Chapter 12

Modeling with Agents

The modeling techniques we have taught up until this point focused on gaining insights using highly aggregated models of a system. This means that when we looked at models of population growth, we did not explore individual people and instead focused on understanding the population as a whole. This high-level aggregate approach to modeling helps us cut through unnecessary details to understand the core dynamics of a system.

For certain models however, this high-level view may hamstring our ability to explore important questions. For instance in a disease model we may care about the physical relationship between people in the model. Are they near each other? How often do they come into contact? Can we attempt to control the disease by manipulating how people move about and relate to each other? These are all questions that are very hard to answer with a standard System Dynamics model.

Heterogeneity, differences between individuals, is difficult to represent using System Dynamics models. One approach to heterogeneity that is sometimes used is simply to duplicate the model structure for each different class of person or entity in the model. We recall seeing one model that explored education in the United States. The modelers wanted to explore the differences between male and female students. To do so, they simply copy and pasted the entire model structure (consisting of dozens of stocks and flows) and calibrated one of these copies for male students and the other copy for female students.

Granted, this approach can be made to work, but it requires a lot of effort to set up and configure even in the simple two-gender case. When you have more than two cases it can quickly become completely unmanageable. Furthermore, duplicating parts of your model is a recipe for creating unmaintainable models afflicted by hard to track down bugs. The reason for this is that when you later make changes to your model, you are going to need to ensure the changes are made correctly to each one of the model copies. Although simple in principle, in practice this is very easy to mess up and it is a direct route for bugs to be introduced into the model.

Fortunately, an alternative modeling paradigm to System Dynamics exists that is excellent for modeling discrete individuals. It is called Agent Based Modeling and is focused on simulating individual agents and the interactions between these agents[1]. In this chapter we will introduce Agent Based Modeling and show how you can use it to explore questions that cannot be answered with pure System Dynamics.

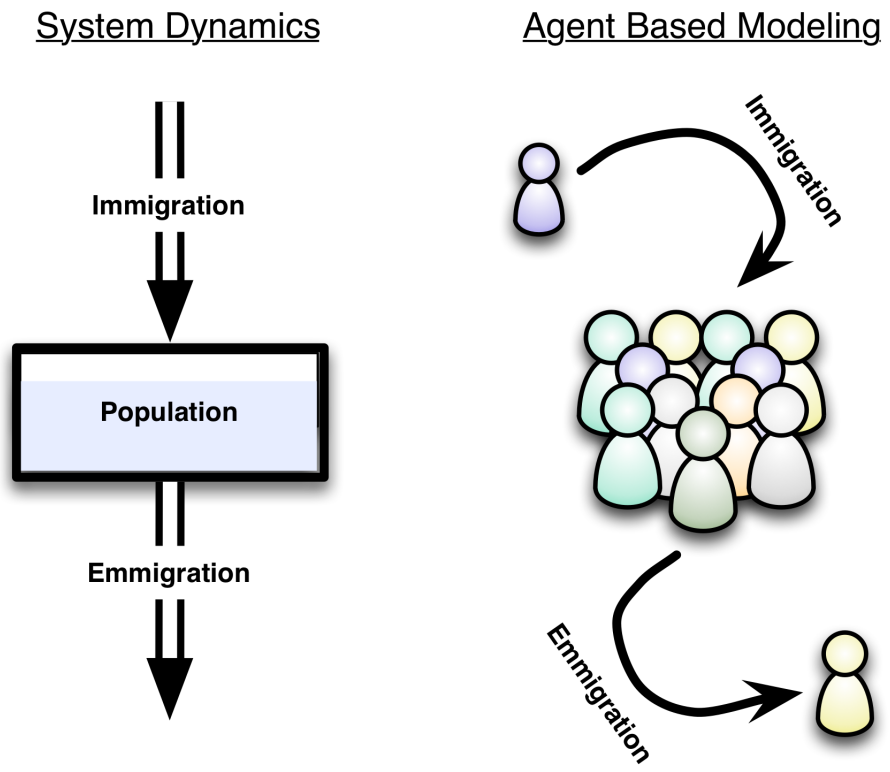


Figure 1. Two paradigms for modeling a population: System Dynamics and Agent Based Modeling.

[1]System Dynamics also has another standard tool for dealing with heterogeneity. This tool is called "arrays" or "indexing" and allows you to transparently create multiple copies of your model during simulation to match different classes. Arrays are not as flexible as fully Agent Based Models though. If you consider the continuum of fully aggregate System Dynamics models on one end to fully individualized Agent Based Models on the other, we can think of arrays as existing part way along this continuum.

The State Transition Diagram

Up until now our primary modeling tool has been the stock and flow diagram. This type of diagram is useful for summarizing systems from a high-level viewpoint. The stock is a primitive that can model entities that take on a range of values and flows are well suited for specifying the changes in stocks

In addition to representing aggregate systems, stock and flow diagrams are also used to model things on an individual level. For instance, a model of a person's motivations could be represented using a stock and flow diagram. The strength or importance of each type of motivation – money, family, etc... – could be represented as stocks with flows modulating the strength of these motivations over time.

When looking at the individual scale however, we will oftentimes find ourselves wanting to define characteristics of the individual using simple on/off logic. For instance, take the issue of an individual's sex. We can represent this using two categories: Male or Female (leaving aside transgendered individuals for the sake of simplicity). Similarly, when constructing a model of a disease, we might want to say a person is either sick or not sick (with no nuances such as "slightly sick" or "highly sick"). You can attempt to represent these different categories using stocks, but the formulation and equations to do so will be overly complicated.

Where the stock and flow diagram is used to model changing systems with continuous stocks, the state transition diagram is used to model systems with discrete on/off states. Within Insight Maker, state transition diagrams are constructed in almost the same way as stock and flow diagrams. The key difference is all stocks are replaced with *State* primitives and all flows are replaced with *Transition* primitives. State primitives can be added to the model by right-clicking on the model diagram and selecting **Add State**. Transition primitives will automatically be created when you connect two state primitives together using the standard "Flow" connection type.

A state primitive is possibly the simplest primitive available as it can only take on one of two values: true or false. When the state value is true, the state is active. When the state value is false, the state is not active and the agent does not occupy that state. When configuring a state primitive, you only need to specified whether the state is **initially active** or not at the start of the simulation. This initial condition can simply be `true` or `false`, but it can also be a logical equation that depends on the values of other primitives in the agent. For example, if you had a variable in the agent called [**Size**], and you wanted a state to be initially active if the value of [**Size**] was greater than 5, you could use the following as the initially active property for the state: `[Size] > 5`.

A transition primitive moves an agent between states. For instance, if you had two states in your model – *Healthy* and *Sick* – you could have one transition primitive moving agents from the healthy state to the sick state (simulating infection) and another transition primitive moving them the other way (simulating recovery).

Figure 2. Sample states you might use for agents in a disease model.

There are three different ways a transition from one state to another can be triggered:

Timeout : In this mode the transition will be triggered a specific amount of time after the first state becomes active. For instance, if we had a disease model where the disease lasted 10 days, we could have a transition from the sick to healthy state using a timeout trigger with a period of 10 days.

Probability : In this mode there is a probability of the transition happening each time period. For instance, in the disease model if the disease only lasted 10 days on average but could randomly last longer or shorter, you could use a probability transition with a daily probability of 0.1.

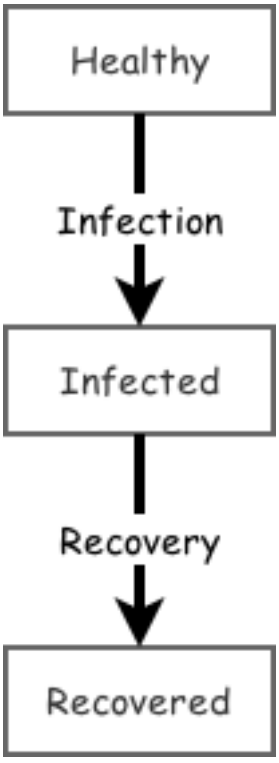
Condition : In this mode you create an equation that will trigger than transition when it becomes true. For instance, if we had a stock, [**Infection Level**] in our agent indicating how sick the agent was, we could have them transition out of the sick state once that stock fell to zero. The trigger condition to enable this could be something like: `[Infection Level] = 0`.

