An Introduction to RePast using ReLogo with Groovy

Jakub Kielbasa*

These notes are available free of charge, with the request that the reader reflect on and question all aspects of their life as they do their field of study, including religion, science, and any other indoctrination.

September 3, 2013

*I am currently a PhD student at the Australian National University. My contact details are jakub.kielbasa.au@gmail.com.
## Contents

1 Introduction to RePast
   1.1 Creating a New Project ........................................ 5
      1.1.1 Creating Agents ........................................... 6
      1.1.2 Structuring Code ......................................... 8
         1.1.2.1 Turtles .................................................. 8
         1.1.2.2 Observer ............................................... 8
         1.1.2.3 User Globals and Panel Factory ....................... 9
   1.2 Other Controls ................................................ 10
      1.2.1 World Dimensions ......................................... 10
      1.2.2 Toroidal vs Non-Toroidal ................................. 10
      1.2.3 Colors ..................................................... 11

2 Basic Methods ..................................................... 12
   2.1 Patches ......................................................... 12
      2.1.1 Adding and Instantiating Patch Variables .................. 12
      2.1.2 Accessing Patch Variables ................................ 14
      2.1.3 Importing Patch Data ..................................... 15
   2.2 Turtles .......................................................... 16
      2.2.1 Initialization .............................................. 17
      2.2.2 Movement .................................................... 17
      2.2.3 Other Behaviors .......................................... 18
      2.2.4 Observing Turtles .......................................... 19
         2.2.4.1 Patches Visited ....................................... 19
         2.2.4.2 Pen Down ............................................... 20
         2.2.4.3 Labels .................................................. 21
      2.2.5 Aggregate Statistics ...................................... 22
      2.2.6 Turtles’ Scope ............................................. 22
   2.3 Links ............................................................. 23
      2.3.1 Creating Links ............................................. 23
      2.3.2 Manipulating Links ....................................... 25
   2.4 Global and Local Variables .................................... 26
      2.4.1 Global Variables ......................................... 26
      2.4.2 Local Variables .......................................... 27
   2.5 Other Methods .................................................. 27
      2.5.1 Agent Sets ................................................ 27
         2.5.1.1 Creating, Adding, Combining, Removing and Clearing .... 27
         2.5.1.2 Applying a Function ................................ 28
      2.5.2 Lists and Maps ............................................ 28
   2.6 Extending Classes and Groups .................................. 29
      2.6.1 Extending Classes ....................................... 29
      2.6.2 Groups .................................................... 31
   2.7 Scheduling ...................................................... 32
      2.7.1 Time-Driven Events (Discrete Time) ....................... 32
      2.7.2 Event-Driven Time (“Continuous” Time) ................... 32
   2.8 Stochastic Elements ............................................ 34
Remark. This document, in its current form, is intended as an introduction to programming agent based models using RePast (specifically ReLogo and Groovy), rather than a philosophical discussion of this type of modelling. Two books which discuss the philosophy of agent based modelling are, for economics, the Handbook of Computational Economics Vol. 2, Eds. Tesfatsion and Judd (2006), and more generally, Agent-Based and Individual-Based Modelling, by Railsback and Grimm (2011).

Useful websites for this book include:

- [http://repast.sourceforge.net/docs/api/repast_simphony/ReLogoPrimitives.html](http://repast.sourceforge.net/docs/api/repast_simphony/ReLogoPrimitives.html) (all ReLogo primitives)
- [http://repast.sourceforge.net/docs/api/repast_simphony/repast/simphony/parameter/StringConverterFactory.html](http://repast.sourceforge.net/docs/api/repast_simphony/repast/simphony/parameter/StringConverterFactory.html) (for other converters, used in Chapter 4)
1 Introduction to RePast

1.1 Creating a New Project

Clicking on the “New Project” icon (Arrow 1 in Figure 1.1.1), we get a dialogue box open which asks for the Project Name, amongst other things. Type in whatever you wish to call your project (say Test01), and then click Finish.
1 Introduction to RePast

1.1 Creating Agents

To create a turtle-class called Walker01, make sure you are clicked on the relogo file in the package explorer (Arrow 1 in Figure 1.1.3), and then press on the turtles icon (Arrow 2 of Figure 1.1.1). A screen will come up (Figure 1.1.4), and in the “Name” box (Arrow 1), type the name of turtle class you wish to create (e.g. Walker01).
1 Introduction to RePast

After clicking “Finish”, a new groovy class, called Walker01, will be created in the Package Explorer. Double clicking on Walker01.groovy will open the area indicated by Arrow 7 of Figure 1.1.1, and this is where you program the commands of what you want the Walker01 class of mobile agents to do. The initial screen should have the following information:

```
01 package test01.relogo
02
03 import static repast.simphony.relogo.Utility.*;
04 import static repast.simphony.relogo.UtilityG.*;
05 import repast.simphony.relogo.BasePatch;
06 import repast.simphony.relogo.BaseTurtle;
07 import repast.simphony.relogo.Plural;
08 import repast.simphony.relogo.Stop;
09 import repast.simphony.relogo.Utility;
10 import repast.simphony.relogo.UtilityG;
11
12 class Walker01 extends BaseTurtle {
13
14 }
```

Line 1 tells the class Walker01 which package it “lives” in; the import statements (lines 3 - 10) tell what methods (i.e. functions) the Walker01 class has access to (these statements are usually contracted so you don’t see them), and the most important bit are the last three line, in that Walker01 extends (i.e. can use all methods from) the BaseTurtle class, which is a pre-programmed class with many useful methods. It is between the braces that all the code of how you want the Walker01 class to behave is written.
Similar processes apply to Patch, Link and Observer classes, but there are some differences, which we will discuss when we get there (these are easiest shown through examples).

1.1.2 Structuring Code

1.1.2.1 Turtles

The structure of the code defining the behavior of a mobile agent (i.e. turtle) is typically done as follows:

```java
01 class Walker01 extends BaseTurtle {
02     double Walker01_variable01 = 13
03     def Walker01_variable02 = 0.01
04
05     def method01(input_011,...,input_01n){
06         implementing method01
07     }
08
09     def method02(input_021,...,input_02m){
10         implementing method02
11     }
12
13     def step(){
14         implementing the order of how an agent
15         behaves, using the methods and constants
16         above
17     }
18 }
19 }
```

A turtle class is typically structured by defining turtle specific variables (lines 2 and 3), followed by some functions (lines 5 - 12) which can, but don’t have to, take inputs, and return some result. The last function is usually called `step` (line 14), and this implements the schedule (i.e. timing of events) of how you want this turtle type to behave, which uses the previous definitions and variables, as well as turtle-specific methods (e.g. to find some local characteristic).

