a FRED job with key “test” that includes 5 runs of FRED with the parameters specified in the text file **program**:

% fred\_job -k test -p program -n 5 > test.out &

The “&” symbol runs the job in the background, so you can continue working in the terminal.

While the job is running, you can see its progress using the following commands:

fred\_jobs -- lists all FRED jobs along with their run status  
fred\_status -k key -- print the status of job  
fred\_log -k key -- print the last few lines of the log file  
fred\_find -k key -- print the location of the job files

#### fred\_status

fred\_status prints to current status of the job:

% fred\_status -k test  
FINISHED Wed Sep 9 12:48:38 2015

#### fred\_log

fred\_log prints the last few lines of the log file:

% fred\_log -k test  
STARTED: Wed Sep 9 12:48:32 2015  
FINISHED: Wed Sep 9 12:48:38 2015  
tail LOG1:  
  
day 99 REPORT population took 0.000067 seconds  
day 99 maxrss 79015936  
day 99 finished Wed Sep 9 12:48:34 2015  
DAY\_TIMER day 99 took 0.000161 seconds  
  
  
FRED simulation complete. Excluding initialization, 100 days took 0.290909 seconds  
FRED finished Wed Sep 9 12:48:34 2015  
FRED took 1.110621 seconds

#### fred\_find

fred\_find shows the location of the directory containing the data for the job:

% fred\_find -k test  
/Users/gref/FRED/RESULTS/JOB/6096

#### fred\_delete

Once a job is created, you cannot create the same job again. If you try, you get an error message:

% fred\_job -k test -p program -n 5  
 fred\_job: key test already used.

To delete the job, use:

% fred\_delete -k test  
or  
 % fred\_delete -f -k test

The latter version will delete the job without further prompts, so it assumes you know what you’re doing.

# Chapter 11: FRED Web Interface

Coming Soon.

# Chapter 12: The FRED Library

FRED includes an expanding Library of *modules* that modelers can incorporate into their FRED program to quickly build complex models.

## Learning about the FRED Library

If you are using FRED Web, a list of available Modules is available on the Library page. From the command line, you can print the contents of the FRED Library by the following command

% fred\_library --list

Aging

Asthma

Influenza

Mortality

Maternity

Each Module includes both a FRED program segment that can be directly included in the user's program and a README file that summarizes the contents of the Module. To see the documentation, use the command:

% fred\_library --doc *Module* (or -d *Module*)

For example:

% fred\_library -d Influenza

**FRED Module FRED::Influenza**

**Author: John Grefenstette**

**Created: 14 Apr 2019**

**Condition: INF**

**States: S E Is Ia R Import**

**Summary: A simple model of pandemic influenza in a modifed S-E-I-R model with rules:**

**1. All individuals start in the susceptible state S.**

**2. An individual enters state E when exposed by another infectious individual or by importation.**

**3. The latent period (state E) lasts between about 1.5 and 3.5 days (lognormal(1.9,1.23)) after which about 67% of exposed individuals become infectious with symptoms (state Is) and the rest become infectious while asymptomatic (state Ia).**

**4. Individuals with symptoms are twice as infectious as those who are asymptomatic.**

**5. Individuals with symptoms have a 50% chance of household confinement for the duration of thie illness.**

**5. The infectious period lasts between about 3.3 and 7.5 days (lognormal(5.0,1.5) distribution).**

**6. After the infectious period, all individuals recover (state R) and remain immune for the remainder of the simulation.**

As shown in this example, the documentation includes the name of the Module, the Condition(s) it defines, the States involved, and a narrative covering the operation of the Rules declared within the module.

To see the complete FRED program segment implementing the Module, use the command:

% fred\_library --prog *Module* (or -p *Module*)

For example,

**% fred\_library --prog Influenza**

**#####################################################**

**# MODULE FRED::Influenza**

**# Author: John Grefenstette**

**# Date: 22 Jul 2019**

**condition INF {**

**states = S E Is Ia R Import**

**import\_start\_state = Import**

**transmission\_mode = proximity**

**transmissibility = 1.0**

**R0\_a = 0.0398238**

**R0\_b = 0.611043**

**S.is\_dormant = 1**

**R.is\_dormant = 1**

**exposed\_state = E**

**}**

**state INF.S {**

**set\_sus(INF,1)**

**wait()**

**next()**

**}**

**state INF.E {**

**set\_sus(INF,0)**

**wait(24\*lognormal(1.9,1.23))**

**next(Is) with prob(0.67)**

**next(Ia) with prob(0.33)**

**}**

**state INF.Is {**

**set\_trans(INF,1)**

**set\_state(StayHome,No,Symptoms)**

**wait(24\* lognormal(5.0,1.5))**

**next(R)**

**}**

**state INF.Ia {**

**set\_trans(INF,0.5)**

**wait( 24 \* lognormal(5.0, 1.5) )**

**next(R)**

**}**

**state INF.R {**

**set\_trans(INF,0)**

**set\_state(StayHome,Yes,No)**

**wait()**

**next()**

**}**

**state INF.Import {**

**import\_count(10)**

**wait()**

**next()**

**}**

**condition StayHome {**

**states = No Symptoms Yes**

**}**

**state StayHome,No {}**

**state StayHome.Symptoms {**

**wait(0)**

**next(Yes) with prob(0.5)**

**default(No)**

**}**

**state StayHome.Yes {**

**absent()**

**present(Household)**

**wait()**

**next()**

**}**

## Using a FRED Library

If you are using FRED Web, you can select modules from the **Library page.** To include a FRED Module into a FRED program, add a statement like:

**use FRED::*Module***

This statement has the effect of insert the module's program segment into the FRED program at the point of the statement.

It is recommended that you include the library modules near the start of the FRED program. This makes it convenient to include additional statements that modify the statements in the library module. For example, the Influenza Module above includes a statement that generates 10 external exposures on day 0:

**INF.Import.import\_max\_cases = 10**

If you wanted to change this to 20 cases for your own FRED program, the entire FRED program might look like this:

**use FRED::Influenza**

**INF.Import.import\_max\_cases = 20**

**start\_date = 2020-Jan-01**

**end\_date = 2020-Apr-01**

**locations = Jefferson\_County\_PA**

A FRED program may include any number of Modules from the Library.

## Conditions Defined in Modules

Each Module in the Library includes one or more code blocks of the form:

**Condition COND { … }**

These statements are cumulative, so that the final program includes all the Conditions that are defined by the modules that are included in the program. The order of the Conditions is the same as the order in which the modules are included. If the user wishes to override this, the FRED program can exclude a Condition explicitly:

**exclude\_Condition = COND**

For example if the FRED program includes several modules, such as:

**use FRED:ModA # with Condition A**

**use FRED:ModB # with Condition B**

**use FRED:ModCD # with Condition C and Condition D**

The user could select to use only Conditions A, B and D by declaring:

**exclude\_Condition = C**

after the modules are included.

## The FRED Community

The Library can also include contributed sub-models from the FRED user community. More details on how to contribute your models to the Library will be announced as the become available. The hope is to build an active community of FRED users that decrease the efforts of the entire modeling community.

# Chapter 13: Programming Tips

This Chapter contains a series of programming examples. Some are complete programs and some are snippets, small section of code that show how specific features work.

## S-E-I-R Model

This snippet shows how we define a disease model using a slight variation of the standard **S-E-I-R** model. In the standard model, the states represent **Susceptible (S), Exposed (E), Infectious (I),** and **Recovered (R).** In this variation, we split the **I** state into **Infectious with Symptoms (Is**) and **Infectious but Asymptomatic (Ia).**

This is the version used in the built-in the Influenza Condition in FRED.

**Condition INF {**

**states = S E Is Ia R Import**

This disease will have cases introduced to the population by the Import Agent:

**import\_start\_state = Import**

This disease is a respiratory disease that is transmissible by proximity:

**transmission\_mode = proximity**

When an agent is exposed, it moves to state **E:**

**exposed\_state = E**

We set the transmissibility to 1.0, which was calibrated to produce an influenza-like spread within the population:

**transmissibility = 1.0**

**}**

Individual agents are all susceptible at the start. They stay susceptible until exposed.

**state INF.S {**

**set\_sus(INF,1)**

**wait()**

**next()**

**}**

When an agent is exposed, it loses susceptibility, and remains in this state for a median of 1.9 days, with a lognormal distribution of latency. The exposed agent then becomes infectious. The probablity of symtpoms is 0.67:

**state INF.E {**

**set\_sus(INF,0)**

**wait(24\*lognormal(1.9,1.23))**

**next(Is) with prob(0.67)**

**next(Ia) with prob(0.33)**

**}**

The agent becomes in infectious and remains infectious for a period following a lognormal distribution with most agents remaining infectious for 3-7 days with a median of 5 days. The agent then enters the recovery state.

**state INF.Is {**

**set\_trans(INF,1)**

**wait(24\* lognormal(5.0,1.5))**

**next(R)**

**}**

We assume that asymptomatic agents are half as infectious as agents with symptoms:

**state INF.Ia {**

**set\_trans(INF,0.5)**

**wait(24\* lognormal(5.0,1.5))**

**next(R)**

**}**

When the agent recovers, it loses transmissibility and remains recovered indefinitely.

**state INF.R {**

**set\_trans(INF,0)**

**wait()**

**next()**

**}**

## Changes in Personal Behavior Based on Symptoms

This snippet illustrates how to have each agent change its social mixing behavior based on having symptoms of some disease.

The approach in this example is to declare a Condition called **StayHome** with states **Symptoms, Yes, and No.** The agent enters the **Symtpoms** state when an illness occurs. In this state, the agent decides whether to stay home or to continue its usual activity schedule. If the agent decides to stay home, it is temporarily absent from all mixing groups except the household itself.