A State Transition Diagram for Disease

This model illustrates the use of state transition diagrams to model a simple disease. This is a disease such as the flu where immunity is obtained once the individual recovers from the disease.

1. Create a new **State** named [**Healthy**].
2. Create a new **State** named [**Infected**].
3. Create a new **State** named [**Recovered**].
4. The model diagram should now look something like this:

5. States represent the condition someone is in. So in our model a person can either be healthy, infected, or recovered from the infection. Now, let's add transitions that move a person from state to state.
6. Create a new **Transition** going from the primitive [**Healthy**] to the primitive [**Infected**]. Name that transition [**Infection**].
7. Create a new **Transition** going from the primitive [**Infected**] to the primitive [**Recovered**]. Name that transition [**Recovery**].
8. Please note that in this model someone who is recovered cannot become sick again. They have gained immunity to the disease.
9. Now that the model structure has been designed, let's add equations and configure the primitives.
10. Change the **Start Active** property of the primitive [**Healthy**] to `True`.
11. The model diagram should now look something like this:



12. When a state is active, it means a person is in that state. By setting [**Healthy**] to start active, we have the person start in the healthy state.

13. Change the **Trigger Type** property of the primitive [**Infection**] to `Probability`.

14. Change the **Value/Equation** property of the primitive [**Infection**] to `0.3`.

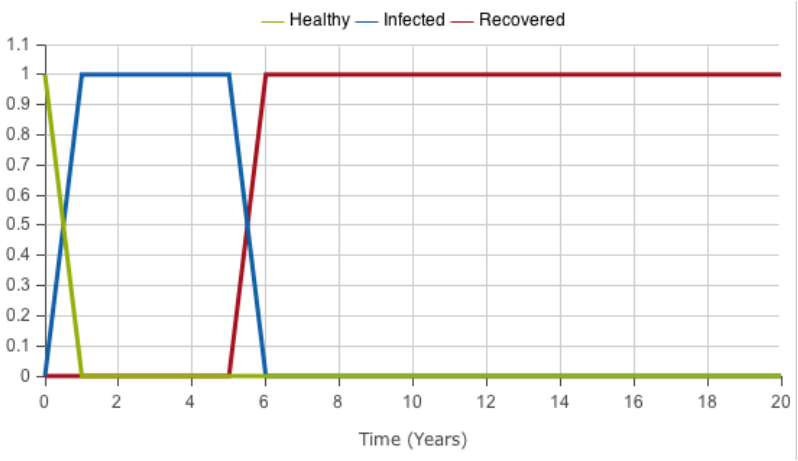
15. Change the **Trigger Type** property of the primitive [**Recovery**] to `Probability`.

16. Change the **Value/Equation** property of the primitive [**Recovery**] to `0.2`.

17. Using the Probability type for the transition trigger means that the person has a fixed probability of transitioning from one state to the next each year. We will assume a 30

18. Let's run the model now.

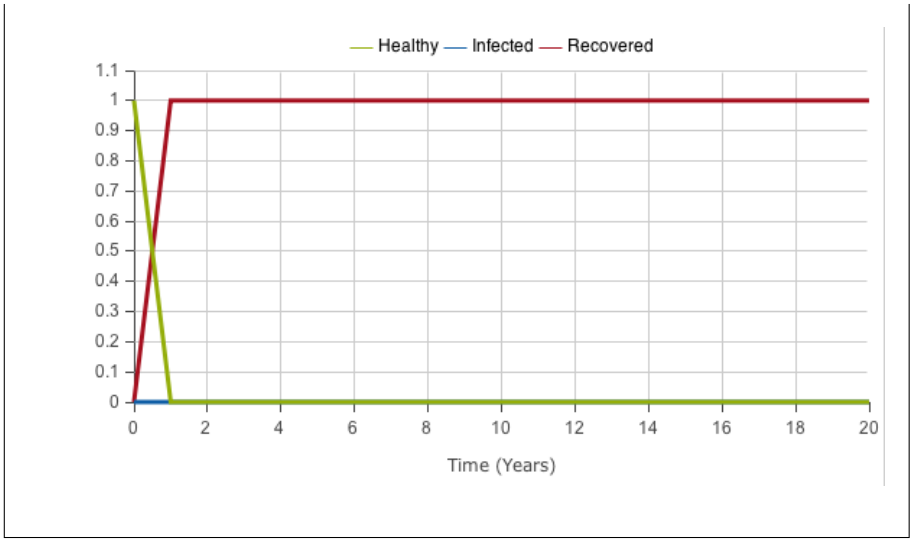
19. Run the model. Here are sample results:



20. A value of 1 for a state primitive means it is active. A value of 0 means it is not active. We can see from this diagram when the individual transitions from the healthy to the infected state and then from the infected state to recovered state.

21. We can run the model again and we will see that we get different results each time we run it. This is because the model is stochastic and the transition triggers are random.

22. Run the model. Here are sample results:



Creating Agents

Now that we have learned about state transition diagrams, we are ready to start creating agents. There are three key parts of creating agents in a model:

1. Defining what an agent is
2. Creating a group of agents
3. Viewing agent results

Defining Agents

We have already introduced the folder primitive as a tool for grouping primitives together and also as a tool for unfolding a model. The folder primitive plays an additional role in Agent Based Modeling as we use folders to define what our agent consists of.

To create an agent construct the state transition diagram for your agent (and also add any stocks, flows or any other primitives you want to this agent). Then create a folder containing all these primitives. Give the folder the name of your agent such as “Person” or “Individual” or even just “Agent”. This is all similar to what we have done with folders before, but now there is one extra step. Edit the folder configuration and set the folder **Behavior** to “Agent”. You have now created the definition of your first agent!

You can have as many different types of agents in your model as you would like. Just create a new agent model and use a new folder to define each of the different types of agents. For instance if you had a predator prey model

you could have one agent definition describing the behavior of the prey, and a second agent definition describing the behavior of the predators.

Creating a Population of Agents

After you have defined an agent in your model, you are ready to create a collection or population of agents. This is done by adding an *Agent Population* primitive to your model. The agent population primitive takes the definition of an agent from an agent folder and creates many copies of that agent from the definition. The agent population primitive keeps track of these copies and allows them to operate and to interact with one another.

There are a number of different settings for the agent population primitive but two of them are of key importance. The first is to select what type of agent will be in the population. Each population primitive can only have one type of agent in it. You can have multiple populations though and the agents in one population can interact with the agents in another population.

After specifying what type of agent is in the population, you need to specify how many agents are in the population at the start of the simulation. This is done by setting the **Size** property for the agent population. Later on you can add or remove agents to a population by using the `Add()` and `Remove()` functions.

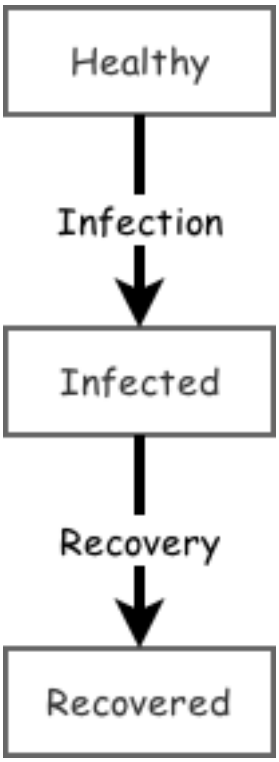
Viewing Agent Results

Many of the standard Insight Maker display types can be used to show the results of an agent based simulation. If you add an agent population to a time series or tabular display, the results for the number of agents in each of the various agent states will automatically be shown. You can also use the **Map** display type to illustrate agents within a geographic region.