1.1.2.2 Observer

The structure of the code describing how the program runs is written in the observer class. Think of this class as one that is the puppet-master of the other classes; that is, it tells which classes to act (including which of their methods to invoke), in what order, at each time step.
In the UserObserver class, two functions always exist, conventionally named setup and go. The setup function creates instances of certain types of agents (called instantiation), and also specifies their initial conditions. The go function implements in what order you want agents to behave (e.g. consumer agents bid first, and seller agents then satisfy the agents in the order they came to the shop), and anything else you want to happen (e.g. stop the simulation after 1000 steps).

1.1.2.3 User Globals and Panel Factory

This is where you make “buttons” and “sliders” to run certain parts of the code, once in the visualization environment.

To add a button that implements a function with the name string, labelling it string2, the method addButtonWL(string, string2) is used. That is, string could be something like “go”, which corresponds to the go function in the UserObserver class, and the label could be “Go Once”, which is nothing more than a label.

The addSliderWL(string, string2, min, inc, max, init) is a function that defines the variable string, labels the slider string2, has minimum value min, maximum value max, increments within this range by inc, and is initially set at init.
1.2 Other Controls

1.2.1 World Dimensions

To change the dimensions of the world, go to your_project.rs file in Package Explorer, then open the parameters.xml file. Using the “expand” arrows, expand each of the cells with name “parameters” (see Arrow 1 of Figure 1.2.1). Then each parameter will have a name (Arrow 2) and its default value (Arrow 3). To change the default value, double click on it and enter the appropriate dimension.\(^\text{1}\) The world dimension must contain the point \((0,0)\), so the smallest \(\maxPxcor\) value must be an integer no smaller than 0, and the largest \(\minPxcor\) is no larger than 0 (similarly for the \(y\)-axis).

![Figure 1.2.1: Changing world dimensions via the parameters.xml file.](image)

This method allows you to change the world dimensions, whilst leaving them as parameters. In the following section, you will find the following line in SimBuilder.groovy:

```groovy
RLWorldDimensions rLWorldDimensions = new RLWorldDimensions(minPxcor,
maxPxcor, minPycor, maxPycor);
```

which if you want a world that you will not change size, then you can directly change the numbers here, as appropriate.

1.2.2 Toroidal vs Non-Toroidal

In Package Explorer, in the scr folder, open your_project.context then the open SimBuilder.groovy file. In the code, about half way down, you should have the code:

```groovy
RLWorldDimensions rLWorldDimensions = new RLWorldDimensions(minPxcor,
maxPxcor, minPycor, maxPycor);
```

\(^\text{1}\)Sometimes, you can double click and enter a number but it seems to revert back to the original. Don’t worry, it hasn’t, just save the file now!
which is the code for a toroidal environment. If an environment is needed that is non-toroidal (i.e. the boundaries are fixed and nothing exists beyond them), then the above code needs to be changed to:

```java
RLWorldDimensions rLWorldDimensions = new RLWorldDimensions(minPxcor, maxPxcor, minPycor, maxPycor, new repast.simphony.space.continuous.BouncyBorders());
```

### 1.2.3 Colors

In RePast, colors are defined by a number in the set \([0, 140]\), with the colors being depicted in Figure 1.2.2. Every number where the last two digits are 5.0 is a “base” color, with any deviation up towards the last two digits being 9.9 is a lightening of the shade (with 9.9 being white), and any deviation of the last two digits towards 0.0 darkens the shade (with 0.0 being black).

![Figure 1.2.2: The colors available in Relogo, with an example of the shades.](image)

Many primitives (inbuilt functions) require a color, such as `setPcolor(·)`, `setColor(·)`, where inside the parentheses is either a number (e.g. 105 for blue), or the corresponding color primitive (e.g. `blue()` for blue).
2 Basic Methods

This chapter builds on from Chapter 1, especially the structuring of the code. ReLogo has many inbuilt functions called primitive methods, or just primitives, ready to use. These are very useful as they reduce the amount of code required, and their performance is already optimized.

Example. An example that will be used throughout this text as an example is what will be referred to as the Mining Example. We want Miner turtles to go in search of high ore deposits.

2.1 Patches

These are the non-mobile agents that mobile agents (turtles) move over.

2.1.1 Adding and Instantiating Patch Variables

To add a variable that each patch possesses, in the class UserPatch, add the following code:

```java
class UserPatch extends BasePatch {
    def patch_var01 = 0.1
    double patch_var02 = 500
    @Diffusible def patch_var03
}
```

The easiest way to create patch variables is shown in lines 2 and 3. This just uses def to define something which follows (it could be anything), or double to create a double precision number (which clearly can only be a double precision number, and not, for example, a list).

There is also a fancy method of creating patch variable, which is convenient in certain circumstance. The @Diffusible tells the patch class that the variables following (i.e. patch_var03) are variables with certain “diffusible” characteristics. That is, these variables can be used to “spread” or “diffuse” themselves in their neighborhood, amongst other things.

@Diffusible specific methods include:

- diffuse(string, double) which diffuses the variable string (e.g. a string could be “toxin”) into the Moore neighborhood by a number double∈ [0, 1] (i.e. a fraction),
- diffuse4(string, double) which is the same as above, but the four adjacent neighbors,
- diffusibleAdd(string, number) which adds number to the variable defined by string, for all patches,
- diffusibleSubtract(string, number) which subtracts number from the variable defined by string, for all patches,

^You can find them at http://repast.sourceforge.net/docs/api/repast_simphony/ReLogoPrimitives.html but I will explain a useful subset of these in the rest of this document.
• **diffusibleMultiply**(string, number) which multiplies the variable string by number, for all patches,

• **diffusibleDivide**(string, number) which divides the current value of the variable defined by string by number, for all patches,

• **diffusibleApply**(string, DoubleFunction) which applies the DoubleFunction to the variable defined by string, for all patches.

To change the patch variable, in **UserObserver**, to the setup function, the following is added:

```plaintext
01 ask(patches()){
02     patch_var01 = some function of some inputs
03 }
```

where line 1 is the notation to tell all patches, one at a time, to execute what follows in braces (i.e. line 2), where patches() is a patch primitive returning a collection with all patches. Essentially, the ask(·) is a loop over all elements in the parentheses. Line 2 says that for each patch, set the patch variable patch_var01 to some function you require.

In the **Mining Example**, we want there to be three random points chosen on the world, and from each of these points, within a certain radius, have ore decrease from the center patch, subject to some stochasticity. To implement this on the patches, the following code would be added to UserPatch:

```plaintext
01 double resource = 0
```

This defines the patch variable resource, and initializes it to zero. Then in setup in UserObserver, the following code is used:

```plaintext
01 def P = nOf(numCenters, patches())
02 ask (P){
03     resource = 10
04     def this_center_patch = it
05     def patch_AS01 = inRadius(patches(),5)
06     patch_AS01.add(it)
07     ask (patch_AS01){
08         resource = 10 - distance(this_center_patch) + Math.random()*2
09         setColor(scaleColor(105,resource,3,12))
10     }
11 }
```

where line 1 selects numCenters (a global variable, defined in Section 2.4.1) patches at random. Then line 2 asks each of these center patches to set resource to 10 (line 3), then line 4 stores the current center patch as the local variable this_center_patch, line 5 creates a new agent set of patches within a radius of 5 (by using the primitive inRadius(·,·) where the first argument is an agent set, and the second argument is the radius). The problem is that the agent set patch_AS01 does not include the center patch, so line 6 adds it (i.e. the center patch in question). Line 7 asks all patches in the agent set patch_AS01 to set their resource to a certain function (line 8), in this case, decreasing resource from the center patch, but with some randomness. The primitive distance(·) is used, which calculates the distance from the current agent (one of the patches in patch_AS01) to the agent in the parentheses (i.e. this_center_patch).