The agent starts in the **No** state, meaning that it follows its usual schedule. By default, the agent remains in the **No** state indefinitely.

**# AGENT BEHAVIOR BASED ON SYMPTOMS**

**condition StayHome {**

**states = No Symptoms Yes**

**}**

**State(StayHome,No) {**

**present()**

**wait()**

**next()**

**}**

**State(StayHome,Symptoms) {**

**wait(0)**

**next(Yes) with prob(0.5)**

**default(No)**

**}**

**State(StayHome,Yes) {**

**absent()**

**present(Household)**

**wait()**

**next()**

**}**

Now we can add rules that reflect the side effects of entering specific states, If the agent enters a disease state that represents having symptoms, the agent changes states to **Symptoms** in the **StayHome** Condition.

**state INF.Is { set\_state(StayHome,No,Symptoms) }**

If the agent enters a disease state that represents the end of symptoms, then the agent resumes its normal activities.

**state INF.R) { set\_state(StayHome,Yes,No) }**

## Measuring Where Transmissions Occur

The following condition reports the place type of each infected for the INF condition:

**###############################################**

**# A condition for reporting where a transmission occurs**

**condition WHERE {**

**states = None H N S C W O X**

**}**

**# the following rule a moves each infected agent to the appropriate**

**# reporting state**

**state INF.E {**

**if (exposed\_in(INF,Household)) then set\_state(WHERE,H)**

**if (exposed\_in(INF,Neighborhood)) then set\_state(WHERE,N)**

**if (exposed\_in(INF,School)) then set\_state(WHERE,S)**

**if (exposed\_in(INF,Classroom)) then set\_state(WHERE,C)**

**if (exposed\_in(INF,Workplace)) then set\_state(WHERE,W)**

**if (exposed\_in(INF,Office)) then set\_state(WHERE,O)**

**if (exposed\_externally(INF)) then set\_state(WHERE,X)**

**}**

**state WHERE.X {}**

**state WHERE.None {}**

**state WHERE.H {}**

**state WHERE.N {}**

**state WHERE.S {}**

**state WHERE.C {}**

**state WHERE.W {}**

**state WHERE.O {}**

The main program is **main.fred:**

**##### Simulated Location**

**locations = Jefferson\_County\_PA**

**# locations = Allegheny\_County\_PA**

**##### Simulated Timeframe**

**start\_date = 2020-Jan-01**

**end\_date = 2020-Sep-01**

**use FRED::Influenza**

**include where.fred**

**enable\_health\_records = 1**

**INF.enable\_health\_records = 1**

**WHERE.enable\_health\_records = 1**

Let's run the program and plot the results:

**% fred\_job -k where -p main.fred**

**fred\_plot -k where -o where-count -v \ WHERE.totH,WHERE.totN,WHERE.totS,WHERE.totC,WHERE.totW,WHERE.totO,WHERE.totX -l "H,N,S,C,W,O,X" -t 'Place of Infection' --leg out -w**



The plot above shows the number of infections in each place. To show the percent of infections in each place, we can use the command:

**fred\_plot -k where -o where -v \ WHERE.totH/INF.totE,WHERE.totN/INF.totE,WHERE.totS/INF.totE,WHERE.totC/INF.totE,WHERE.totW/INF.totE,WHERE.totO/INF.totE,WHERE.totX/INF.totE -l "H,N,S,C,W,O,X" -t 'Place of Infection' --leg out -w**

The command above illustrates how to divide the time series of each WHERE state by the time series of INF.totE, thereby creating a fraction of transmissions in each place type. The results are shown here:



## Using Group-Related Variables

Suppose a model needs to define a new characteristic of some Group (that is, a Place or a Network). One approach is the define the characteristic as a variable and have the administrator for the group set the value for each specific group. Then members of that Group can access the variable using the **value()** function.

For example, suppose the model needs a specific vaccination rate for each School. The school-specific values could be set by each school's administrator:

**place School { has\_administrator = 1 }**

**my VaxRate**

**state COND.Start {**

**…**

**if (admin(School), School == 0) then set(VaxRate, 0.91)**

**if (admin(School), School == 1) then set(VaxRate, 0.83)**

...

}

Explanation: The predicate **admin(School)** is true if the agent executing the rule is an adminsitrator of a school. The predicate **School==0** is true of the agent's school is the first school in the school file.

The students in each school can be vaccinated according to the school's vaccination rate:

**state COND.ApplyRate {**

**wait(0)**

**next(Vaccinated) with prob(value(admin\_of\_School,VaxRate))**

**default(Unvaccinated)**

**}**

Explanation: We assume that each student enters the state **ApplyRate** due to other rules. The student then transitions to either the **Vaccinated** or the **Unvaccinated** states. The probability of becoming **Vaccinated** is obtained by the **value()** function, which in this case queries the administrator of the agent's School for the value of the variable **VaxRate**.

FRED provides a short cut: If the first argument of **value** is a Group name, then the administrator of the group that the agent executing the rule belongs to is queried for the value of the variable given as the second argument, so the rule could also be written as:

**state COND.ApplyRate {**

**wait(0)**

**next(Vaccinated) with prob(value(School,VacRate))**

**default(Unvaccinated)**

**}**

Note 1: if the agent in the first argument doesn't exist or the variable has not been set, then **value(agentId,Variable)** returns 0.0.

Note 2: The function **value(Expression1, Expression2)**  allows any mathematical expressions as either argument.

## Writing Loops

The recommended way to write a loop in FRED is shown in the following snippet:

**# Assume variable Count contains the desired number to repeat actions**

**global Count LoopCounter**

**Count = … # set the number of time to repeat the loop**

**# Initialize LoopCounter and enter the loop if Count > 0**

**state COND.State {**

**set(LoopCounter,0)**

**wait(0)**

**if (LoopCounter < Count) then next(ActionLoop)**

**default(Finished)**

**}**

**# Perform an action Count times**

**state COND.ActionLoop {**

***action(...)***

**set(LoopCounter, LoopCounter+1)**

**wait(0)**

**if (LoopCounter < Count) then next(ActionLoop)**

**default(Finished)**

**}**

Explanation: we assume that the goal is to repeat some action a number of times, and that number is stored in the variable Count. In state **COND.State**, we set a variable called LoopCounter to 0, and then enter the **ActionLoop** state if Count is greater than 0. Otherwise, we transition to a state called **Finished**. In the **ActionLoop** state, we have one or more action rules that perform the desired action once. The term ***action*** in the snippet would be replaced by the specific actions to be repeated, such as **set(x,y)**. Then we increase the loop counter variable by 1. If the loop counter is still less than the desired number, Count, we re-enter the **ActionLoop** state. Otherwise, we enter the **Finished** state.

Example: the following program computes the average age of the members of the agent's household.

**#### Simulated Location**

**locations = Jefferson\_County\_PA**

**##### Simulated Timeframe**

**start\_date = 2020-Jan-01**

**end\_date = 2020-Jan-02**

**##### Output Options**

**enable\_health\_records = 1**

**enable\_var\_records = 1**

**condition COND {**

**states = StartLoop ActionLoop Finished Report Done**

**enable\_health\_records = 1**

**}**

**my Ave**

**global\_list Members**

**global Count LoopCounter**

**state COND.StartLoop {**

**set\_list(Members, pool(Household))**

**set(Count, list\_size\_of\_Members)**

**set(Ave, 0)**

**set(LoopCounter,0)**

**wait(0)**

**if (LoopCounter < Count) then next(ActionLoop)**

**default(Finished)**

**}**

**state COND.ActionLoop {**

**set(Ave, Ave + value(select(Members, LoopCounter), age))**

**set(LoopCounter, LoopCounter+1)**

**wait(0)**

**if (LoopCounter < Count) then next(ActionLoop)**

**default(Finished)**

**}**

**state COND.Finished {**

**if (Count > 0) then set(Ave, Ave/Count)**

**wait(0)**

**if (Ave < 10) then next(REPORT)**

**default(Done)**

**}**

**state COND.Report { }**

**State(COND,Done) { }**

Explanation: The ActionLoop adds the age of each household member to the personal variable **Ave**, and then we divide by the number of household members in state **Finished**.The function:

**value(select(Members, LoopCounter), age)**

first selects the agent in the **Members** list at position indicated by **LoopCounter**, and gets that agent's age.

Note: In the above example, each agent was given its own **Ave** variable (assuming this info is needed elsewhere in the program), but the other variables were declared as global variables. Since **StartLoop** and **ActionLoop** are both transient states (i.e. with zero wait time), the entire loop is executed for each agent before moving on to the next agent. So re-using the temporary variables **Member, Count,** and **LoopCounter** for all agents saves a significant amount of memory.