An Agent Based Model of Disease

Here we convert a state transition diagram into a model containing multiple agents.

1. The model diagram should now look something like this:



2. We start with our state transition diagram from our previous modeling example.

3. Create a new **Folder** named [**Person**]. The folder should surround the primitives [**Healthy**], [**Infected**], [**Recovered**], [**Infection**] and [**Recovery**].

4. Change the **Type** property of the primitive [**Person**] to `Agent`.

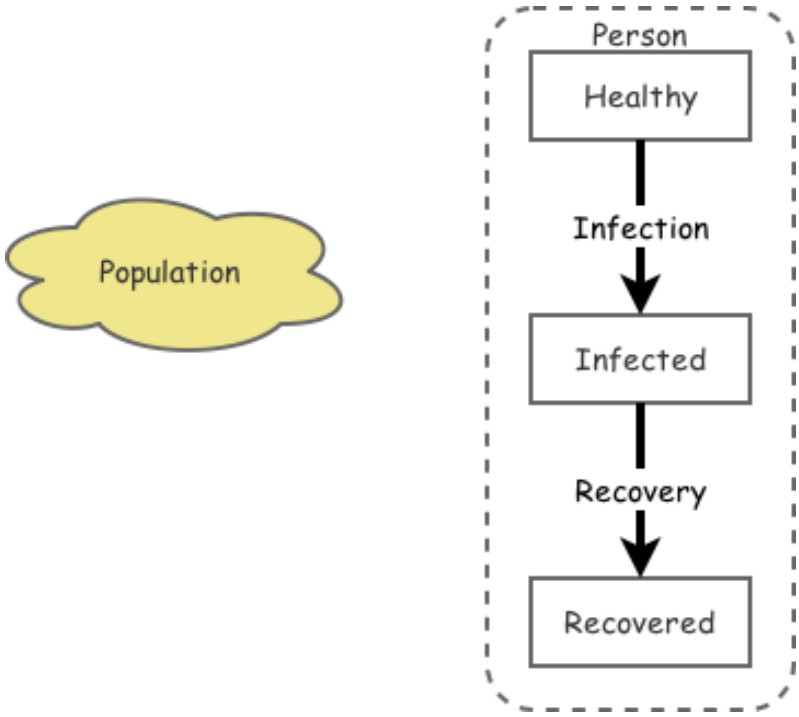
5. First we create an agent folder to encapsulate our state transition diagram. This is a definition of what an agent in our model will be and we make sure the folder behavior is set to "Agent".

6. Create a new **Agent Population** named [**Population**].

7. Change the **Agent Base** property of the primitive [**Population**] to `Person`.

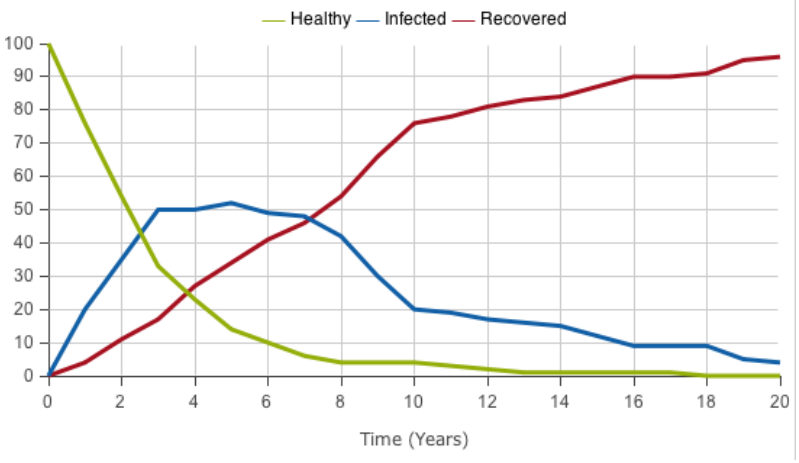
8. Next we create an agent population [**Population**] and set it to contain instances of our **Person** agent. We'll start with a population size of 100 agents.

9. The model diagram should now look something like this:



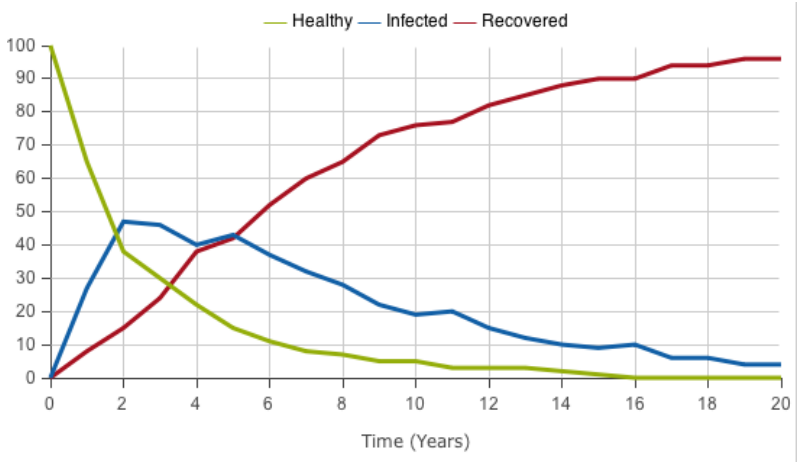
10. We can now run the model to see how the disease affects the 100 people in our population.

11. Run the model. Here are sample results:



12. Each time we run the model we will get different results due to the stochasticity in the model.

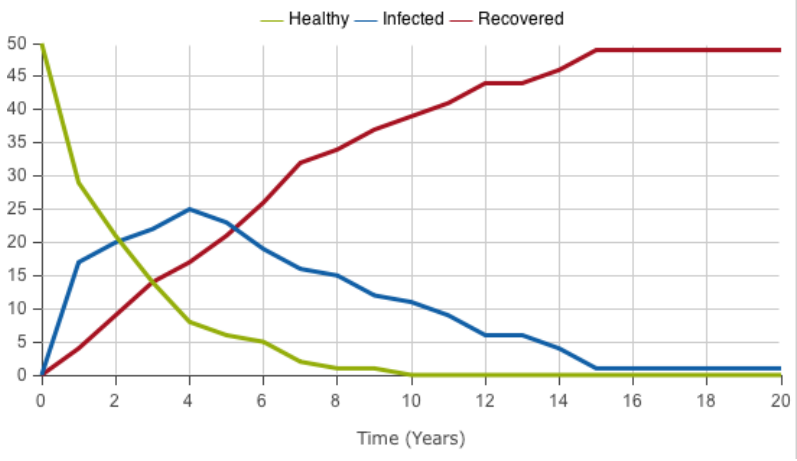
13. Run the model. Here are sample results:



14. Change the **Size** property of the primitive [**Population**] to `50`.

15. We can easily change the number of people in the population. Let's set it to 50 then run the model again.

16. Run the model. Here are sample results:



17. When we have a smaller number of people, the overall population changes are more variable. As we add more and more people, the randomness in the model has less of an effect and trends become smoother approaching the average results we would see if we had used a System Dynamics model.

Working with Agents

Working with agents is fundamentally different from working with primitives in a pure System Dynamics your model. For instance, if you have a regular model and refer to the value of a variable or stock you get a single value back. With agents, however, when you refer to the value of a primitive you might get a separate value for each individual agent in your model. So for instance, if you have 100 agents and you refer to the primitive [**Size**], you will get100 different sizes one for each of the agents in the model.

You will need to extend your modeling toolkit in order to be able to effectively manage agents and accomplish your goals in your model. The key building block of this extended toolkit is the vector[2]. In the following sections we will first introduce the general concept of vectors and then show how you can use them to interact with agents.