---

2 See Section 2.5.1 for more on agent sets, but the intuitive notion of it being simply a set of agents is sufficient.
Line 9 uses `setPcolor(·)`, a patch primitive which sets the current patch to whatever color is in brackets. Another primitive is `scaleColor` which takes four inputs: the first is the color that will be scaled (105 corresponds to blue), the second is the variable that will be used to define the percentage shade of color 105 the patch will be set to, and the last two numbers determine the scale on which that percentage will be calculated. For example, say the `resource` of a patch is 6.3, then the shade of color 105 will be \((6.3 - 3)/(12 - 3) \approx 0.37\) (i.e. approximately 37% of the “full color”). This means that the larger the value of `resource`, the lighter the shades. This is not very intuitive, but it can easily be reversed (i.e. the larger `resource`, the darker the shade), the last two inputs in `scaleColor` simply need to be reversed so `scaleColor(105, resource, 12, 3)` would be the code.

With `numCenters = 3`, you should get something similar to the following figure.

![Figure 2.1.1: The visual output of patches, when numCenters = 3 in the Mining Example.](image)

**Exercise.** There is a small problem with setting patches in this way: if the random patches are close to one another, you get overlap, and then you might not get the radii of resource densities as wanted. To see this, set `numCenters` at around 10. Using `Math.max(·)`, try fix this.

### 2.1.2 Accessing Patch Variables

Suppose we are creating the `step` function for `Walker01` turtles, and they need to access a patch variable (say `patch_var01`) of an arbitrary patch \(P\) (appropriately defined). This is done by using the dot notation of object oriented programming; that is, we would use \(P.patch\_var01\) to access this variable. For example, we can use `patchHere().patch_var01` to obtain the value of `patch_var01` on the patch the turtle is currently on.\(^3\)

In the *Mining Example*, in the step function of the Miner class, we might want a Miner to sense the resource levels of the neighboring patches, and move to a random patch in this set which has at least as high a resource as the current patch. In the `step` function in `Miner.groovy`, we would have code similar to:

\(^3\)In RePast, a turtle has access to all variables of the patch they are currently residing on, so they can just use `patch_var01` to access the value of `patch_var01` of the patch it is currently on. However, for ease of debugging and readability purposes, I would recommend using the dot notation.
2 Basic Methods

01 def N = neighbors()
02 def P = patchHere()
03 N.add(P)
04 ask (N){
05 if (resource < P.resource){N.remove(it)}
06 }

Just a reminder, this is code in the \textit{step} function of the Miner class. Line 1 defines a local variable \( N \) which is an agent set of all neighbors in the Moore neighborhood (using the primitive \textit{neighbors()}\), line 2 defines a local variable \( P \) which is the current patch the turtle is on, line 3 adds the current patch the turtle is on to the agent set \( N \). Then Lines 4 and 5 asks all 9 patches in \( N \), one at a time, to be removed from \( N \) if their resource is less than that of the patch the turtle is currently on. Now, you must be careful, as line 5 is in a patch context, so we can use \textit{resource} as it refers to the resource of whichever patch we referring to (as line 4 has us iterating over the patches in \( N \)). In the same line, the \( P\.resource \) notation is required as we are referring to a different patch (i.e. \( P \)), and that is why we needed to define \( P \) as a local variable.

2.1.3 Importing Patch Data

Suppose the world dimensions are 100 \times 100 and we have a text file containing some patch data (e.g. income) in the following form:

0 0 50.6
0 1 55.1
: : :
0 99 73.0
1 0 42.4
1 1 44.2
: : :
99 99 23.1

where the first column is the \( x \)-coordinate, the second is the \( y \)-coordinate, and the last is the income (we could have additional variables in additional columns). Now, we can use ReLogo primitives to set each \((x, y)\) pair to the appropriate income value. This is done as follows:

01 fileOpen("Income_data.txt")
02 def r = 0
03 def W = worldHeight() * worldWidth()
04 while ( r++ <= W ){
05 def x = fileRead()
06 def y = fileRead()
07 def I = fileRead()
08 patch(x,y).income = I
09 }
10 fileClose()

where line 1 opens the text file of interest, line 2 defines a number \( r \) which will be used later to stop a loop, line 3 finds the number of patches and stores it as the variable \( W \). Then, line 4 encodes the \textit{while} loop that, while we have read less (or equal) lines than the
number of patches, to proceed; the \( r++ \) notation means that each time the loop is run, \( r \) is incremented by 1. Lines 5 to 7 use the primitive `fileRead()` which reads the next part of the file, where the “parts” are separated by a whitespace (i.e., a space in the text file). Then line 8 actually sets the patch variable, and line 10 closes the file.

Exercise. Implement the above code. What do you notice about how long it takes to run setup? To see this, between lines 4 and 5, insert `print r`, and then look on your console to see which patch it is on (there are 10,000 patches).

Clearly, this takes too long to be useful (especially when first debugging, where you have to run the code many times). A much cleaner alternative is to use the Groovy language.\(^4\) When you open the text file of interest, what we want to do is to have the program read each line at a time, using the first string (once converted to an integer) as the \( x \)-coordinate, the second string as the \( y \)-coordinate, and the third string as the variable of interest (e.g. income).

The code in setup is:

```groovy
01 String fileContents = new File("C:/Jakub’s C Drive Documents/ReLogo Projects/Examples/Income Distribution/IncomeData.txt").text
02 fileContents.eachLine(){ line ->
03    def A = line.split()
05 }
```

where line 1 defines a new variable called `fileContents` as a string, but more specifically as `.txt` which is a text file, with the new declaring that it is a new text file. Clearly, the string inside the parentheses is the location of the file I want to convert to a text file (luckily, it is already a text file).

Line 2 defines our newly created file `fileContents` be iterated line by line (which is the `.eachLine()` statement). Then for each line, we want something done to it, so we use `->` to define that we want each line to perform a task in the code following. Line 3 defines a new variable \( A \), an array, which splits each line according to some delimiter (in our case, as the delimiter is by default whitespace, we do not need to declare anything). If the data in our file was split according to comma separated values (.csv file types), we would declare the delimiter to be a comma by having “,” in the parentheses.