The example program above includes some statements to verify the computation. Agents whose average household age is less than 10 end up in the **Report** state. By examining the health\_records file, we can check that correctness of the program:

**% run\_fred -p age.fred**

**fred\_compile age.fred ...**

**No errors found.**

**No warnings.**

**(export OMP\_NUM\_THREADS=1 ; FRED -p age.fred -r 1 -d OUT 2>&1 > OUT/RUN1/LOG)**

**% fred\_get\_records -s REPORT | tail -1**

**HEALTH RECORD: 2020-01-01 12am day 0 person 40543 age 18 sex F race 1 household H-11052086 school NONE income 1000 CONDITION COND CHANGES from Finished to Report**

**% fred\_get\_records -g H-11052086 | grep Rep**

**HEALTH RECORD: 2020-01-01 12am day 0 person 18146 age 1 sex F race 1 household H-11052086 school NONE income 1000 CONDITION COND CHANGES from Finished to Report**

**HEALTH RECORD: 2020-01-01 12am day 0 person 40543 age 18 sex F race 1 household H-11052086 school NONE income 1000 CONDITION COND CHANGES from Finished to Report**

## School Closure by Admin Agents

The following example program shows how to close schools according a fixed schedule of holiday breaks. It illustrates several aspects of the FRED language:

* Use of administrative agents
* Closing a place by an administrator
* The use of the date and data\_range predicates
* Several examples of looping states
* The use of wait\_until rules (in the second example)

**##################################################**

**##### Simulated Location**

**locations = Jefferson\_County\_PA**

**##### Simulated Timeframe**

**start\_date = 2020-08-01**

**end\_date = 2022-07-31**

**##### OUTPUT OPTIONS**

**enable\_health\_records = 1**

**##### School Administrators decide when to close schools**

**place School { has\_administrator = 1 }**

**condition ADMIN {**

**states = Check Open WinterBreak SpringBreak SummerBreak**

**admin\_start\_state = Start**

**enable\_health\_records = 1**

**}**

**# school administrators go to Check; all other agents (e.g. individuals) are Excluded**

**state ADMIN.Start {**

**if (admin(School)) then next(Check)**

**default(Excluded)**

**}**

**# If we start in the middle of a break, go to the correct break state**

**state ADMIN.Check {**

**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)**

**}**

**# Rules for starting a school break (check each day)**

**state ADMIN.Open {**

**wait(24)**

**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)**

**}**

**# Rules for WinterBreak**

**state ADMIN.WinterBreak {**

**close(School)**

**wait(24)**

**if (date(Jan-03)) then next(Open)**

**default(WinterBreak)**

**}**

**# Rules for SpringBreak**

**state ADMIN.SpringBreak {**

**close(School)**

**wait(24)**

**if (date(Mar-16)) then next(Open)**

**default(SpringBreak)**

**}**

**# Rules for SummerBreak**

**state ADMIN.SummerBreak {**

**close(School)**

**wait(24)**

**if (date(Aug-26)) then next(Open)**

**if default(SummerBreak)**

**}**

Comments: There is one admin agent per school, but since they all have the same transition rules in this example, all schools will open or close simultaneously. The **Open** state checks the date each day, and if the current date is within the range of one of the school break periods, the admin agent proceeds to the appropriate break period state. When the admin agent enters a break state, it closes the school that it administers. Each of the states **WinterBreak**, **SpringBreak**, and **SummerBreak** cause the agent to check the date each day, and either loop back to the same state, or move to the **Open** state. The **Open** state does not include the **close(School)** action, so the school operates on its normal schedule while the agent is in that state.

By plotting the counts for the states, **WinterBreak**, **SpringBreak**, and **SummerBreak**, we can verify that all 26 schools in the example location open and close according to the desired schedule:



We can improve the efficiency of the program by using wait\_until rules for the break states. That is, instead of testing the date every day, we can just wait until the end of the school break period. Here are the revised transition rules:

**# Rules for WinterBreak**

**state ADMIN.WinterBreak {**

**close(School)**

**wait(until\_Jan-03)**

**next(Open)**

**}**

**# Rules for SpringBreak**

**state ADMIN.SpringBreak {**

**close(School)**

**wait(until\_Mar-16)**

**next(Open)**

**}**

**# Rules for SummerBreak**

**state ADMIN.SummerBreak {**

**close(School)**

**wait(until\_Aug-26)**

**next(Open)**

**}**

The program using these rules produces exactly the same results as the original program.

## School Closure of Individual Schools Trggered by an Epidemic

In this snippet, we illustrate the following school closure policy:

* Each school open or closes independently
* A school closure if more than 5 students in the school are staying home due to illness.
* Once closed, the school remains closed for 14 days.

This is just an example and is not intended as a recommended school closure policy.

Building on the last example, we first set up School administrators. Then add a state called **Close** that indicates that their school should be closed. Add Rules that close the school according to their policy, and open it after 14 days.

The school administrator using the information about how many students in the school are staying home due to illness (see [Changes in Personal Behavior Based on Symptoms](#_Changes_in_Personal)).

**place School { has\_administrator = 1 }**

**condition ADMIN {**

**states = Check Open WinterBreak SpringBreak SummerBreak Close**

**}**

**# Rules for an epidemic-related school closure:**

**state ADMIN.Open {**

**if (current\_count\_of\_StayHome.Yes\_in\_School > 5) then next(Close)**

**}**

**state ADMIN.Close {**

**close(School)**

**wait(14\*24)**

**next(Check)**

**}**

## Seasonality by Month

Many Conditions are seasonal, meaning that are more likely to occur at certain times of the year. This snippet illustrates how to change the transmissibility of a Condition based on the time of year.

The approach is to define a start state for the Import Agent (called Seasonality in this example) in which the Import Agents changes the transmissibility according to the time of year. In this example, we set the transmissibility to 0.2 in April, 1.0 in May and 0.5 in June. The Import Agent first sets the transmissibility to 0 each day and then increases it during one of the three months of April. May, or June. Therefore, the transmissibility will be 0 during the other 9 months of the year.

This method could be extended to set the transmissibility to different values for each month, for individual days or even according to time of day.

**condition COND {**

**states = Start ... Seasonality**

**import\_start\_state = Seasonality**

**}**

**state COND.Seasonality {**

**set\_trans(COND, 0.0)**

**if (month == 4) then set\_trans(COND, 0.2)**

**if (month == 5) then set\_trans(COND, 1.0)**

**if (month == 6) then set\_trans(COND, 0.5)**

**wait(24)**

**next(Seasonality)**

**}**

## Seasonality by Day of Year

Many Conditions are seasonal, meaning that are more likely to occur at certain times of the year. This snippet illustrates how to change the transmissibility of a Condition based on the day of the year.

The approach shown in this snippet is to define a start state for the Import Agent (called Seasonality in this example) in which the Import Agents changes the transmissibility according to a formula that varies over the year. In this example, we model seasonality as a cosine wave that varies from 1 (at the peak day of transmissibility) to 0 (at a half year later).

We use four global variables to set:

* the maximum transmissibility of the Condition (set to 1.5 in this example)
* the peak day of the year (between 1 and 365, with 1 representing Jan-01)
* the fraction of seasonal reduction (1.0 means no transmission one half-year off-peak)
* the current number of days off-peak

**condition COND {**

**states = Start ... Seasonality**

**import\_start\_state = Seasonality**

**}**

**global max\_transmissibility**

**global peak\_day\_of\_year**

**global seasonal\_reduction**

**global days\_from\_peak**

**max\_transmissibility = 1.5**

**peak\_day\_of\_year = 1**

**seasonal\_reduction = 1.0**

**days\_from\_peak = -1**

**# Determine how many days away from the peak day:**

**state COND.Seasonality {**

**set(days\_from\_peak, abs(day\_of\_year - peak\_day\_of\_year))**

**# Set the transmissibility according to a cosine wave with value of max\_transmissibility on**

**# the peak and a period of 365 days:**

**set\_trans(COND, max\_transmissibility \* \**

**(1.0 - seasonal\_reduction\*(1 - 0.5\*(1+cos(2\*3.14159\*days\_from\_peak/365)))))**

**# wait one day:**

**wait(24)**

**# and repeat:**

**next(Seasonality)**

**}**

The snippet above produces the following results:



As another example, suppose we can to have peak transmissibility in both the Spring and the Fall, and have the off-peak transmissibility fall to half the peak transmissibility. We could make the following changes:

**peak\_day\_of\_year = 91**

**seasonal\_reduction = 0.5**

and change the period of the curve to be 182.5 days:

**set\_trans(COND, max\_transmissibility \* \**

1. **- seasonal\_reduction\*(1 - 0.5\*(1+cos(2\*3.14159\*days\_from\_peak/182.5)))))**

The resulting transmission curve follows:



## Using Elevation Data

The built-in synthetic population files include the elevation (measured in meters above sea-level) for every built-in place type (Households, School, Workplaces and Hospitals).

This example shows how to use that data to assign agents to different states. This might be useful for models in which a risk is associated with a place's elevation, such as the risk of flooding.

Consider the following FRED program:

**##### Simulated Location**

**locations = Allegheny\_County\_PA**

**##### Simulated Timeframe**

**start\_date = 2020-Jan-01**

**end\_date = 2020-Jan-02**

**##### CONDITIONS**

**condition ELEV {**

**states = Min Low Med High Max**

**}**

**state ELEV.Start {**

**if (elevation\_quintile\_of\_Household == 1) then next(Min)**

**if (elevation\_quintile\_of\_Household == 2) then next(Low)**

**if (elevation\_quintile\_of\_Household == 3) then next(Med)**

**if (elevation\_quintile\_of\_Household == 4) then next(High)**

**if (elevation\_quintile\_of\_Household == 5) then next(Max)**

**}**

**state ELEV.Excluded {}**

**state ELEV.Min {}**

**state ELEV.Low {}**

**state ELEV.Med {}**

**state ELEV.High {}**

**state ELEV.Max {}**

Remarks: the program assigns a state in the ELEV Condition to each agent. The transition rules in the Start state move each agent to one of the states: Min, Low, Med, High or Max, depending on the elevation quintile of the agent's household. Each quintile contains 20% of the households in the population, sorted by elevation in this case. These quintiles are computed automatically when the place files are read in. Quartiles are also computed. Quartiled and quintiles are also computed for **size** and **income.**

Once an agent is assigned to a state, it remains there indefinitely. If we want to visualize the results, we could add the following statements:

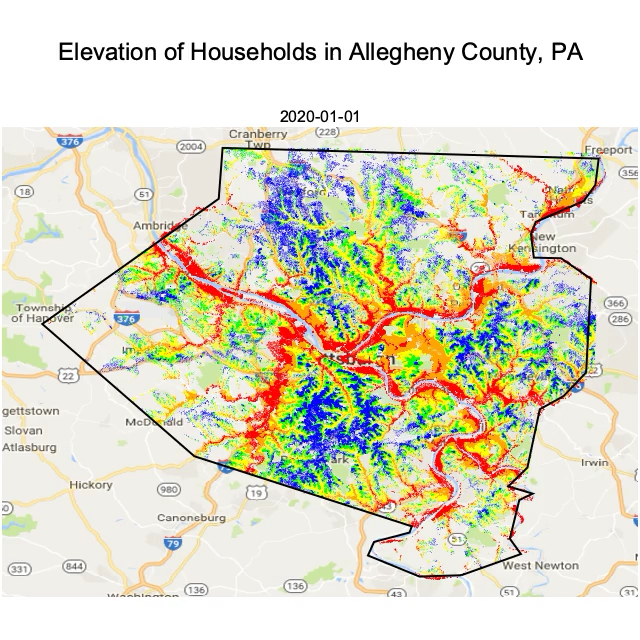
**enable\_visualization\_layer = 1**

**condition ELEV {**

**Low.visualize = 1**

**Min.visualize = 1**

**Med.visualize = 1**

** High.visualize = 1**

**Max.visualize = 1**

**Low.is\_dormant = 1**

**Min.is\_dormant = 1**

**Med.is\_dormant = 1**

**High.is\_dormant = 1**

**}**

The latter statements tell FRED that the visualization data only needs to be recorded once (to save space). The movie below shows the elevation of household using the colors red, orange, yellow, green and blue for the states Min, Low, Med, High and Max, respectively.

In addition to the factors using quartiles or quintiles, you can directly test the elevation of an agent's place, for example:

**if (elevation\_of\_School < 250) then …**

## Surveys: Taking a Sample of the Population

FRED can simulate a population survey by taking a sample of the population. To do so, we can create a Condition called, say, SURVEY and assign some set of individual agents to a selected state.

Suppose we want to survey a specific number of individuals. We could use the following Condition:

**condition SURVEY {**

**states = Ready Participating Import**

**import\_start\_state = Import**

**exposed\_state = Participating**

**}**

**state SURVEY.Ready {**

**set\_sus(SURVEY,1)**

**wait()**

**next()**

**}**

**state SURVEY.Participating {}**

**state SURVEY.Import {**

**import\_count(1000)**

**wait()**

**next()**

**}**

The READY state makes all agents susceptible to the survey at time 0. The Import Agent then selects 1000 of the susceptible agents at random and exposes them to SURVEY, which causes the exposed agents to enter the **Participating** state. All other agents remain in the **Start** state forever.

The survey group can be used by any other Condition COND by putting them in a state called, say, **Ask** with a rule such as:

**if (current\_state\_in\_SURVEY == Participating) then next(Ask)**

As long as the SURVEY Condition is defined before the COND Condition in the FRED program, this rule will select the participating individuals, even if it is applied at time 0.

If it is desired to limit the survey participants to a specific subset of the population, say, females between 18 and 35 years old, just add that specification to the START pseudostate:

**state SURVEY.Start {**

**if (sex == female, 18 <= age, age <= 35) then next(READY)**

**default(Excluded)**

**}**

To select a fraction of the population instead of a fixed number, change the rule for the **Import** state to something like:

**state SURVEY.Import {**

**import\_per\_capita(0.001)**

**wait()**

**next()**

**}**

Now the Import Agent will select about one out of 1000 of the susceptible individuals for the survey.

## Taking a Temperature Survey of the Population

This snippet illustrates the use of personal variables. In the model, each individual is population is assigned a personal variable called **temperature.** This variable is tracked in a 1% sample of the population. For each person in the sample, theirtemperature is set to 98.6 at the start of the simulation. Each day, a random value between -1 and 1 is added to the agent's temperature. If the new temperature is above a threshold, the agent moves to the **High** state. If the temperature is below another threshold, the agent goes to the **Low** state. Otherwise the agent goes to the **Normal** state.

**##### Simulated Location**

**locations = Jefferson\_County\_PA**

**##### Simulated Timeframe**

**start\_date = 2020-Jan-01**

**end\_date = 2020-Jul-01**

**# enable\_health\_records = 1**

**## each person has an individual temperature**

**my temp**

**condition TEMP {**

**states = Check Normal Low High**

**enable\_health\_records = 1**

**}**

**# select a 1% sample of the popualtion:**

**state TEMP.Start {**

**next(Check) with prob(0.01)**

**default(Excluded)**

**}**

**# check temperature and go to a temperature category:**

**state TEMP.Check {**

**if (sim\_day==0) then set(temp, 98.6) # initialize to 98.6**

**set(temp, temp + uniform(-1,1)) # add a random value between -1 and 1**

**wait(0)**

**if (temp < 96) then next(Low)**

**if (temp > 101) then next(High)**

**default(Normal)**

**}**

**# check temperature every day:**

**state TEMP.Normal { wait(24); next(Check) }**

**state TEMP.Low { wait(24); next(Check) }**

**state TEMP.High { wait(24); next(Check) }**

The model is in a file called **temperature.fred.** Run the model with the command:

**% fred\_job -k temperature -p temperature.fred**

Next we plot the number of agents in each state:

**% fred\_plot -k temperature -o temperature -v TEMP.Normal,TEMP.High,TEMP.Low -n \ -s 1 -l "Normal,High,Low" --legend out -t "Temperature Groups"**

****

Notice that all the agents start in the **Normal** state, but over time their temperatures drift so that the number of agents in the **High** and **Low** states increases. In this artificial model, termperatures follow a random walk and there is no pressure to return to normal, so the temperature variation simply increase over time.

As part of a larger model, agents might transition to other states based on their TEMP state; for example, agents in the High state may seek medical treatment.

As an exercise for the reader, try to change the model so that the temperature is regulated within a viable range.

## Selecting the Closest Place

This snippet shows how to have agents select the closest place of a given type, for example, selecting the closest pharmacy to an agent's home. This snippet also illustrates:

* How to define new places within a FRED program
* How to define a network
* How to use the edge weights of a network

Let's define the location of 5 pharmacies and then assign all the agents in the population to their closest pharmacy.

We can define 5 new phamacies in the FRED program using the following statements:

**Place Pharmacy {**

**has\_administrator = 1**

**add = 94200301 40.436276 -79.985191 0**

**add = 94200302 40.366906 -80.086822 0**

**add = 94200303 40.351389 -79.960502 0**

**add = 94200304 40.346519 -79.855764 0**

**add = 94200305 40.469444 -79.95553 0**

**}**

Because the Pharmacy place type has administrative agents, FRED will automatically create a global list variable called **PharmacyList** that contains a list of all ther admin agents for pharmacies.

The **add** properties above create 5 new pharmacy locations, giving the place id, the latitude, longitude and elevation of each place. Elevation is set to 0 because it is not used in this example. Place id's for user-defined places begin with the digit 9 to avoid conflicts with any places defined in the synthetic population. Here we are using the next 5 digits to code the county FIPS code for Allegheny County, PA (42003), followed by 2 digits to index the pharmacies.

The following Condition called **CUSTOMER** implements the following steps:

1. Make a network to connect each agent at least age 65 to all the pharmacies.
2. Give each edge in the network a weight inversely proportional to distance between the customer's household and the pharmacy.
3. Have each customer select the closest pharmacy.
4. Customer joins the selected pharmacy.

**network PharmacyNet { }**

**my MyPharmacy**

**condition CUSTOMER {**

**states = Select**

**enable\_health\_records = 1**

**}**

**state CUSTOMER.Start {**

**if (age > 64) then next(Select)**

**default(Excluded)**

**}**

**state CUSTOMER.Select {**

**# Step 1. Make a network to connect each participant to all the pharmacies.**

**add\_edge\_to(PharmacyNet, PharmacyList)**

**# Step 2: Give each edge in the network a weight inversely proportional to**

**# distance between the customer's household and the pharmacy.**

**set\_weight(PharmacyNet, PharmacyList, 1.0/dist(Household,other:Pharmacy))**

**# Step 3. Have each customer select the closest pharmacy.**

**# Note: Use value function to get the pharmacy place id.**

**set(MyPharmacy, value(id\_of\_max\_weight\_outward\_edge\_in\_PharmacyNet, Pharmacy))**

**# Step 4: The customer joins the closest pharmacy**

**join(Pharmacy,MyPharmacy)**

**wait()**

**next()**

**}**

The **add\_edge\_to** action connects the agent in the PharmacyNet to all the pharmacy administrative agents in the PharmacyList. Remember that networks are connections between agents, in this case, connections between individual agents and administrative agents for pharmacies.

In Step 2, the agent sets the weights to all the admin agents with the value of 1 divided by the distance between the agent's household and the admin agent's pharmacy location.

In Step 3, the agent sets it MyPharmacy variable to the place id of the pharmacy associated with the agent id attached to the edge with the maximum weight in the network, that is, the admin agent whose pharmacy is closest.

In Step 4, the agent finally joins the selected pharmacy, meaning that from this point on, that agent is associated with the given pharmacy location.

The next snippet shows the results of this process on a Report.

## Mapping a Catchment Area

A *catchment area* is the geographical area served by an institution. This snippet shows how to display the catchment area for the pharmacies defined in the last two snippets.

We use a Condition called MAP. The purpose of this condition is to put individuals into specific states depending on the specific pharmacy they selected as the closest pharmacy to their home.