Working with Vectors

A vector is an ordered list of items. In Insight Maker vectors can be written using the '«' sign (or '<<') followed by the '»' sign (or '>>'). For instance the following is a vector of the first four integers:

```
«1, 2, 3, 4»
```

Insight Maker has an extensive set of capabilities and functions for manipulating vectors. For instance you can do multiplication using vectors:

```
2*«1, 2, 3, 4» # = «2, 4, 6, 8»
```

Or do elementwise multiplication:

```
«1, 2, 1, 2»*«1, 2, 3, 4» # = «1, 4, 3, 8»
```

Or addition using vectors:

```
2+«1, 2, 3, 4» # = «3, 4, 5, 6»
```

Again, we can also do addition elementwise:

```
«1, 2, 1, 2»+«1, 2, 3, 4» # = «2, 4, 4, 6»
```

There are also a large number of functions you can use to summarize vectors. For instance, you can find the average value in a vector:

```
mean(«1, 2, 3, 4») # = 2.5
```

Or the number of elements in a vector:

```
count(«1, 2, 3, 4») # = 4
```

There are also functions you can use to change the ordering of elements in a vector:

[2]In other programing languages and modeling environments vectors are sometimes called "Arrays" or "Lists".

```
reverse(«1, 2, 3, 4») # = «4, 3, 2, 1»
```

```
sort(«3, 4, 2, 1») # = «1, 2, 3, 4»
```

A couple of very useful functions are available to combine vectors together:

```
Union(«1, 2 ,3», «2, 3 ,4») # = «1, 2, 3, 4»
```

```
Intersection(«1, 2 ,3», «2, 3 ,4») # = «2, 3»
```

```
Difference(«1, 2 ,3», «2, 3 ,4») # = «1, 4»
```

And lastly there are two very powerful vector functions we should mention: `Map()` and `Filter()`. Map takes each element in a vector and applies some transformation to it and returns a vector of the transformations. For instance, if we wanted to find the square of each element in a vector we could use the following:

```
Map(«1, 2, 3, 4», x^2) # = «1, 4, 9, 16»
```

Here the function `x^2` is applied to each element in the vector (with `x` representing the element value) and the results are returned.

Filter takes a function and applies it to each element in a vector. If the function evaluates to true, the element is included in the resulting vector; if the function evaluates to false, the element is not included in the results. For instance, the following function filters out all elements that are smaller than 2:

```
Filter(«1, 2, 3, 4», x >= 2) # = «2, 3, 4»
```

There are many more vector functions available, but these are some of the key ones. They will prove invaluable when you come to working with vectors of agents.

Accessing Agents

Insight Maker includes a number of functions to access the individual agents within a population. The simplest of these is the `FindAll()` function. Given an agent population primitive that we'll call [**Population**], the FindAll function returns a vector containing all the agents within that agent population:

```
FindAll([Population])
```

So if your agent population currently had 100 agents in it, this would return a vector with 100 elements where the first element referred to the first agent, the second element referred to the second agent and so on. It is important to note that these elements are agent references, not numbers. So you can use a function like `Reverse()` on the resulting vector, but you cannot directly use functions like `Mean()` or `Sort()` as the agent references are not numerical values[3]. We will see how to access the values for agents next.

[3]The agents certainly contain many numerical values in their stocks, variables, or states; but an agent reference itself is not numerical and so you cannot do things such as directly taking the average of the agents or sorting them.

In addition to the FindAll function, there are other find functions that return a subset of the agents in the model. For instance, the `FindState()` and `FindNotState()` functions return, respectively, agents that either have the given state active or not active. For instance, if our agents had a state primitive called [**Smoker**] that represented if the agent was a smoker or not, we could get a vector of the agents in our population that were smokers using the following:

```
FindState([Population], [Smoker])
```

And we could obtain a vector of the agents that were not smokers with:

```
FindNotState([Population], [Smoker])
```

Find functions can also be nested. For instance, if we wanted a vector of all male smokers, we could do something like the following:

```
FindState(FindState([Population], [Smoker]), [Male])
```

Nesting find statements is effectively using Boolean AND logic (like you might use on a search engine: "Smoker AND Male"). To do Boolean OR logic (e.g. "Male OR Smoker") and return all the agents that are either a smoker or a man (or both), you can use the Union function to merge two vectors:

```
Union(FindState([Population], [Smoker]), FindState([Population],
[Male]))
```

If you wanted the agents that were either smokers or men (but not both simultaneously), you could use:

```
Difference(FindState([Population], [Smoker]), FindState([Population],
[Male]))
```

Agent Values

Once you have a vector of agents, you can extract the values of the specific primitives in those agents using the `Value()` and `SetValue()` functions.

The Value function takes two arguments: a vector of agents and the primitive for which you want the value. It then returns the value of that primitive in each of the agents. For instance, let us say our agents have a stock primitive named [**Age**]. We could get a vector of the age of all the people in the model like so:

```
Value(FindAll([Population]), [Age])
```

A vector of ages by itself is generally of not too much use. Often we will want to summarize it, for instance by finding the average age of the people in our population:

```
Mean(Value(FindAll([Population]), [Age]))
```

In addition to determining the value of a primitive in an agent, you can also manually set the agents' primitive values using the SetValue function. It takes the same arguments as the Value function in addition to the value you want to

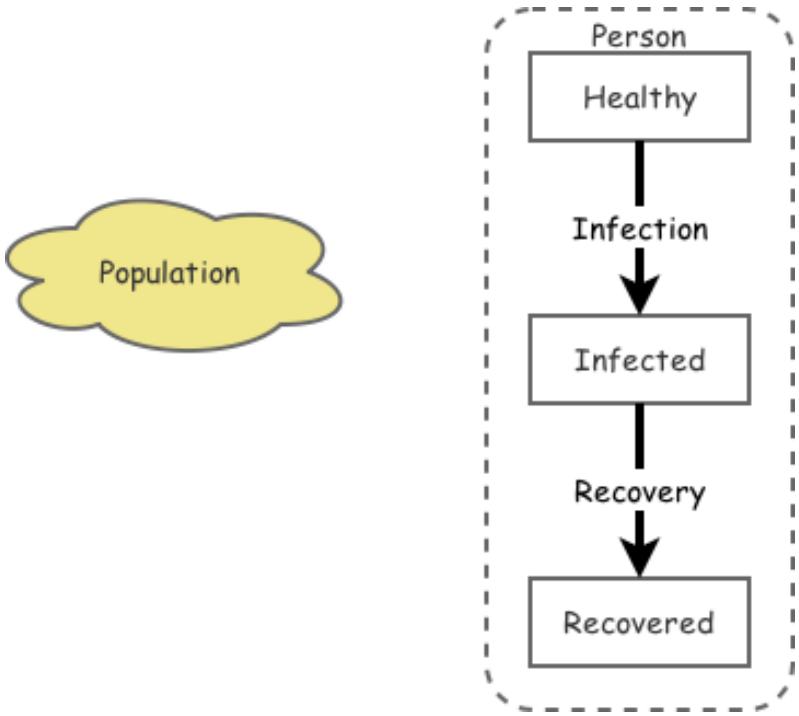
set primitives to. For instance, we could use the following to set the age of all our agents to 25:

```
SetValue(FindAll([Population]), [Age], 25)
```

Agents Interacting

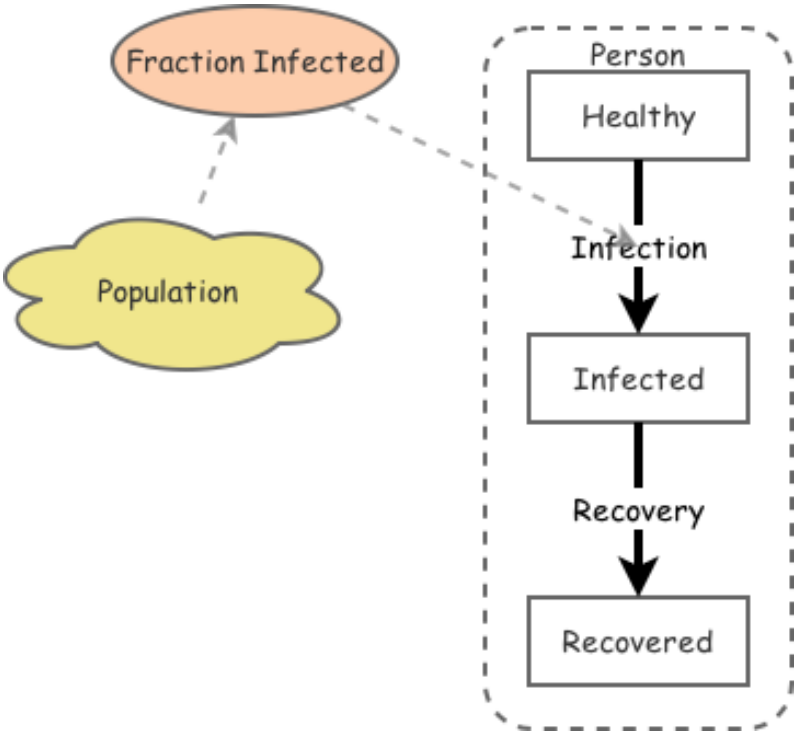
This example shows how agents can interact with each other using the Find functions.