Now, each element of \( A \) is a string, which cannot be used to identify a patch (as patches are identified by integers). Thus in line 4 we need to convert the first two elements of \( A \) to integers (this is done by using `.toInteger()`), and then the last number in \( A \), income, is also a string, but we do not want an integer, but rather want a double precision number, so we use `.toDouble()` instead.

Running this code will give the same output as using the ReLogo primitives above, but does it in literally a blink of an eye. Further, the amount of code required is much less, which is very useful in debugging a program!

### 2.2 Turtles

These are the mobile agents.

2.2.1 Initialization

To create turtles, we use the primitive createTurtles(·) where instead of Turtle in the name of the primitive, you use whatever name you gave to the class of turtles you wish to create (notice that it must be capitalized, and s must also be added to the end). The input to this method is an integer, which tells how many such turtles to create. In the setup function, if you just type createWalker01s(5), all 5 Walker01 turtles will be created, all on the patch (0,0).

To initialize turtles on the center of random patches, we use the primitive setxy(·,·). That is, we would have the code:

```plaintext
01 createWalker01s(numWalker01s){
02    setxy(randomPxcor(),randomPycor())
03 }
```

in the setup function, where randomPxcor() returns the x-coordinate of a random patch, and similarly for randomPycor(). If you wish for the turtles to be initialized to random locations (rather than the center of a random patch), then instead of randomPxcor(), we would use randomXcor(), and similarly for the y-coordinate.

Sometimes we want to put turtles on the center of random patches, but in such a way as to not have more than one turtle on each patch. This can be obtained by using the following code when creating the turtles:

```plaintext
01 createWalker01s(numWalker01s){
02    while (count(other(Walker01sOn(patchHere()))) > 0){
03       setxy(randomPxcor(),randomPycor())
04    }
05 }
```

where line 2 has a while loop, which says that while the condition count(other(Walker01sOn(patchHere()))) > 0 is true, then execute line 3. This condition counts (using count(·)) the number of other (using other(·)) Walker01 turtles on (using Walker01sOn(·)) the patch the turtle was instantiated to. If this number is larger than 0, then the Walker01 turtle has its location set randomly, and then the condition is checked again. This clearly requires the number of turtles be less (and usually quite a lot less) than the number of patches.

Exercise. Can you think of a cleaner way to achieve the same result?

2.2.2 Movement

Not all models will require movement, but those that do, require the turtles to have their movement defined. This is written in the step function. Recall that the step function implements what you wish the turtles to do, and in what order. To ask a turtle to move, primitives such as moveTo(·), forward(·), uphill(·) etc. are used.

In the Mining Example, suppose we want Miner turtles to move, conditional on their current location. That is, if resource of the patch the turtle is currently on, is zero, then alter the current heading by a random degree between $-30^\circ$ and $+30^\circ$, and move forward a length of 2 units; alternatively, if the resource of the current patch is positive, then of the eight neighboring patches and the current patch, move to a patch that has resource no less than the current patch. This is implemented by the following code:
2 Basic Methods

```groovy
01 def step(){
02   if (patchHere().resource == 0){
03     setHeading(getHeading() + (Math.random()*60 - 30))
04     forward(2)
05   }
06   else {
07     def N = neighbors()
08     N.add(patchHere())
09     def P = patchHere()
10     ask (N){
11       if (resource >= P.resource){}
12       else {N.remove(it)}
13     }
14     moveTo(oneOf(N))
15   }
16 }
```

where line 2 conditions execution of lines 3 and 4 on the resource of the patch the Miner turtle is currently on being zero. Then line 3 sets the heading (using `setHeading(·)`) to the current heading (obtained using `getHeading(·)`) plus a random number in the interval [−30, 30]. Then line 4 tells the turtle to move forward 2 units.

Line 6 has the alternative to line 2. That is, if resource of the current patch is non-zero (it has been constructed to be positive), then lines 7 to 13 was described on p.13, which essentially just obtains the set of neighboring patches (including the current patch) which have resources at least as large as the current patch. Then line 14 tells the agents to move to (i.e. `moveTo(·)` a random patch (i.e. `oneOf(·)`) from the set of neighboring patches at least as good as the current patch (i.e. the set N).

Suppose we wanted the turtle to move to the highest neighboring patch (including the current patch). This can be done using by changing line lines 6 to 14 above, to:

```groovy
01 def N = neighbors()
02 N.add(patchHere())
03 def P = patchHere()
04 moveTo(maxOneOf(N,{it.resource}))
```

where the movement (line 4) is to the patch with the maximum resource on it (using `maxOneOf(·,·)` where the first argument is the agent set, and the second is the “closure” or condition over which the maximum will be evaluated). But as this is a very common requirement, we can just use the primitive `uphill(“resource”)` which automatically finds the patch of the Moore neighborhood and the current patch which has the highest resource, and then moves to it.

2.2.3 Other Behaviors

We don’t just want a turtle to move, but we also want them to possibly perform some task, react to its environment. This would also be encoded in the `step` function.

In the Mining Example, we may want a turtle to find a local maximum (this was already done in the previous section), and extract 50% of the resource, and add it to its stockpile (which for now, it carries). Clearly, we also require the patch to reduce by that same amount of resource which was extracted. Firstly, we need to define a new Miner variable using `def stockpile = 0` which is the amount of resource the Miner has stockpiled. Then a new function called `mineResource` is also written in Miner.groovy:
2 Basic Methods

```ruby
01  def mineResource(){
02      stockpile += 0.5*patchHere().resource
03      patchHere().resource = 0.5*patchHere().resource
04  }
```

where in line 2, the stockpile has added to it (using the += notation) half of the resource of the patch the Miner is currently on. Line 3 reduces the resources on that patch by the amount that was extracted. Next, the code for the step function needs to be adjusted; specifically, the else part:

```ruby
01  else {
02      def N = neighbors()
03      N.add(patchHere())
04      def P = patchHere()
05      ask (N){
06          if (resource < P.resource){N.remove(it)}
07      }
08      moveTo(oneOf(N))
09      if (patchHere().resource >= maxOneOf(neighbors(),{it.resource}).resource){
10         mineResource()
11      }
12  }
```

where the added lines 9 and 10 say that if no patch in the 9 patch neighborhood has more resource than the current patch (line 9), then mine it (line 10). Lastly, to visualise the patches decreasing in resource, in go in UserObserver, we add:

```ruby
01  ask (patches()){
02      setPcolor(scaleColor(105,resource,0,12))
03  }
```

which updates the colors of the patches (but notice there is a zero as the minimum resource, as the resources are being mined and have a lower bound of zero).

Running the model as is will have the Miner turtles extract resource, and the color of the patches with resources become darker and darker until it can no longer be seen (but these patches will still have resources, as resources are reduced by a relative rather than absolute amount when it is mined).

Remark. When you want a turtle to do behave in a certain way, you will usually need to add/adjust code in multiple contexts.