**Condition MAP {**

**states = Individual Agent Admin \**

**Pharmacy1 Pharmacy2 Pharmacy3 Pharmacy4 Pharmacy5**

**admin\_start\_state = Start**

}

The rules in the Start state separate the administrative agents (that have id < 0) from the individual agents, so that we can display the pharmacy locations associated with the administrative agents:

**state MAP.Start {**

**if (id < 0) then next(Admin)**

**default(Individual)**

**}**

The rules for the Individual state separate the individual agents into states associated with each pharmacy in the model. Individuals with no pharmacy are excluded:

**state MAP.Individual {**

**wait(0)**

**if (MyPharmacy == 94200301) then next(Pharmacy1)**

**if (MyPharmacy == 94200302) then next(Pharmacy2)**

**if (MyPharmacy == 94200303) then next(Pharmacy3)**

**if (MyPharmacy == 94200304) then next(Pharmacy4)**

**if (MyPharmacy == 94200305) then next(Pharmacy5)**

**default(Excluded)**

**}**

The remaining states are all absorbing state, that is, the agent remain in those states forever:

**State(MAP,Admin) {}**

**State(MAP,Pharmacy1) {}**

**State(MAP,Pharmacy2) {}**

**State(MAP,Pharmacy3) {}**

**State(MAP,Pharmacy4) {}**

**State(MAP,Pharmacy5) {}**

We want to display the household locations of the individuals of these states on a map:

**condition MAP {**

**Pharmacy1.visualize = 1**

**Pharmacy2.visualize = 1**

**Pharmacy3.visualize = 1**

**Pharmacy4.visualize = 1**

**Pharmacy5.visualize = 1**

**}**

We also want to map the locations of the phamacies themselves. To do this, we can visualize the pharmacy associated with each adminstrative agent:

**condition MAP {**

**Admin.visualize = 1**

**Admin.place\_type\_to\_visualize = Pharmacy**

**}**

For convenience, we include the entire program here.

**################################################################**

**#**

**# Mapping a Catchment Area**

**#**

**# File: catchment.fred**

**# Author: John Grefenstette**

**# Date: 5 Dec 2019**

**#**

**##### SIMULATED LOCATION**

**locations = Allegheny\_County\_PA**

**##### SIMULATED TIMEFRAME**

**start\_date = 2020-Jan-01**

**end\_date = 2020-Jan-01**

**##### OUTPUT OPTIONS**

**enable\_visualization\_layer = 1**

**##### Places to be generated:**

**place Pharmacy {**

**has\_administrator = 1**

**add = 94200301 40.436276 -79.985191 0**

**add = 94200302 40.366906 -80.086822 0**

**add = 94200303 40.351389 -79.960502 0**

**add = 94200304 40.346519 -79.855764 0**

**add = 94200305 40.469444 -79.95553 0**

**}**

**################################################################**

**#**

**# SURVEY CONDITION**

**#**

**# Select a random sample of the population for a survey**

**#**

**condition SURVEY {**

**states = Ready Participating Import**

**import\_start\_state = Import**

**exposed\_state = Participating**

**}**

**state SURVEY.Ready {**

**set\_sus(SURVEY,1)**

**wait()**

**next()**

**}**

**state SURVEY.Import {**

**import\_count(1000)**

**wait()**

**next()**

**}**

**State(SURVEY,Participating) {}**

**#**

**# END OF SURVEY CONDITION**

**#**

**################################################################**

**################################################################**

**#**

**# CUSTOMER CONDITION**

**#**

**# For each participant in the SURVEY, select the closest pharmacy.**

**network PharmacyNet {}**

**my MyPharmacy**

**condition CUSTOMER {**

**states = Select**

**enable\_health\_records = 1**

**}**

**state CUSTOMER.Start {**

**wait(0)**

**if (current\_state\_in\_SURVEY == Participating) then next(Select)**

**default(Excluded)**

**}**

**state CUSTOMER.Select {**

**# Step 1. Make a network to connect each participant to all the pharmacies.**

**add\_edge\_to(PharmacyNet, PharmacyList)**

**# Step 2: Give each edge in the network a weight inversely proportional to**

**# distance between the customer's household and the pharmacy.**

**set\_weight(PharmacyNet, PharmacyList, 1.0/dist(Household,other:Pharmacy))**

**# Step 3. Have each customer select the closest pharmacy.**

**# Note: Use value function to get the pharmacy place id.**

**set(MyPharmacy, value(id\_of\_max\_weight\_outward\_edge\_in\_PharmacyNet, Pharmacy))**

**# Step 4: The customer joins the closest pharmacy**

**join(Pharmacy,MyPharmacy)**

**wait()**

**next()**

**}**

**#**

**# END of CONDITION CUSTOMER**

**#**

**################################################################**

**################################################################**

**#**

**# MAP CONDITION**

**#**

**Condition MAP {**

**states = Admin Individual \**

**Pharmacy1 Pharmacy2 Pharmacy3 Pharmacy4 Pharmacy5 Excluded**

**Pharmacy1.visualize = 1**

**Pharmacy2.visualize = 1**

**Pharmacy3.visualize = 1**

**Pharmacy4.visualize = 1**

**Pharmacy5.visualize = 1**

**Admin.visualize = 1**

**Admin.place\_type\_to\_visualize = Pharmacy**

**}**

**state MAP.Start {**

**if (id < 0) then next(Admin)**

**default(Individual)**

**}**

**state MAP.Individual {**

**wait(0)**

**if (MyPharmacy == 94200301) then next(Pharmacy1)**

**if (MyPharmacy == 94200302) then next(Pharmacy2)**

**if (MyPharmacy == 94200303) then next(Pharmacy3)**

**if (MyPharmacy == 94200304) then next(Pharmacy4)**

**if (MyPharmacy == 94200305) then next(Pharmacy5)**

**default(Excluded)**

**}**

**state MAP.Admin {}**

**state MAP.Pharmacy1 {}**

**state MAP.Pharmacy2 {}**

**state MAP.Pharmacy3 {}**

**state MAP.Pharmacy4 {}**

**state MAP.Pharmacy5 {}**

**#**

**# END of MAP CONDITION**

**#**

**################################################################**

Assuming the job called catchment is run, the following command produces the map below:

**% fred\_make\_movie -k catchment --play 1 -v**

**MAP.Pharmacy1,MAP.Pharmacy2,MAP.Pharmacy3,MAP.Pharmacy4,MAP.Pharmacy5,MAP.Admin**

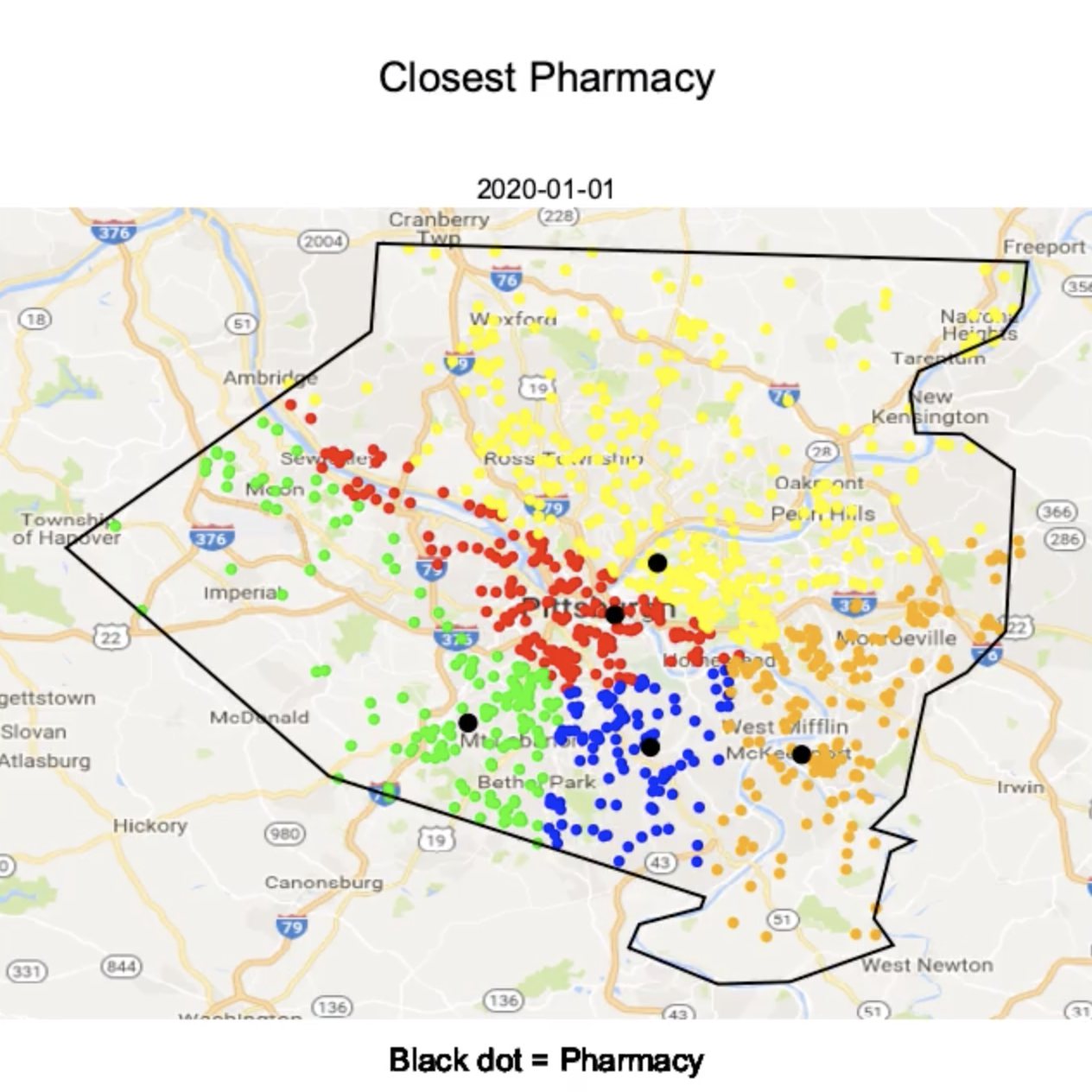
**--ps 0.005,0.005,0.005,0.005,0.005,0.008**

**--colors red,green,blue,orange,yellow,black**

**--title "Closest Pharmacy"**

**--center\_caption "Black dot = Pharmacy"**

**--caption\_colors black,black,black**



This map can also be created using the Make Movie page on FRED Web.