1. The model diagram should now look something like this:



2. Let's make our agent based disease model from earlier more realistic. We will add a variable [**Fraction Infected**] that calculates what fraction of the population is currently infected. We will then use this variable to determine the infection rate so the more people in the population who are infected, the faster the disease will spread.
3. Create a new **Variable** named [**Fraction Infected**].
4. Create a new **Link** going from the primitive [**Population**] to the primitive [**Fraction Infected**].
5. Create a new **Link** going from the primitive [**Fraction Infected**] to the primitive [**Infection**].

6. The model diagram should now look something like this:



7. Now let's configure the value of [**Percent Infected**] and change the [**Infection**]transition to use it.

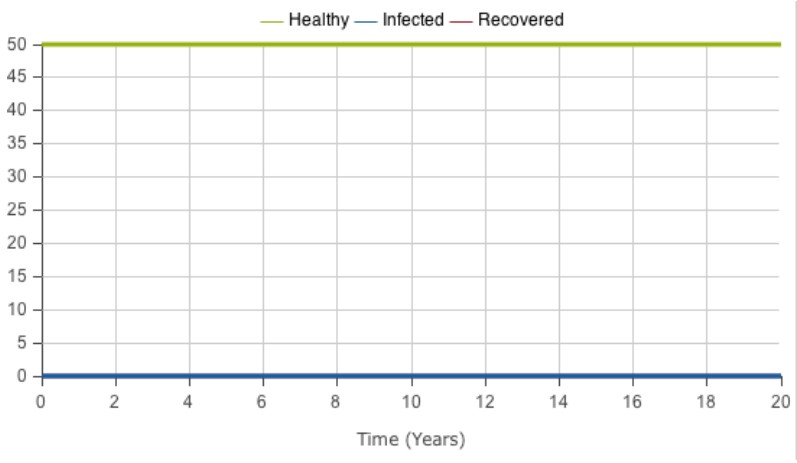
8. Change the **Equation** property of the primitive [**Fraction Infected**] to `Count(FindState([Population], [Infected]))/PopulationSize([Population])`.

9. This equation uses the FindState function to select all the people in the [**Population**] who are in the Infected state. It then divides the number of those people by the total size of the population.

10. Change the **Value/Equation** property of the primitive [**Infection**] to `[Fraction Infected]`.

11. Now that we have set our infection probability to the value of the [**Fraction Infected**] primitive, we are ready to run the model.

12. Run the model. Here are sample results:



13. That was a bit of a disappointment wasn't it? Nothing happened. Why is this?

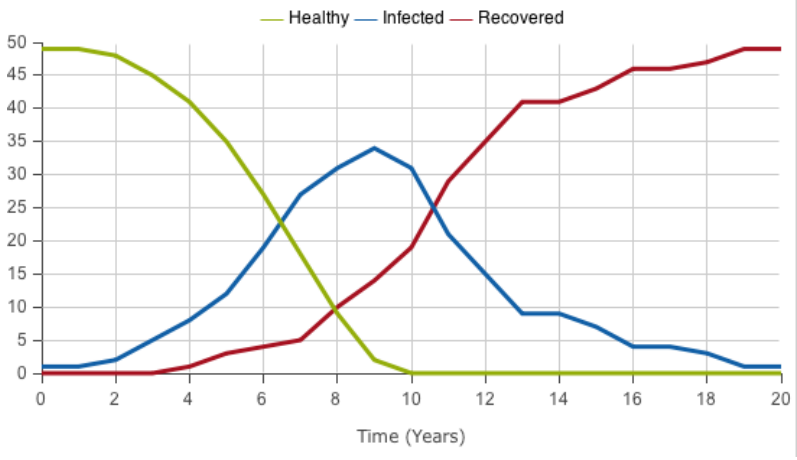
14. Well since our infection rate now depends on the number of people who are infected we have to have at least one person infected to get the epidemic going. Let's change the [**Healthy**] and [**Infected**] states so one person starts in the infected state at the beginning of the simulation.

15. Change the **Start Active** property of the primitive [**Healthy**] to `Index([Self]) <> 1`.

16. Change the **Start Active** property of the primitive [**Infected**] to `Index([Self]) == 1`.

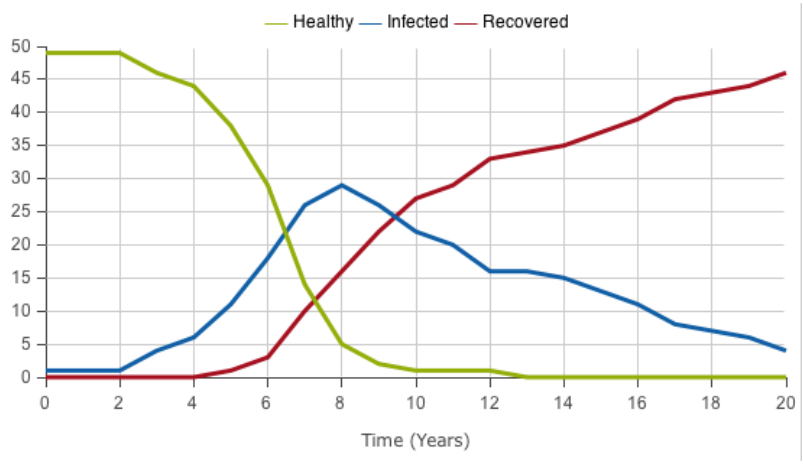
17. Each agent has an index starting with 1, we have set our initially active equations so the first agent in the population will start the simulation in the infected state. Let's run the model to see this working.

18. Run the model. Here are sample results:



19. Each time we run the model we will get a different set of results. Sometimes the infection will die off after the first infected person recovers. Many other times an epidemic spread of the disease will occur.

20. Run the model. Here are sample results:



Agent Geography

One of the key strengths of Agent Based Modeling is that it allows us to study the geographic relationship between our agents. So if we are developing a disease model we do not have to assume that all the agents are perfectly mixed together like atoms in a gas (such as we generally would in System Dynamics). Instead, using Agent Based Modeling we can explicitly define the physical relationship between the different agents and study how this geography affects the spread of the disease.

In general when we talk about geography we mean spatial geography: the locations of people within a region in terms of their latitude and longitude (and sometimes their elevation). Insight Maker supports this kind of geography, but it also supports a second kind of geography: network geography. Insight Maker allows the specification of "connections" between agents. This leads to a new type of geography where you have centrally located agents (ones connected to many other agents) and agents far from the network's center (those that are unconnected or just connected to a very few other agents).

Both these types of geographies can be useful in exploring important features of real-world systems. In the following sections, we will introduce their properties and show you how to utilize them in your own models.