2.2.4 Observing Turtles

When first creating a model, it is recommended that you observe a small number of turtles, and some of their properties. This can be done in a number of ways.

2.2.4.1 Patches Visited

You can record which patches have been visited by turtles by creating a binary patch variable. In UserPatch, define a new patch variable as `def usedQ = false`. Then when creating agents in setup, use:
2 Basic Methods

createMiners(10){
    setxy(randomXcor(),randomYcor())
    usedQ = true
}

where the added line 3 sets the patch variable usedQ to true for the patch the turtles were created on. Then in go, we use:

ask (patches(){
    setPcolor(scaleColor(105,resource,0,12))
    if (usedQ == true){
        setPcolor(45)
    }
}

where lines 3 and 4 set the color of any used patches yellow (45).

If you run this code and pause the output at around 50 ticks, you should get something like the following figure.

Figure 2.2.1: The visual output of used patches (yellow) when numCenters = 3, paused at 50 ticks, in the Mining Example.

If you run it for longer, you will see that the majority of patches visited are those that have resources, and only a few which do not (when the Miner turtles were first searching for patches with resources).

2.2.4.2 Pen Down

A much simpler way to follow turtles is to use the primitives penDown(), penUp() and penErase(). To see what happens, in the Mining Example, create only two Miner turtles, and when creating them, on a new line after setxy(randomXcor(),randomYcor()), add penDown(), then run the program. After about 50 ticks, you should see something similar to Figure 2.2.2.
2 Basic Methods

Figure 2.2.2: The visual output of tracking turtles using \texttt{penDown()} when \texttt{numCenters = 3}, paused at 50 ticks, in the Mining Example.

2.2.4.3 Labels

You can add labels to be seen in the visual interface by using the primitives such as \texttt{setLabel(\cdot)} and \texttt{setPlabel(\cdot)}. Before showing how to do this, you must adjust some code “under the hood” of RePast, as it has a bug in updating of labels. This is done as follows. Exit RePast, and in Windows Explorer, go to wherever you have RePast-Symphony installed. Then go to eclipse/pulgins/saf.core.ui_2.0.1, and you will find a Jar file by the name of saf.core.v3d; this is where the bug is. Rename this file saf.core.v3d(OLD). From the website http://repast.10935.n7.nabble.com/label-in-StyleOGL2D-td9431.html, about half way down, download the corrected file, and place it in this location. Now, we will have the labels doing what we want.

Suppose you wish to observe some characteristic a turtles (e.g. the direction turtles are heading towards, the income of turtles) in real time. This is achieved by adding a label to each turtle. When first creating the set of turtles, add the following:

```java
01 def setup(){
02     :
03     createWalker01s(numWalkers){
04         :
05         setLabel(precision(getHeading(),0))
06         :
07     }
08 }
```

The added line being line 5. Here, we use \texttt{setLabel(\cdot)} to set a label on all \texttt{Walker01} turtles, and then inside the parentheses, we add the variable of interest; here it is the heading. We could just write \texttt{setLabel(getHeading())}, but then the heading will be to approximately 10 decimal places. We can use the utility primitive \texttt{precision(\cdot,\cdot)} where the first argument is the big decimal, and the second argument is the number of decimal places you wish to reduce it to. So we get a heading that has no decimal places (i.e. just the degrees as integers between 0 and 359).

Now, if we were to run this program, you would see the label get instantiated by the setup
function, but then it would not change, even if the turtles did change their direction. This is because the label is not being updated. To update the label at each step, in the go function, we would have the following code:

```python
01 def go(){
02   :
03     ask (Walker01s()){  
04       :
05       setLabel(precision(getHeading(),0))  
06       :
07     }
08 }
```

which updates the label each time the go function is called (i.e. each time step). It might just be easier to ignore adding the label in the setup function, and just add it to the go function if you judge it’s fine to ignore the direction when first instantiated (e.g. if the instantiated direction is irrelevant).

It is also possible to add multiple variables to a label by encapsulating those variables in a list. That is, suppose we want the heading, x-coordinate and y-coordinate in a label. Line 5 above would simply be changed to `setLabel([precision(getHeading(),0), getXcor(), getYcor()])`. Again, we may want to use precision on `getXcor()` and `getYcor()` as they are big decimals.

Labels can also be added to patches, but with obvious modifications; you ask patches rather than turtles to show a variable (e.g. amount of a resource remaining, number of turtles on it).

### 2.2.5 Aggregate Statistics

In many cases, we need a summary value of all turtles of a particular type with a certain characteristic. For a Walker01 class, suppose we want the mean value of a variable `w_var01`, and set a global variable (say called `mean_w_var01`) to this value. In the go function, we would add the following:

```python
01   mean_w_var01 = mean(walker01s().w_var01)
```

In the Mining Example, after defining a global variable called `meanSP`, then in the go function, including the code `meanSP = mean(miners().stockpile)` will update the mean stockpile across Walker01s.

There are other common summary statistics and functions, but if you want to define your own function, then see Section 2.5.2 on lists.

### 2.2.6 Turtles' Scope

Scope refers to the variables accessible by a given part of the code. When in reference to turtles, scope is interpreted as the variables a given turtle has access to (e.g. the patch variables the turtle is currently on, possibly it’s neighbors, etc.). Suppose we want a turtle to access the patch variable `patch_variable01` of all patches in the Moore neighborhood. The code would be:

---

5 Again, at http://repast.sourceforge.net/docs/api/relogo/ReLogoPrimitives.html
when we are in a turtle context (e.g. in \texttt{ask (turtles())})\. We can also have turtles have access to the variable for all patches within a radius of the turtle (rather than the center of the patch it is on). This would be given by the code:

\begin{verbatim}
01 inRadius(patches(), 4)
\end{verbatim}

again, when in a turtle context. This returns all patches within a radius of 4 of the turtle. Care needs to be taken, as a patch is only returned if its centre is within 4, not if only part of the patch is within 4 units. Further, we can replace \texttt{patches()} by any agent set, so it could be \texttt{Walker01s()}, which is the agent set of all Walker01 turtles, or an agent set you previously defined (e.g. all consumer agents with income over 100).

We have already seen the use of these primitives, but now you should see what we were doing more holistically.

To set a variable of a set of patches (or other set of agents) that is associated with a particular turtle, we would use code similar to the following:

\begin{verbatim}
01 ask (Walker01s()){ \\
02 def list01 = patchHere().neighbors() \\
03 ask (list01){ \\
04 patch_variable01 = 0.9 * patch_variable01 \\
05 setPcolor(scaleColor(105,patch_variable01,0,1)) \\
06 } \\
07 }
\end{verbatim}

where line 1 is needed as we need to be in a turtle context (Walker01s in this case), line 2 defines a new list (of patches in this case), which I have called \texttt{list01}, which is iterated over in line 3, and line 4 tells these patches to reduce \texttt{patch_variable01} by 10\%, and lastly line 5 rescales the color (so it is seen visually on the output \textemdash it won’t be otherwise).