## Changing the Contact Rates of Places

We can use administrative agents to change the rate transmissibility contacts in particular places. This is done through the administrative action **set\_contacts(Expression).** The original contact rate for the place administered is multiplied by the value of the Expression.

Consider the following model called **full-contacts.fred**:

**##### Simulated Location**

**locations = Allegheny\_County\_PA**

**##### Simulated Timeframe**

**start\_date = 2020-Jan-01**

**end\_date = 2020-Jul-01**

**##### simple influenza model:**

**include inf.fred**

**##### to report where infections occur:**

**include where.fred**

This model uses the WHERE condition discussed previouslyto keep track of how many agents are infected in each type of place.

For comparison, we define a model call **reduced-contacts.fred** in which the contacts in each neighborhood are reduced at simulation day 10 (perhaps due to increased social distancing). This is accomplished by defining an administrative agent for each neighborhood. The CONTROL condition is designed for these administrative agents (all others are Excluded). After 10 days, the contacts are deuced by 50% in the SetContacts state:

**##### Simulated Location**

**locations = Allegheny\_County\_PA**

**##### Simulated Timeframe**

**start\_date = 2020-Jan-01**

**end\_date = 2020-Jul-01**

**##### simple influenza model:**

**include inf.fred**

**##### to report where infections occur:**

**include where.fred**

**# include an admin agent for each neighborhood**

**place Neighborhood {**

**has\_administrator = 1**

**}**

**# sample admin agent condition**

**condition CONTROL {**

**states = Wait SetContacts**

**admin\_start\_state = Start**

**}**

**state CONTROL.Start {**

**if (admin(Neighborhood)) then next(Wait)**

**default(Excluded)**

**}**

**state CONTROL.Wait {**

**wait(24\*10)**

**next(SetContacts)**

**}**

**state CONTROL.SetContacts {**

**set\_contacts(0.5)**

**wait()**

**next()**

**}**

The following plot compares the weekly incidence for the full-contact and reduced-contact models:



We can drill down to see the difference in incidence within the neighborhood:



Perhaps a more realistic version of the previous rule would be one that take the density of the neighborhood into account directly. We created a third model called **proportional-contacts.fred** that is identical to **reduced-contacts.fred** except for the SetContact state:

**state CONTROL.SetContacts {**

**if (size\_of\_Neighborhood < 1000) then \**

**set\_contacts(0.5 + size\_of\_Neighborhood\*0.0005)**

**if (1000 <= size\_of\_Neighborhood) then \**

**set\_contacts(size\_of\_Neighborhood\*0.0001)**

**wait()**

**next()**

**}**

In this case, the contact factor is set to 0.5 plus 0.0005 for each individual for neighborhoods with up to 999 individuals. For neighborhoods with 1000 or more individuals, the contact factor is set to 0.0001 time the number of individuals. The effect is that contact rates grow slowly for the first 1000 individuals, then faster for larger neighborhoods. This is not meant to represent an actual contact pattern, but is meant to illustrate that the model can include any form of contact pattern than the modeler needs to specify.

The following plot compares the weekly incidence for the full-contact and proportional-contact models:



Please see FRED/models/set\_contacts for working copies of the models described aboved as well as the METHODS script for the commands used to generate the plots shown.

## Density-Based Transmission in Urban vs Rural Locations

This model illustrates the use of the *transmission by probability* mode, or *density-based transmission*. This mode is useful for models in which a condition might be easily transmitted to everyone an individual comes into slight contact with (for example, a highly contagious respiratory disease like measles). With density-based transmission, an infectious individual may transmit the condition to a large number of people in a crowded place.

The following model is called **urban-density.fred.** It includes the INF and WHERE conditions discussed previously. It adds the properties to the **Neighborhood** place type to use density-based transmission for the INF condition, with a probability = 0.0003 of contact between any two individuals in a neighborhood in any given hour. The location is Allegheny County, which surrounds the city of Pittsburgh, PA.

**##### Simulated Location**

**locations = Allegheny\_County\_PA**

**##### Simulated Timeframe**

**start\_date = 2020-Jan-01**

**end\_date = 2020-Jul-01**

**include inf.fred**

**include where.fred**

**place Neighborhood {**

**density\_transmission\_for\_INF = 1**

**density\_contact\_prob\_for\_INF = 0.0003**

**}**

We compare this with another model called **rural-density.fred** in which the only change is the location, a rural county in Pennsylvania:

**##### Simulated Location**

**locations = Jefferson\_County\_PA**

Note that both models use identical parameters for the transmissibility of influenza and the same contact rates for households, schools and workplaces. The probability of contact in the neighborhood is also the same, but we expect different outcomes because the density of people is substantially less in the rural county than in the urban county.

After running both models, we can compare the two resulting influenza epidemics to see the different effects of density-based transmission in urban and rural locations. First, we see that the attack rate, or percentage of the population that gets infected, varies from about 36% in the urban county to about 25% in the rural county.



The next plot shows difference in weekly incidence. We can see that the epidemic in the urban setting is larger and happens more quickly than does the epidemic in the rural location:



By plotting the states in the WHERE condition, we can see differences in where individual get infected. In the urban model, about 28% of infections occur in the densely population neighborhoods, nearly the same as the percentage in households:

In the rural location, the percent of infection in the neighborhood drops to about 8%, reflecting the lower density in this county. The percentage of infection occurring in the other place types (Household, Classroom, School, Office and Workplace) are in the same relative order as in the urban setting, but they vary in absolute percentages because of the relatively small number of infections in the neighborhood.





As this example shows, the use of density-based transmission can lead to significant differences int he spread of conditions in urban and rural locations.

Please see FRED/models/density/METHODS for the commands used to generate the plots shown above.

## Small Ad Hoc Gatherings

This section illustrates the process of defining small ad hoc gathering such as house parties. We use two advanced features of FRED:

* dynamically generated places
* transmission of places (i.e., invitations)

Narrative of the model: A subset of the households is selected to host house parties. House parties occur each Saturday evening at 6pm, last for 4 hours, and are limited to 25 guests. Guests are invited based on the social contacts of the head of the household. Each week, the host invites up to 25 people, randomly selected from the pool of neighbors and co-workers. The invited guests and the host attend the party each week, during which time they may transmit other communicable conditions.

To define a dynamically generated Place Type, we use the declaration:

**place Party {**

**base\_type = Household**

**starts\_at\_hour\_18\_on\_Sat = 4**

**max\_size = 45**

**}**

The base\_type property defines which of the place types of the host is used as the location for an instance of the type. In this case, we are using the host's household location. The next property specifies the days and times that the place is open for meetings. The final property gives the maximum size for the meeting.

We also define a condition called PARTY that contains states for both hosts and guests with the declaration:

**condition PARTY {**

**states = Start Host Invite StopInviting \**

**Guest Waiting Invited Accepted \**

**Attend Over Excluded**

**transmission\_mode = proximity**

**transmissibility = 1**

**place\_type\_to\_transmit = Party**

**exposed\_state = Invited**

**Host.start\_hosting = 1**

**}**

The condition is transmissible with proximity transmission mode and transmits a place type. This means that transmissible agents (e.g. party hosts) can transmit their specific party to others that the host comes into contact with. The person receiving the invitation will join the host's place (unless it is already at maximum capacity). The person receiving the invitation proceeds immediately to the **Invited** state.

The states associated with hosts are:

* Host - the agent creates an instance of a party and waits before inviting people to the next party.
* Invite - the host of the party invites others to join the party.
* StopInviting - the host cuts off the invitation process
* Over - the party is over, return to Host state.

The states associated with guests are:

* Guest - the set of potential guests.
* Waiting - the agents waits to be invited to a party.
* Invited - the agent received an invitation to a party.
* Accepted - the agent accepts an invitation to a party.
* Attend - the agent attends a party and interacts with other attendees.
* Over - the party is over. Return to the Waiting state.

The **Start** pseudostate decides which agents enter the Host Pool, the Guest Pool, or are Excluded.

A global variable called **HostingRate** is defined and gives the fraction of households that can host a party in this model.

The model also defines how the PARTY condition spreads. Since PARTY has proximity transmission, the host can transmit invitations to anyone who shares activity locations with host without any further definitions in the model. However, we can override the usual transmission rates if desired. In this example, we define the transmission mode to be density transmission in neighborhoods with a contact probability of 0.1 (a high rate of contact).

**place Neighborhood {**

**density\_transmission\_for\_PARTY = 1**

**density\_contact\_prob\_for\_PARTY = 0.1**

**}**

Note that the host is transmissible only during the Invite state, which is set to last 4 days in this model.

Whenever agents enter the Waiting state, they quit any PArty that they may be a member of. This allows guests to change the party they attend, depending on which host (if any) invites them this week.

Once an agent enters the Invited state, due to have been transmitted a party invitation, the agent accepts the party if they actually obtain membership. They will not be a member if the party already has the maximum number of members.