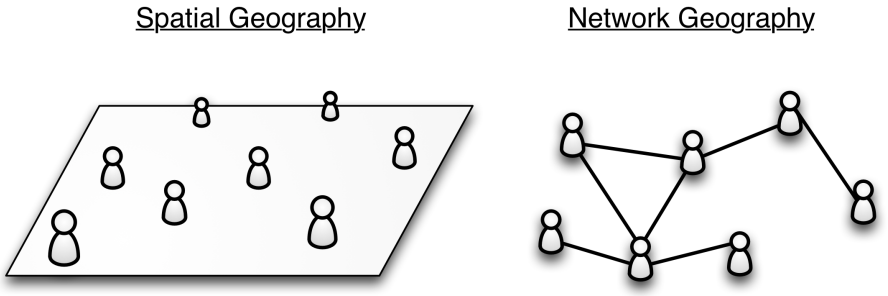


Figure 3. Spatial geography and network geography.

Spatial Geography

In Insight Maker, each Agent Population can be given dimensions in terms of a width and a height. By default, agents are placed at a random location within this region. You can, however, choose a different placement method for the starting position of the agents. The following placement methods are available:

Random : The default. Agents are placed at random positions within the geometry specified for the agent population.

Grid : Agents are aligned in a grid within the population. When using this placement method, you will need to ensure that you have enough agents so that the grid is complete. You might need to experiment with increasing or decreasing the number of agents to make the grid fit perfectly for a given set of region dimensions.

Ellipse : Agents are arranged in a single ellipse within the region. If the region geometry is a square, then the agents will be arranged in a circle.

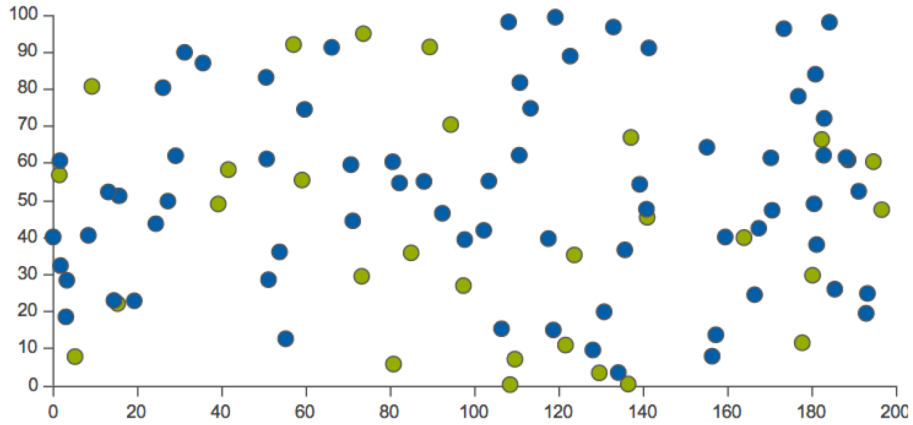
Network : Assuming network connections between agents have been specified, the agents will be arranged in an attempt to create a pleasing layout of the network structure.

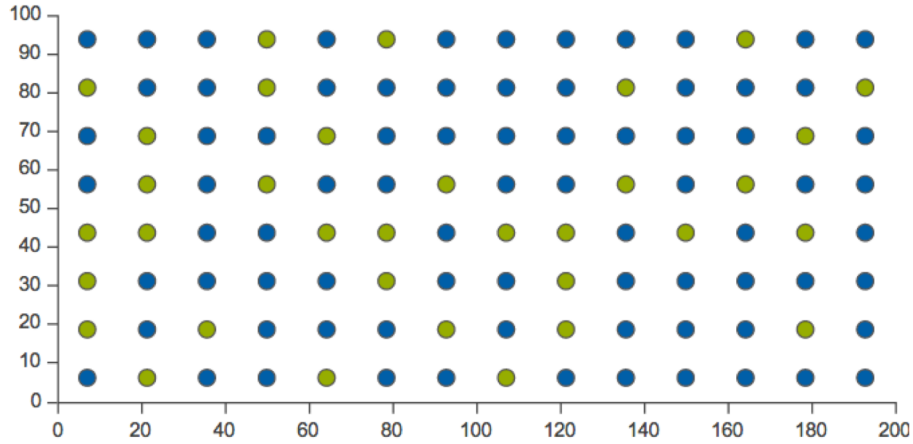
Custom Function : Here you can specify a custom function to control the layout of the agents. This function will be called once for each agent in the population and should return a two-element vector where the first element is the x-coordinate of the agent, and the second element is the y-coordinate. The primitive **[Self]** in this function will refer to the agent that is being positioned.

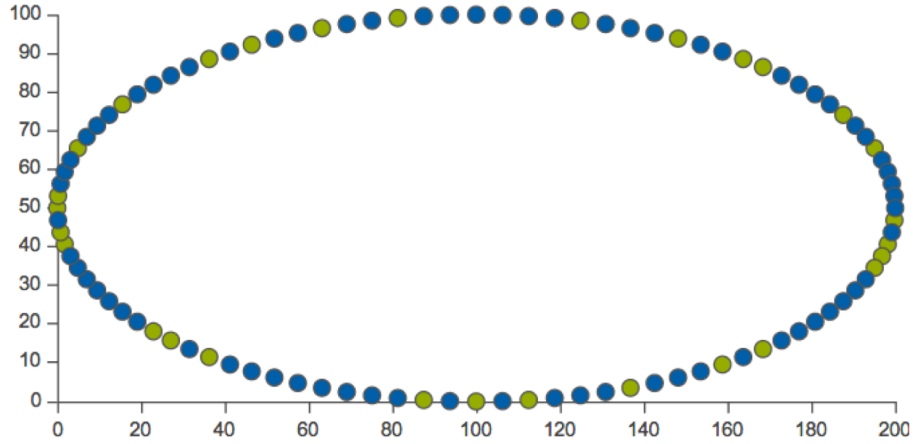
Spatial Find Functions

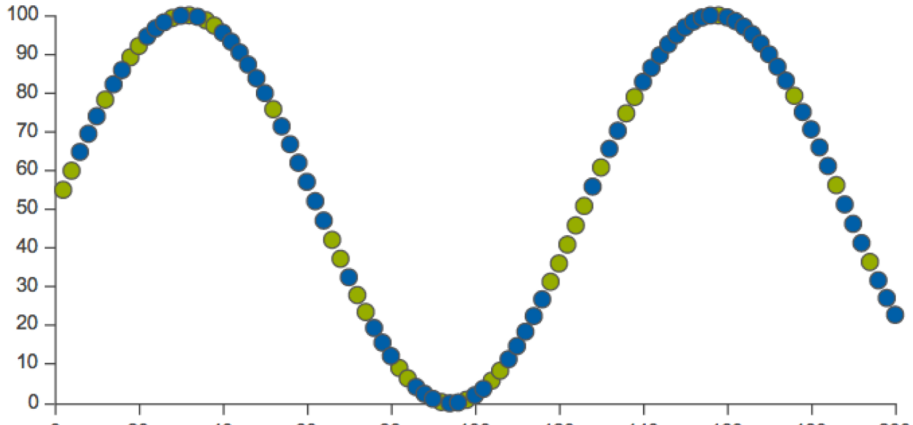
When working with a spatially explicit model, a number of additional find functions are available for you to obtain references to agents that match a given spatial criteria.

`FindNearby()` is a function that returns a vector of agents that are within a given proximity to a target agent. It takes three arguments: the agent


100
90
80
70
60
50
40
30
20
10
0
0
20
40
60
80
100
120
140
160
180
200


100
90
80
70
60
50
40
30
20
10
0
0
20
40
60
80
100
120
140
160
180
200


100
90
80
70
60
50
40
30
20
10
0
0
20
40
60
80
100
120
140
160
180
200


100
90
80
70
60
50
40
30
20
10
0

population primitive, the agent target for which you want nearby neighbors, and a distance. All agents within the specified distance to the target agent will be returned as a vector.