\section{2.3 Links}

The links class (Arrow 3 of Figure 1.1.1) is a class which provides a way to connect different agents together. For example, turtles can be connected via a social network which can be represented by links. There can also be different types of connections, such as a set of links which describes family, a different set to describe friends, a different set to describe work colleagues. Notice that here, a link between two agents can be in more than one set (e.g. a sister can be a family member as well as a friend).

\subsection{2.3.1 Creating Links}

To create \texttt{numLinks} (a global variable) random links with other turtles, when turtles are first created, we use the following code:

\begin{verbatim}
01 createWalker01s(numWalker01s){ \\
02 \\
03 createLinksTo(nOf(numLinks,other(walker01s()))) \\
04 \\
05 }
\end{verbatim}
where for each newly created Walker01, line 3 creates links (using the primitive `createLinksTo(·)` to numLinks random (i.e. `nOf(·, ·)`), with the first argument an integer, and the second an agent set) other Walker01 agents. We can also create links from (by using `createLinksFrom(·)` which creates an incoming directed link, or `createLinksWith(·)` which creates an undirected link. We can also create an agent set of turtles of undirected links (using `linkNeighbors()`, in links (using `inLinkNeighbors()` or out links (using `outLinkNeighbors()`) for a particular turtle, whichever is appropriate for the task.

Suppose we are in a turtle context, and have created an agent set of turtles (say AS1) we wish to be linked with, in an undirected way. This would be written as `createLinksWith(AS1)`. 

In the *Mining Example*, in setup, adding:

```java
01 ask (miner(1)){
02 def AS1 = other(miners())
03 createLinksWith(AS1)
04 }
```

with 20 Miners should result in something similar to the following figure:

![Figure 2.3.1: The visual output of links created from Miner 1 to all other miners.](image)

We can also use the primitive `createLinksWith(·, ·)` where the first argument is the agent set you wish to consider when creating links, and the second is a closure (condition) you wish to be satisfied before a link is created. Suppose in the *Mining Example*, we create 10 Miner turtles, and set the first five to be red, and the other five to be blue. Then suppose we want all red turtles to be connected with one, and only one blue, and vice versa (i.e. all reds have only one partner (blue), and all blues have only one partner (red)). When creating turtles in the `setup` function, we need to add the following code:

```java
01 if (getWho() < 5){setColor(red())}
02 else {setColor(blue())}
```

where the first line is a conditional, in that if `getWho()`, an integer identifier of the turtle in question, is less than 5, then set its color to `red()`, and the second line is the alternative. Then to the setup function, we would add something like:
where lines 1 and 2 define empty turtle sets, and lines 3 to 5 fill in these sets. That is, we iterate over all Miner turtles, and if their color is red, add that turtle to the red agent set `rAS` (and similarly for blue Miners). Then lines 7 to 10 ask each turtle in `rAS` to create a link (i.e. `createLinkWith(·)` with `rand_blue`, a random turtle from `bAS`, and then remove that agent from `bAS` (line 10), so that another link is not created with it. The output should look similar to Figure 2.3.2.

![Figure 2.3.2: The visual output of links created from red Miner turtles to blue Miner turtles, so that each Miner has exactly one link.](image)

**Exercise.** Try using `createLinkWith(·,·)`, notice the two arguments, to give the same result as Figure 2.3.2.

### 2.3.2 Manipulating Links

Once we have a set of links between turtles, we have a whole set of primitives we can use. Given we are in a turtle context, we can obtain all the undirected links of the turtle in question by using the primitive `myLinks()`. If you do this, you will get an agent set of links, rather than the agents at the end of the links.

Suppose we wanted to obtain the set of agents at the end of all undirected links. We can do this the easy way (using the primitive `linkNeighbors()`), or the hard way,\(^6\) which would use the code:

\(^{6}\)This is only used to show the primitive `otherEnd()`.
2 Basic Methods

```groovy
01  def AS1 = noTurtles()
02  ask (myLinks()){
03      AS1.add(otherEnd())
04  }
```

where line 2 asks all the links of a particular turtle, to add to the agent set AS1, the turtle at the other end of the link (i.e. `otherEnd()`).

Suppose that we wish to remove a link from a set of links of a turtle. That is, if we are in a turtle context, and we have identified a link (call it `link_of_int`) that we wish to be removed. This is done by using the code `link_of_int.die()`. This works when we have identified a link which we want to remove. A related task is if we wish to delete a link, but we are not sure whether or not it exists. This can be done by using the query primitive `isAgent(·)`, as the following code demonstrates:

```groovy
01  if (isAgent(link(miner(1),miner(2))) == true){
02      link(miner(1),miner(2)).die()
03  }
```

where the first line uses the conditional, that if the “agent” (link!) exists, as `isAgent(·)` returns `true` if the agent exists, and `false` otherwise, with that agent being a link, then we remove it by using the `.die()` primitive. We can use this same structure to apply a function to a link, if it exists, simply by replacing line 2 with the function of interest.

2.4 Global and Local Variables

2.4.1 Global Variables

Global variables are used when all agents in the system need access to a particular value (say a `risk_free_rate` in a macroeconomic model), without having to instantiate each agent with a new characteristic which is the same across all agents).

Adding global variables is achieved by using the following code in the UserGlobalsAndPanelFactory.groovy class:

```groovy
01  addGlobal("risk_free_rate", 0.03)
```

where `addGlobal(string, number)` is a method that defines a global variable, where `string` is the name of the variable, and `number` is the initial value. This variable can then be accessed from other classes by simply using its name `risk_free_rate`. Further, it can be modified by other classes. If this global variable is something like `GDP_global`, which needs to be accessed by all `consumer` turtles each period, but changes across periods, and we wish to plot this in real-time (see Section 3.2), we get an error and the run won’t even load. So what we do is create a reporter variable in `UserObserver` by using `double GDP_reporter = 0`, then in the `go` function, include `GDP_reporter = GDP_global`, and then use `GDP_reporter` in the real-time plot.

A cruder way of creating “global variables”, which should generally be avoided unless for some reason, the above method does not work, is to create a variable in `UserObserver` as follows:

```groovy
01  static def global_variable01 = 0.03
```
Then whenever it is to be accesses by other classes (e.g. turtles), we would simply use the notation UserObserver.global_variable01. You need to include the static otherwise it will not be read by other classes.

### 2.4.2 Local Variables

In many cases, you want a variable created for a certain local process and for it to be inaccessible in other parts of the program. Say we want a variable local_variable to be used only in the UserObserver class. We would just use def local_variable = 5 or double local_variable = 5 (if the local variable is a double precision number), and not include static before def or double.