The full model is shown below:

**#####################################################**

**#**

**# A model of ad hoc house parties**

**#**

**place Party {**

**starts\_at\_hour\_18\_on\_Sat = 4**

**max\_size = 25**

**density\_transmission\_for\_INF = 1**

**density\_contact\_prob\_for\_INF = 0.1**

**}**

**global HostingRate**

**HostingRate = 0.01**

**condition PARTY {**

**states = Start Host Invite StopInviting \**

**Guest Waiting Invited Accepted \**

**Attend Over Excluded**

**transmission\_mode = proximity**

**transmissibility = 1**

**place\_type\_to\_transmit = Party**

**exposed\_state = Invited**

**Host.start\_hosting = 1**

**}**

**place Neighborhood {**

**density\_transmission\_for\_PARTY = 1**

**density\_contact\_prob\_for\_PARTY = 0.1**

**}**

**state PARTY.Start {**

**wait(0)**

**if (household\_relationship==householder) then next(Host) with prob(HostingRate)**

**default(Guest)**

**}**

**state PARTY.Host {**

**wait(24)**

**next(Invite)**

**}**

**state PARTY.Invite {**

**set\_trans(PARTY, 1.0)**

**wait(24\*4)**

**next(StopInviting)**

**}**

**state PARTY.StopInviting {**

**set\_trans(PARTY, 0.0)**

**wait(24)**

**next(Over)**

**}**

**state PARTY.Guest {**

**wait(1)**

**next(Waiting)**

**}**

**state PARTY.Waiting {**

**quit(Party)**

**set\_sus(PARTY, 1.0)**

**wait()**

**next()**

**}**

**state PARTY.Invited {**

**wait(0)**

**if (not(member(Party))) then next(Waiting)**

**default(Accepted)**

**}**

**state PARTY.Accepted {**

**wait(1)**

**default(Attend)**

**}**

**state PARTY.Attend {**

**wait(until\_Sat\_at\_11pm)**

**next(Over)**

**}**

**state PARTY.Over {**

**wait(until\_Sun)**

**if (host(Party)) then next(Host)**

**default(Waiting)**

**}**

After running the model, we can plot the number of agents in states **Host, Invite** and **Attend** to see the dynamics of the party activities:



Finally, the demonstrate that the agents at parties do actually interact, we included the INF and WHERE conditions described previously in the model's main program:

**##### Simulated Location**

**locations = Jefferson\_County\_PA**

**# locations = Allegheny\_County\_PA**

**##### Simulated Timeframe**

**start\_date = 2020-May-01**

**end\_date = 2020-Aug-15**

**include party.fred**

**include inf.fred**

**include where.fred**

After running this model, we can plot the incidence of influenza that occurs at parties:



Please see FRED/models/party for the files for this example. The METHODS file contains the commands used to generate the plots shown above.

## Mass Gatherings

Mass gathering are of particular interest due to the possibility of *super-spreader* events in which a susceptible person infects a large number of people at once in a densely crowded space. In this example, we show how to define a new place type called a **Concert** that attracts a large crowd.

The model begins by defining the **Concert** place type. Each place of this type has an administrative agent, and all places have hours of operations, in this case, Saturday from 6pm to 10pm. The place type has a contact rate of 100 (this is just for illustration purposes -- not based on observational data.)

The **add** property is used to define three specific places, corresponding in this case to actual concert venues in Pittsnurgh, PA. Some global variables are defined so that the rest of the program can refer to these places by their symbolic names.

The model then defines a condition called **CONCERT** that corresponds to an individual's state of participation at concerts. The **Start** start selects some individuals who attend concerts in these venues. These agents then proceed to the **Select** state where they select their next concert venue. Agents then **Wait** for the next Saturday at 6pm, and if their concert is open, they **Attend** the concert. When the concert is **Over,** the customer leaves the concert (that is, quit being a member of the concert), and returns to the **Select state.**

The admin agents start in the **Schedule** state and stay there for a week. Since the agent's conceert is open by default, this amount to holding a concert during that week. The admin agent then **closes** the concert and waits a period of time before returning to the **Schedule** state and re-opening the concert.

**#####################################################**

**#**

**# A model of mass gatherings -- concerts**

**place Concert {**

**has\_administrator = 1**

**starts\_at\_hour\_18\_on\_Sat = 4**

**contacts = 100.0**

**# Heinz Field:**

**add = 100 40.446686 -80.015851 200.0**

**# Stage AE**

**add = 101 40.446160 -80.012300 200.0**

**# Peterson Event Center:**

**add = 102 40.444016 -79.962144 200.0**

**}**

**global HeinzField StageAE Peterson**

**# give each venue a name**

**HeinzField = 100**

**StageAE = 101**

**Peterson = 102**

**my myConcert**

**myConcert = 0**

**condition CONCERT {**

**states = Start Select Wait Attend Over Schedule Reschedule Excluded**

**admin\_start\_state = Schedule**

**}**

**state CONCERT.Start {**

**if (range(age, 18, 60)) then next(Select) with prob(0.02)**

**default(Excluded)**

**}**

**state CONCERT.Select {**

**set(myConcert, HeinzField)**

**wait(24)**

**next(Wait)**

**}**

**state CONCERT.Wait {**

**join(Concert, myConcert)**

**wait(until\_Sat\_at\_6pm)**

**if (open(Concert)) then next(Attend)**

**default(Select)**

**}**

**state CONCERT.Attend {**

**wait(4)**

**next(Over)**

**}**

**state CONCERT.Over {**

**quit(Concert)**

**wait(24)**

**next(Select)**

**}**

**# administrative states:**

**# Example: hold a concert once a week, every 20 days**

**state CONCERT.Schedule {**

**wait(24\*7)**

**next(Reschedule)**

**}**

**state CONCERT.Reschedule {**

**close(Concert)**

**wait(24\*13)**

**default(Schedule)**

**}**

The particular rules used here could be replaced with more elaborate rules as needed to model concert-going behavior. For this example, we just want to illustrate that large crowds attend the concerts when they are open, and other conditions can be transmitted within the crowds at a concert.

First let's verify that actions of the admin agents. The next plot shows the number of concerts open as a function of time. It is clear that all three venues are scheduled for business during the first week after each 20-day period.



Next we plot the number of individual in the **Attend** state, which shows that about 14,000 guests attend concerts early in each 20-day period.



Now we run a model that includes the influenza condition described previously. Since influenza is transmitted by proximity, it is possible that infectious agents can transmit influenza to susceptible agents who share a place with the infectious agents. In this model, we have defined the. contact rate for concerts to be 100, so each infectios agents is expected to make about 100 potentially transmissible contacts during each hour at the concerts. When we plot out the incidence curve for influenza showing the number of transmission events at concerts, we see the spikes of transmissions occurring during the same periods of time when the guests were attending the concerts:



Why is there no significant spike after the concert on day 64? In this simple model, all 14,000 people attend the same conference at once, and the earlier concerts appear to have exhausted the supply of susceptibles. This is unlikely to happen in more realistic forms of this model.

Please see the directory **FRED/models/concert** for the files for this example. The METHODS file contains the commands used to generate the plots shown above.

# Chapter 14: Applications

This Chapter include several complete FRED programs along with sample of outputs.

## Pandemic Influenza

The following program is included in the FRED library. It comprises a simple S-E-I-R model of pandemic influenza. Influenza is spread largely through inhalation of viral particles shed by infectious individuals, so the transmission mode is by proximity (i.e., sharing the same place). A pandemic disease is one that spreads within many countries, and usually means that immunity is very low across the human population. In this model, we assume that all individuals are completely susceptible to the disease. This model was used to calibrate the place-specific contact rates in FRED; therefore, the transmissibility of the INF Condition is set to 1.0.

In this model, we assume that 10 randomly selected individuals are exposed at the start of the simulation. It is futher assumed that infectious individuals are all equally infectious (e.g., no "super-spreaders" are included in the model). Infected individuals become infectious after a period of time drawn from a lognormal distribution with median of 1.9 days and dispersion of 1.23 days (citation). Two-thirds of infectious individuals become symptomatic, and one-third become infectious but asymptomatic. It is assumed that asymtpomatic individual are half as infectious as symptomatic cases. The infectious phase lasts for a median duration of 5 days, with a lognormal dispersion of 1.5 days. This means that majority of infectious periods lasts between about 3.3 and 7.5 days. After the infectious phase, individuals recover and remain immune to influenza for the duration of th simulation. Case mortality is not modeled.

The model includes a simple model of agent behavior based on their disease status: one half of all individuals with influenza symptoms are assumed to remain confined to the household for the duration of their illness. After recovery, the agents resume their normal schedule of activities.

**#### Simulated Location**

**locations = Jefferson\_County\_PA**

**##### Simulated Timeframe**

**start\_date = 2020-Jan-01**

**end\_date = 2020-Apr-10**

**condition INF {**

**states = S E Is Ia R Import**

**import\_start\_state = Import**

**transmission\_mode = proximity**

**transmissibility = 1.0**

**R0\_a = 0.0398238**

**R0\_b = 0.611043**

**S.is\_dormant = 1**

**R.is\_dormant = 1**

**exposed\_state = E**

**}**

**state INF.S {**

**set\_sus(INF,1)**

**wait()**

**next()**

**}**

**state INF.E {**

**set\_sus(INF,0)**

**wait(24\*lognormal(1.9,1.23))**

**next(Is) with prob(0.67)**

**next(Ia) with prob(0.33)**

**}**

**state INF.Is {**

**set\_trans(INF,1)**

**set\_state(StayHome,No,Symptoms)**

**wait(24\* lognormal(5.0,1.5))**

**next(R)**

**}**

**state INF.Ia {**

**set\_trans(INF,0.5)**

**wait( 24 \* lognormal(5.0, 1.5) )**

**next(R)**

**}**

**state INF.R {**

**set\_trans(INF,0)**

**set\_state(StayHome,Yes,No)**

**wait()**

**next()**

**}**

**state INF.Import {**

**import\_count(10)**

**wait()**

**next()**

**}**

**condition StayHome {**

**states = No Symptoms Yes**

**}**

**state StayHome,No {}**

**state StayHome.Symptoms {**

**wait(0)**

**next(Yes) with prob(0.5)**

**default(No)**

**}**

**state StayHome.Yes {**

**absent()**

**present(Household)**

**wait()**

**next()**

**}**

Suppose that we run 8 simulations of this model under the job name **baseline**. Then we can produce the following plots using the command lines shown (or via the FRED Web interface).