It is useful now to introduce a concept that will be very helpful to you. When used in an Agent, [**Self**] always refers to the agent itself. If you have a primitive within an agent, [**Self**] can be used from that primitive to get a reference to the agent containing the primitive. So the following equation in an agent will return a vector of agents that are within 15 miles of the agent itself:

```
FindNearby([Population], [Self], {15 Miles})
```

Two other useful functions for finding agents in spatial relation to each other are `FindNearest()` and `FindFurthest()`. FindNearest returns the nearest agent to the target while FindFurthest returns the agent that is farthest away from it. Each of them also supports an optional third argument determining how many nearby (or far away) agents to return (this optional argument defaults to one when omitted).

For example, the following equation finds the nearest agent to the current agent:

```
FindNearest([Population], [Self])
```

While this finds the three agents that are furthest from the current agent:

```
FindFurthest([Population], [Self], 3)
```

Movement Functions

You can also move agents to new locations during simulation. To do this, it is helpful to introduce a new primitive we have not yet discussed. This primitive is the *Action* primitive. Action primitives are designed to execute some action that changes the state of your model. For instance, they can be used to move agents or change the values of the primitives within an agent. An action is triggered in the same way a transition is triggered. Like a transition, there are three possible methods of triggering the action: timeout, probability, and condition.

For instance, we can use an action primitive in an agent and the `Move()` function to make agents move during the simulation. The Move function takes two arguments: the agent to be moved, and a vector containing the x- and y-distances to move the agent. Thus, we could place an action primitive in our agent and give it the following action property to make the agent move randomly over time[4]. The equation will move the agent a random distance between -0.5 and 0.5 units in the x-direction and a random distance between -0.5 and 0.5 units in the y-direction.

```
Move([Self], «rand, rand»-0.5)
```

[4]What we are implementing here is known as a "random walk" or Brownian motion. It is a commonly studied pattern of movement with wide applications in science.

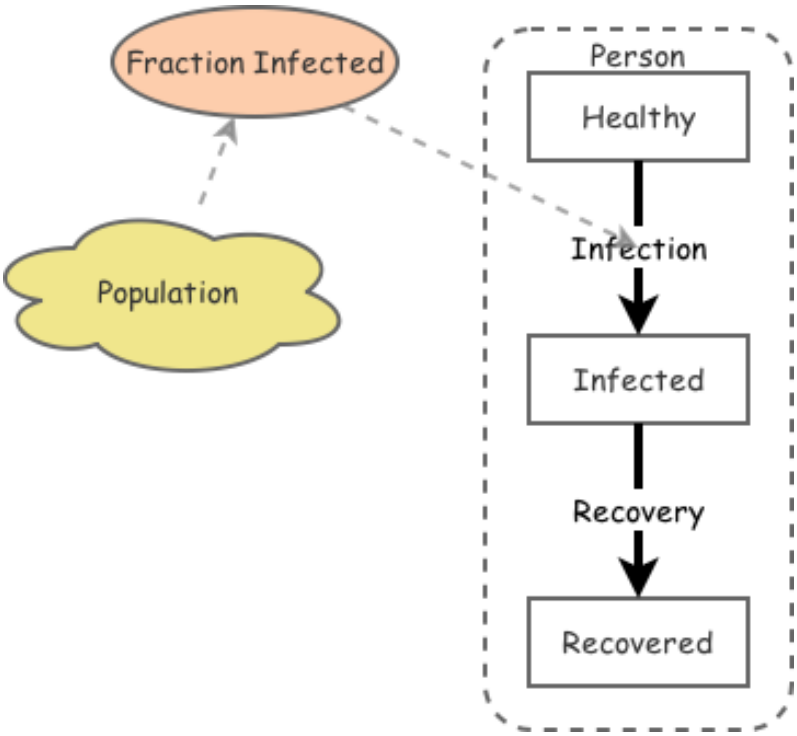
Another useful movement function is the `MoveTowards()` function. MoveTowards moves an agent towards (or away from) the location of another agent. MoveTowards takes three arguments: the agent to be moved, the target agent to move towards, and how far to move towards that agent (with negative values indicating movement away). The following command would move an agent one meter closer to its nearest neighbor in the population.

```
MoveTowards([Self], FindNearest([Population], [Self]), {1 Meter})
```

Agent Movement

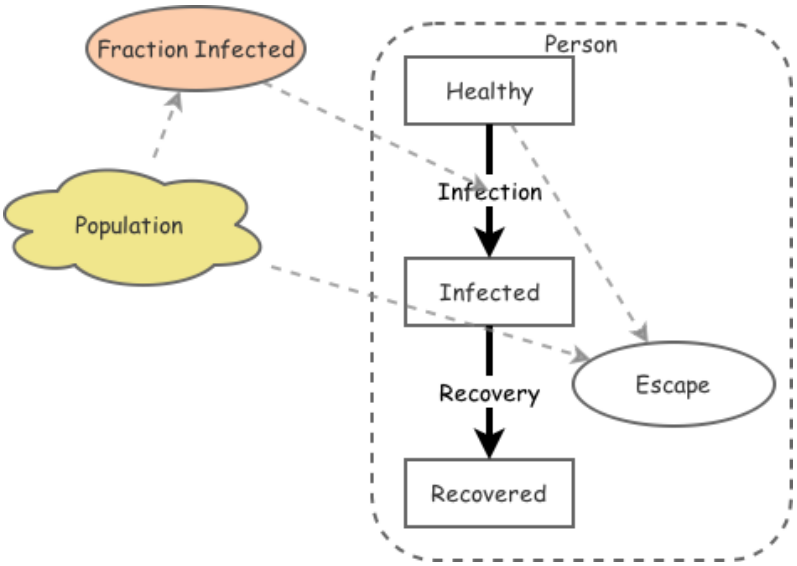
This model illustrates the use of movement within agent based models. We adapt the previous disease model so that healthy agents flee from the nearest infected agent.

1. The model diagram should now look something like this:

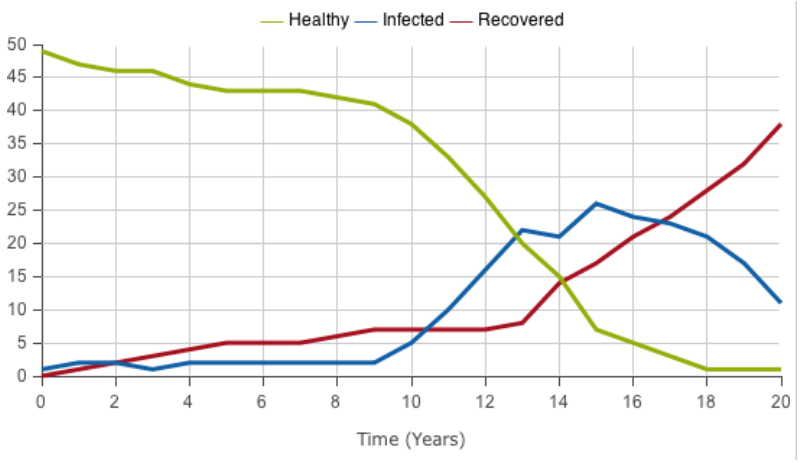


2. We will extend our disease model from earlier by adding movement to the agents. First we need to create an action primitive.
3. Create a new **Action** named [**Escape**].
4. Create a new **Link** going from the primitive [**Healthy**] to the primitive [**Escape**].