Along similar reasoning, if we wanted a local variable in the setup function, we would define it similarly inside the setup function. We then cannot use it in the go function (or any other function) inside UserObserver. You should make sure that you define local variables in this way so that they can’t accidentally be accessed or changed by parts of the program that should not have access to them.

### 2.5 Other Methods

#### 2.5.1 Agent Sets

We have already used agent sets widely, maybe not referencing them as such, but essentially these are simply sets of agents. For example, the primitive turtles() or patches() returns the agents of that type as a set. We can also create custom agents sets, and then add or remove certain agents. For example, we can create an agent set with all Walker01s within a radius of 5 by defining the agent set def AS1 = inRadius(5) when in reference to a particular Walker01 turtle.

There are many manipulations you can do with an agent set.\(^7\)

#### 2.5.1.1 Creating, Adding, Combining, Removing and Clearing

To create an empty agent set of turtles, using def turtle_set_name = noWalker01s(), and similarly for patches, def patch_set_name = noPatches(). To add an agent to such a set, we would use turtle_set_name.add(·) where the argument is the turtle to be added to the agent set. For example, in the step function for turtles, we can use the following code to define an agent set of patches, and add the patch the turtle is currently on:

```python
01 def my_patch_set = noPatches()
02 my_patch_set.add(patchHere())
```

To combine two agent sets, AS1 and AS2, whilst retaining both, we use the primitives patchSet(·) or turtleSet(·) as appropriate, so then def AS3 = turtleSet(AS1,AS2) results in an agent set with all agents from AS1 and AS2.

When wanting to remove turtles from an agent set, we have a few different ways this can be done. Suppose we have a set of Walker01 agents called AS1, and we have the identifier (say it’s called ident, obtained, for example, using getWho()) of an agent we do not want in AS1 (the agent might or might not be in AS1). We would then use the code AS1.remove(walker01(ident)). We can also remove an agent if we have a primitive which gives an agent (not just it’s identifier). For example, if we have an agent set AS2 of patches, then we can use AS2.remove(patchHere()) when we are in a turtle context, as patchHere() is a turtle primitive.

---

\(^7\)These can be found at http://repast.sourceforge.net/docs/api/hpc/classrepast_1_1relogo_1_1_agent_set.html
If we have two agent sets \textit{AS1} and \textit{AS2}, and we want a third agent set \textit{AS3} of all turtles in \textit{AS1} but not in \textit{AS2}, then we can use code such as \texttt{def AS3 = AS1} and then on the following line, \texttt{foreach{{if (it in AS3){AS3.remove(it)}}},AS2)} which uses the iterator primitive \texttt{foreach(·, ·)} where the first argument is a closure,\footnote{Which is a function of some sort, iterating over the elements of the second argument.} and the second the set/list over which to apply that closure to.

If an agent set \textit{AS1} needs to be cleared, use \texttt{AS1.clear()}, and if it is to be shuffled, use \texttt{AS1.shuffle()} to randomly shuffle all elements.

### 2.5.1.2 Applying a Function

Some common functions are already encoded as primitives, which means that you do not need to program them. A common requirement is, for an agent set, choose the agent that maximizes (or minimizes) a particular function. A function that returns this agent would look similar to:

```groovy
01 def agent_of_interest(){
02     return temp_var = maxOneOf(some_agent_set){
03         your function of agent and other variables
04         that gives a scalar output
05     }
06 }
```

where the first line defines the function \texttt{agent_of_interest()}, to return (line 2) an agent from \texttt{some_agent_set} that maximizes (i.e. \texttt{maxOneOf(·)}) lines 3 and 4, where you specify your scalar output function. A specific example might be:

```groovy
01 def patch_of_max_utility(){
02     return maxOneOf(patchHere().neighbors()){  // it
03         tech * it.labor**0.4 * it.capital**0.6
04     }
05 }
```

where \texttt{tech} is a global variable, \texttt{labor} and \texttt{capital} are patch specific variables, and \texttt{it} refers to the patch in question in the agent set \texttt{neighbors()} — it is a special word in Groovy, similar to \texttt{for}, \texttt{else} and \texttt{if} also being special words. Furthermore, we must be in a turtle context as \texttt{patchHere()} is turtle specific. Further, such a function is typically used in conjunction with a turtle primitive such as \texttt{moveTo(·)} (e.g. \texttt{moveTo(patch_of_max_utility())}).

### 2.5.2 Lists and Maps

A common occurrence is that a variable from each member of a particular agent set is stored in a list. You may want to pass each element in this list through a function, and then choose the largest of that new set. In the class that you have the list (e.g. \texttt{UserObserver}), define a (single variable) function as follows:

```groovy
01 def function_name {
02     Math.log(abs(it))
03 }
```

where line 1 has \texttt{function_name} as the name of the function, and line 2 defines the scalar output function that you want on the variable \texttt{it}; in this case, the function is the logarithm of the absolute value of \texttt{it}.
Then wherever you wish to apply this function to a list element by element (say your list is named list01), then we would just use list01.collect(function_name).

You might want to store a subset of variables of a particular type of agent as a map. Suppose we have a turtle, and it needs to count the interaction with all other turtles. Define a new turtle variable by def your_map = [:] which creates an empty map. Then suppose we want to count the number of times all other turtles are within a radius of 5 of the turtle in question. We would use the code:

```
01 for (guy in inRadius(miners(),10)) {
02   if (your_map[guy.getWho()] == null){
03       your_map[guy.getWho()] = 1
04   }
05   else {
06       your_map[guy.getWho()] += 1
07   }
```

where iterating over all miners within a radius of 10 (line 1) using a for loop (used to refer to each agent as, in this case, guy). If the agent who is calling this loop does not have guy in their map (i.e. line 2), then line 3 creates it. Otherwise the mapping is increased by one (line 6). If you have experience with other programming languages, you will understand the use of maps (sometimes called dictionaries). If not, then it is recommended that you do a bit or research on it.

You can also have maps within maps. As an example, suppose we define each turtle with two types of variables, which we will call open and closed variables; open variables being accessible by all linked turtles, and closed variables being accessed only by the turtle itself.

We can then define two maps as follows:

```
01   def openMap = ["open_var01": 4.1, "open_var02": true, "open_var03": "Jakub"]
02   def closedMap = ["closed_var01": 28, "closed_var02": "broken leg"]
```

then suppose we are in a turtle context, and we want to obtain the average of "open_var01" of all (undirected) linked turtles. This could be obtained by the code mean(linkNeighbors().openMap["open_var01"]), and as long as we do not allow other agents to use closedMap, then those variables will be inaccessible by anyone but the turtle itself. Notice that if the keys in a map are names of variables, they need to be strings (i.e. in quotes ","), which is slightly different that using the def term. Maps can make life easier sometimes, but may make it more complex at other times. If we only had two variables above, one that we wanted linked turtles to access, we could just not use the other variable whenever talking about the linked turtle set; using maps would just complicate things.