% fred\_plot --key baseline --var INF.newE --bars --weekly



% fred\_plot -k baseline -v INF.newE --all



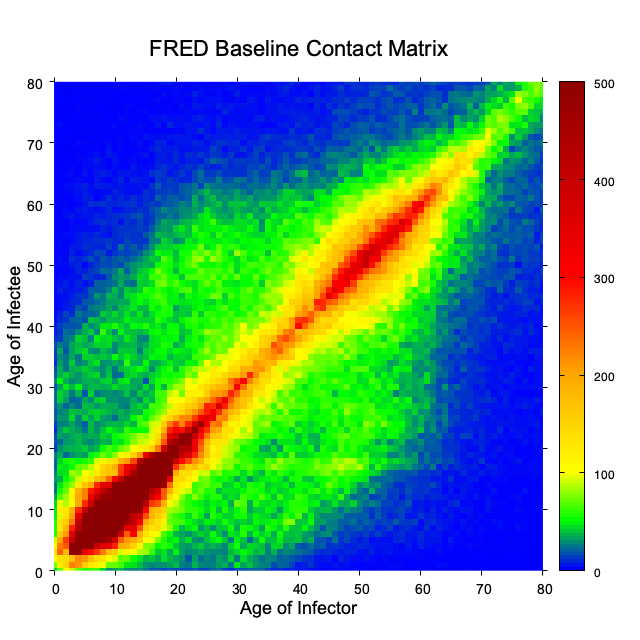
# Chapter 15: Calibration

## THE FRED Contact Matrix

A contact matrix is a square matrix showing the rate of contact between people of various ages.

A FRED baseline contact matrix was generated by running a simple SEIS model in which infections only last 1 day and the infection rate was lower than normal influenza, thus producing very short chains of infections. New cases were imported to 10% of the population every 10 days over one year. This heuristc method gives everyone an even chance to be infected by any one of their contacts. The population in this study was Allegheny County, PA.

The plot shows the age of infector (x-axis) and infectee (y-axis).



As expected, most contacts are among age-similar pairs, with the most contacts among school age children (age 5-18).

The least emphasis on same-age contacts appears to be among the 30-45 yrs olds. Individuals in this age group seem to have their contacts spread out more evenly between their younger kids and their older workmates.

# Acknowledgments

FRED Team

* John Grefenstette, lead architect
* Mary Krauland, David Galloway, Mike Lann, Robert Frankeny, David Sinclair
* Mark Roberts, Don Burke

Support

* NIH / NIGMS (MIDAS Program)
* Bill and Melinda Gates Foundation
* Robert Wood Johnson Foundation
* Benter Foundation

# Appendix A: The Structure of the FRED SImulation Engine

FRED is designed to enable users to build agent-based models without computer programming. To this end, the FRED language supports the addition of new Conditions and interaction rules without altering the underlying simulation software. However, it is anticipated that changes to the underlying software may be required from time to time. This Chapter is intended for developers who want to extend the capabilities of the FRED system itself.

The FRED Core simulation engine is written in C++. This system was built using a modular, object-oriented programming approach. FRED consists of five interacting modules. Each module consists of several classes, described in the following sections.

## Core Module

The Core Module provides the fundamental data structures used in FRED, including global data types, utilities for processing strings and other data types, date-related functions, event queues, rnadom number generators, properties of state spaces, the FRED language parser and compiler, including properties, expressions and rules. The Core Module comprises the following classes:

### Fred

### Global

### Age\_Report

### Utils

### Date

### Events

### Random

### State\_Space

### Parser

### Factor

### Expression

### Predicate

### Clause

### Rule

## Geospatial Module

The Geospatial Module provides the geo-spatial data structures used in FRED, including utilities for processing location data, administrative units including states, counties, census\_tracts and block\_groups. It also provides classes for producing visualization data. The Geospatial Module comprises the following classes:

### Geo

### Abstract\_Grid

### Abstract\_Patch

### Admin\_Division

### State

### County

### Census\_Tract

### Block\_Group

### Neighborhood\_Layer

### Neighborhood\_Patch

### Regional\_Layer

### Regional\_Patch

### Visualization\_Layer

### Visualization\_Patch

## Agent Module

The Agent Module provides data and methods related to individual agents. The Agent Module comprises the following classes:

### Person

### Demographics

### Link

### Travel

### Preference

## Mixing Module

The Mixing Module provides mixing groups that permit agents to interact with one another, including networks and places. The Mixing Module comprises the following classes:

### Group\_Type

### Place\_Type

### Network\_Type

### Group

### Place

### Network

### Household

### Hospital

## Epidemic Module

The Epidemic Module provides data and methods related to the Conditions being tracked in the population of agents. It includes the defintion of Conditions, including the natural history of each Condition, various transmission modes for spreading Conditions across the population, and methods for updating agents during each time step. The Epidemic Module includes the classes:

### Condition

### Epidemic

### Natural\_History

### Transmission

### Environmental\_Transmission

### Network\_Transmission

### Proximity\_Transmission

# Appendix B: Errors and Warnings

The following is a list of error messages that FRED generates. If any of these errors occurs the model will not execute. The fred\_compile script prints the line in the model that generates each of these error messages.

**"Action set needs two arguments: "**

**"Bad AND clause: "**

**"Bad admin\_start\_state:"**

**"Can't parse action rule "**

**"Can't parse default rule: "**

**"Can't parse exposure rule "**

**"Can't parse rule: "**

**"Can't parse state in rule: "**

**"Can't parse state rule: "**

**"Can't parse wait rule "**

**"Count\_all\_import\_attempts takes no arguments: "**

**"Destination state not recognized: "**

**"Distance expression not recognized:"**

**"Expression not recognized: "**

**"Global list var expression not recognized:"**

**"Global var expression not recognized:"**

**"Group name not recognized: "**

**"Group not recognized: "**

**"Import\_census\_tract rule needs 1 argument: "**

**"Import\_count rule needs 1 argument: "**

**"Import\_list rule needs 1 argument: "**

**"Import\_per\_capita rule needs 1 argument: "**

**"Index expression not recognized:"**

**"List Variable expression not recognized:"**

**"List expression not recognized:"**

**"List index expression not recognized:"**

**"List\_var not recognized: "**

**"Max degree expression not recognized: "**

**"Mean degree expression not recognized: "**

**"Need a list-valued expression: not recognized: "**

**"Needs 2 arguments: "**

**"Needs 3 arguments: "**

**"Network not recognized: "**

**"No Next State: "**

**"No THEN clause found "**

**"No unconditional wait rules found for state "**

**"No wait rule found for state "**

**"Person Expression not recognized: "**

**"Second arg to join not recognized: "**

**"Select function needs 2 arguments:"**

**"Source condition not recognized: "**

**"Source state not recognized: "**

**"Unknown Rule Action: "**

**"Value Expression not recognized: "**

**"Value expression not recognized:"**

**"Value function needs 2 arguments:"**

**"Var not recognized: "**

**"Variable expression not recognized:"**

**"import\_ages rule needs 2 argument: "**

**"import\_location rule needs 3 argument: "**

**"set\_contacts rule needs 1 argument: "**

The following is a list of warning messages that FRED generates. Warnings are intended to alert the user of situations that may need to be checked by the user, but that do not prevent the model from running. The fred\_compile script prints the line in the model that generates each major warning. Minor warning messages are printed in the LOG file.

**"become\_a\_teacher: person %d age %d ineligible -- already goes to school %d"**

**"check these entries carefully. they may be misspelled."**

**"classroom\_assign classroom returns null: person %d age %d school"**

**"delete\_events: item not found"**

**"duplicate household label found:"**

**"duplicate school label found:"**

**"duplicate workplace label found:"**

**"fred warning (file %s, line %d) ignored duplicate property statement: %s ="**

**"fred warning (file %s, line %d) unrecognized property statement: %s ="**

**"house %d label %s has zero size"**

**"no head of household found for household id %d label %s groupquarters: %d"**

**"no head of household found for household id %d label %s size %d groupquarters: %d"**

**"no school found on day %d in admin\_code = %d grade = %d schools = %d r = %f sum = %f"**

**"no school found on day %d in admin\_code = %lld grade = %d schools = %d r = %f sum = %f"**

**"no school vacancies found on day %d in admin\_code = %d grade = %d schools = %d"**

**"no school vacancies found on day %d in admin\_code = %lld grade = %d schools = %d"**

**"no transition rules found for state"**

**"no workplace available for person"**

**"office no office assigned for person %d workplace %d"**

**"person %s -- no school found for label ="**

**"person %s -- no workplace found for label ="**

**"person %s age %d is too old to attend school"**

**"person %s assigned to workplace"**

**"place %d %s has bad patch, lat = %f (not in [%f, %f]) lon = %f (not in [%f, %f])"**

**"skipping person"**

1. Strictly, this value is not yet a probability, since it will be combined with other probabilities from other rules to finally form a valid probability distribution. This is explained a little later. [↑](#footnote-ref-1)