5. Create a new **Link** going from the primitive [**Population**] to the primitive [**Escape**].
6. The model diagram should now look something like this:

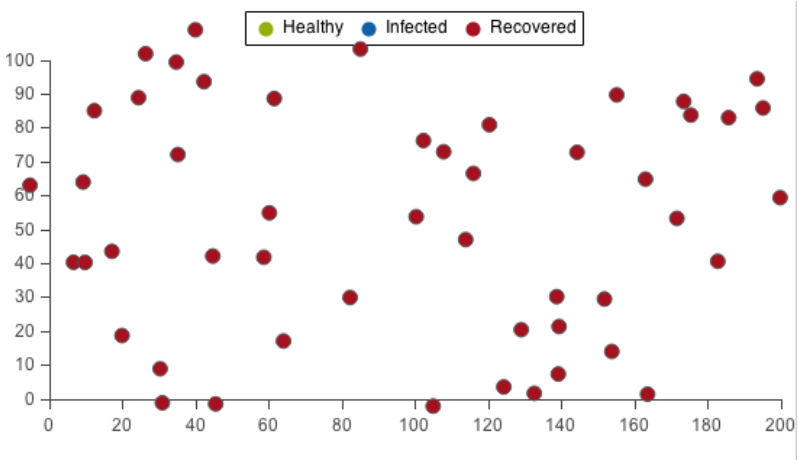


7. We will have this action be triggered when the agent is healthy and there is at least one infected agent in the simulation.
8. Change the **Value/Equation** property of the primitive [**Escape**] to `[Healthy] and Count(FindState([Population], [Infected])) > 0`.
9. The action will cause healthy agents to move away from the nearest infected agent. In effect, fleeing from sick individuals.
10. Change the **Action** property of the primitive [**Escape**] to `MoveTowards([Self], FindNearest(FindState([Population], [Infected])), [Self]), -2)`.
11. We can now run the simulation.
12. Run the model. Here are sample results:



13. These results are not too interesting as they do not show the locations of the agents. To see the agents moving, we need to change the display type to **Map** which gives us a visualization of the location of agents. Then we run the simulation again.

14. Run the model. Here are sample results:



Network Geography

To create connections and remove connections between agents you can use the `Connect()` and `Unconnect()` functions. Both of these functions take two arguments: the agents that should be connected or disconnected. For example, to connect an agent to its nearest neighbor, you could use the following

```
Connect([Self], FindNearest([Population], [Self]))
```

To disconnect an agent from its nearest neighbor (assuming they are connected), you would use:

```
Unconnect([Self], FindNearest([Population], [Self]))
```

To obtain a vector of connections to an agent, use the `Connected()` function:

```
Connected([Self])
```

Connections are not directed so creating a connection from agent A to agent B is the same as creating a connection from agent B to agent A. Also only one connection between a given pair of agents will exist at a time. So creating two connections between a given pair of agents will have the same effect as creating a single connection.

By default, no connections are created when a simulation is initially started. If you change the **Network Structure** configuration property of the agent

population primitive, you can specify a function to create connections when the simulation is started. This function is called once for each pair of agents in the model. The agents are available in the function as the variables *a* and *b*. If the function evaluates to `true`, then the agents will start connected. If the function evaluates to `false`, the agents will not be initially connected.

You could use this function to, for instance, specify that 40% of agents will be directly connected to each other at the start of the simulation. The following equation would do that by generating a random true/false value with 40% probability of returning `true` each time it is called:

```
RandBoolean(0.4)
```

Multiline Equations

So far in this book, the equations we have looked at have generally been straightforward mathematical formulae. We have introduced some more advanced concepts, such as vectors, but for the most part our equations have been relatively simple one-liners. When doing Agent Based Modeling however, at some point you will find these one-line equations to be limiting. When you begin to run into these limitations with your own models, you may need to start using multiline equations to achieve certain agent behavior.

Almost everyplace in Insight Maker where you can write a mathematical expression, you can also right a multiline equation. It turns out that Insight Maker's language for specifying equations is actually a complete computer programming language and you can exploit the strength of this programming language by writing your equations over several lines instead of using a single line mathematical formula.

We delayed introducing these capabilities until now, as they can sometimes be a distraction from focusing on understanding a system. However, when you build complex Agent Based Models, they can be necessary to express the model logic you wish. Given this need, we will provide a brief introduction to the programming features that can be used as part of Insight Maker equations. You do not need to delve deeply into these capabilities now, but be aware that they are available for when you need them in your own models.

Variables

Variables are temporary slots to store values to be reused within your equations. Variables are created using the '`<-`' symbol meaning assignment. For instance:

```
a <- 2 # The variable 'a' holds the value 2
b <- a + 2 # The variable 'b' holds the value 4
a <- b^2 # a=16, b=4
```

Variable names can contain any number of letters and numbers and must always start with a letter.

If-Then-Else

You should be familiar with the `IfThenElse()` function. A multiline alternative to it exists. The following is equivalent to `IfThenElse([Lake] > 10, 1, 2)`.

```
If [Lake] > 10 Then
   1
Else
   2
End If
```

One of the benefits of these multiline equations is that they can be more readable than the single line functions. This is especially true if you are trying to do nested *if* statements. Compare `IfThenElse([Lake] > 10, 1, IfThenElse([Lake] < 5, -1, 2))` to:

```
If [Lake] > 10 Then
   1
Else If [Lake] < 5 Then
   -1
Else
   2
End If
```

The second one is much more readable. This makes it easier to maintain and more resilient to potential typographical errors.

Loops

Loops are a programming construct that repeat some code multiple times. There are several different types of loops. One important loop is the *for* loop which repeats a command a specified number of times. Here is an example of it being used:

```
sum <- 0
For i From 1 To 3
   sum <- sum + i
End Loop
sum
```

The inner part of the loop is run three times here. The first time the variable i is assigned the value of 1, the next time 2, and the last time 3. So this sums up the values of 1, 2, and 3 resulting in 6.

Another variant of the *for* loop is the *for-in* loop. This uses a vector to assign the values of the iterations. The following code sums the numbers 1, 5, and 10 to get 16.

```
sum <- 0
For i In «1, 5, 10»
   sum <- sum + i
End Loop
sum
```

For-in loops can be very useful to iterate through a vector of agents. Another useful loop is called the *while* loop. It does not repeat a predefined number of times and instead repeats until a condition becomes true. Here is an example:

```
total <- 2
While total < 100
   total <- total^2
End Loop
total
```

This code keeps squaring the *total* variable until the total is greater than 100. In this case, this will result in 256.

Functions

Functions allow you to reuse code in multiple places in your model. For instance, imagine you had a model that dealt with temperatures in both Degrees Fahrenheit and Celsius. If you could not use the built in unit conversion functionality, every time you wanted to convert from one form to the other you would have to include the standard conversion formula in your equations. Not only would this be tedious, it would also be error prone as the more times you type an equation, the higher the chance of making a mistake.

You can define functions in two ways. One is a short one-liner:

```
FtoC(f) <- 5/9*(f+32)
```

And another is a multiline form allowing you to incorporate multiline logic in your functions:

```
Function FtoC(f)
```

```
   5/9*(f+32)
End Function
```

A great place to include your functions is in the *Macros* section of your model. You can enter macros by clicking the **Macros** button in the **Tools** section of the toolbar. The functions you define here will be accessible in any equation in any part of your model.

Integrating SD and ABM

System Dynamics modeling and Agent Based Modeling are two different ways of approaching a system. In general, System Dynamics looks at highly aggregated systems and encourages the study of feedback. Agent Based Modeling explores individuals and the interactions between these individuals.

Some software packages only do System Dynamics or Agent Based Modeling leading to the perception that they are somehow incompatible methodologies. Although these techniques can be thought of as quite different, it important to realize that, at the end of the day, both of them are simply applied mathematics. To emphasize this, Insight Maker integrates both these techniques together seamlessly in its modeling environment. There is no such thing as an "Insight Maker Agent Based Model" or an "Insight Maker System Dynamics Model". There are simply models where you may use agent-based techniques, System Dynamics techniques or a mixture of the two.

Insight Maker (and other modeling packages such as AnyLogic http://www.anylogic.com/) allows you to integrate the two seamlessly together. For instance, in this chapter we have used state transition diagrams within our agents. We could have just as well used stock and flow diagrams within the agents so that each agent in effect contained its own System Dynamics model of its state. Similarly if you have a large System Dynamics model you could create an agent-based sub-model that feeds into the main model dynamics.

When doing modeling, it is important to not get focused on labels or taxonomies of different techniques. Given a modeling task, you want to think about what tools and techniques are best used to approach it. You want to make sure not to approach a modeling task by trying to figure out how to force that task into the constraints of a favorite modeling paradigm.