**Exercise.** A common example is when patches need to “remember” how many times the different turtles have visited them. In UserPatch, create a map called visitsMap and write code in the go function to achieve the desired result.

### 2.6 Extending Classes and Groups

#### 2.6.1 Extending Classes

In many cases, we may want to create different types of turtles, with differing characteristics. For example, we may want to create turtles called CEO, SeniorManager, MiddleManager, LowerManager and Worker, but we want CEO turtles to have access to all the
methods of all other turtles, SeniorManager turtles to have all methods of other workers other than CEOs, and so on. A very crude way of achieving this is through programming all classes individually (i.e. all classes start with `class Name_of_class extends BaseTurtle{`), however, once we have constructed a turtle class (say the Worker class), we can create a new turtle class (say the LowerManager class), but we change `class LowerManager extends BaseTurtle{` to `class LowerManager extends Worker{`. Then we have all the methods of the Worker class available to the LowerManager class.\(^9\)

**Example.** In time.

A common way to model a number of turtles as a group, which acts as a single entity is by the creation of another agent. Similar to what we did above, we can model a firm as a collection of turtles. This is done as in reality, firms, rather than individual workers, set prices, output, etc.\(^10\) This is easiest done by defining groups of humans as a new turtle (which we called a Firm turtle), and it behaves in certain ways.

Suppose we created a new Firm turtle class as:

```java
01  class Firm extends BaseTurtle {
02      def employeeList = noTurtles()
03      def ownerList = noTurtles()
04  }
```

and we want Walker01 turtles to be part of it. We can use the following code in the `setup` function to create a new Firm for approximately 10% of Walker01 turtles:

```java
01  createWalker01s(numWalker01s){
02      setxy(randomXcor(),randomYcor())
03      if (getWho() < (0.1 * numWalker01s).toInteger()){
04          def me = getWho()
05          hatch(1, {
06              it.setShape("circle")
07              it.ownerList.add(me)
08          }, Firm)
09      }
10  }
```

where if the turtle identifier (i.e. `getWho()`) is below approximately 10% of the number of turtles (i.e. `(0.1 * numWalker01s).toInteger()`), then line 4 creates a local variable called `me`, which is the identifier of the current turtle, and then it hatches a single instance of a Firm turtle, with its shape set to be a circle, and `me` being the owner.

We would also like for each Walker01 turtle to have a new variable which defines their employer (simply done by `def employer = null` and then updated appropriately). A related point is that you need to be vigilant\(^11\) in always updating both the variable `employer` of Walker01 turtles and the `employeeList` of each Firm, as it is a common error for this to be done at inappropriate locations in the code (just make sure that it is updated before using it).

---

\(^9\) This could also be done when first creating a class, but as one of my goals is for you to learn the, in many ways, better, Java implementation of a RePast model, then it’s good to have these small introductions.

\(^10\) Yes, in reality, there is no such thing as a firm, but rather just humans making decisions. However, many times, if modeling a part of your model would be too complex and judged to have a negligible impact on the question trying to be answered, then we can simplify it’s working (this is the case when using random numbers, as the underlying processes may be too complex, and thus need to be simplified for the purpose at hand).

\(^11\) like Batman.
We can then define manager actions in the Firm class, by adding code similar to the following:

```groovy
01 def managerList = noTurtles()
02 def manager_function1(){
03     some function you define managers to have
04 }
05 def manager_function2(){
06     something other function
07 }
08 def managersMove(){
09     manager_function1()
10     manager_function2()
11 }
12 def hiringFunction(some_turtle_set){
13     ask (some_turtle_set){
14         if (some condition here){hire the turtle}
15         else {}
16     }
17 }
18 }
```

Now, we may not want to display firms on the output, so we can add, between lines 6 and 7, the code `it.hideTurtle()` and after setup is run, the circle will disappear from the display (it will still exist).

### 2.6.2 Groups

Extending classes works well if the number of “layers” is exogenous to the model (e.g. we say there are only Worker and Manager turtles). However, if we want the number of “layers” to be an endogenous property of the model, then this method would not be appropriate. A better method would be to group the turtles into a new turtle structure (e.g. a Firm turtle), and then in that turtle class, write methods for the different types of turtles.

As an example, if we want the number of layers of management to be variable, then in a Firm turtle, we would include a map of the sort `def managerLayerMap = [:]`, and then for managers of layer 1 (say one above workers), we would create `managerLayerMap[1] = noTurtles()`, and add to it any turtles who are managers. Once, say, the number of managers of level 1 reaches a certain level (say the global variable `managThresh`), we then create a new manager level by `managerLayerMap[2] = noTurtles()`, and so on. This can be defined recursively so that we don’t need to specify, _ex ante_, the number of management layers. Groovy code for this would be something similar to:
2 Basic Methods

```python
managerLayerMap = [:]

def updateManagerLayers():
    hiLayer = managerLayerMap.keySet().max()
    if (hiLayer != null):
        if (count(managerLayerMap[hiLayer]) > managThresh):
            managerLayerMap[hiLayer + 1] = noTurtles()
    else:
        managerLayerMap[1] = noTurtles()
```

where line 1 creates a map, line 2 creates an update of the manager layers, where line 3 defines the highest management layer. Then line 4 conditions on a management layer existing, and if the number of managers in the highest layer `count(managerLayerMap[hiLayer])` is above the threshold `managThresh`, then add a management layer (line 6). If no management layers exist, then lines 9 and 10 create one.

We may want the firm, or specific subsets of turtles of the firm, to perform certain actions. We can extend classes and also use groups. We can create a Manager turtle, which has methods that they perform on an arbitrary turtle set (e.g. the management layer below), and at the same time, create a Firm turtle which groups together all turtles associated with that firm.

2.7 Scheduling

2.7.1 Time-Driven Events (Discrete Time)

So far we have been using an implicit schedule to determine the timing of events, with the events occurring once each tick. But that means that we assume time is discrete, and events are driven according to time. This is discussed further in Section 4.1.

2.7.2 Event-Driven Time (“Continuous” Time)

What we can have instead is time being driven forward by events, and these events can be scheduled at non-integer times (continuous time).\(^\text{12}\) For example, suppose farmer turtles need to harvest wheat from patches, and how long it takes depends on their ability, a farmer variable, which say comes from a Pareto distribution (specifically `time_to_harvest = wheat / (1 + ability)`). Then once all farmer turtles calculated their `time_to_harvest`, the model could be advances by the minimum of this set (i.e. the next calculation of the model is when the next farmer turtle has finished their harvest). When using “continuous” time, you might want to change the `go` procedure to another name, such as `begin` or `start`, as your code in this procedure will tell the program, using the primitive `tickAdvance()` with a number inside the parentheses, to advance ticks. Compare this with having a `go` procedure, where at the beginning or end, you have `tick()`, and then you press `go` to execute one time step. In continuous time, you have `setup`, which initializes the model, and then `begin`, which starts the model, and it then runs on its own accord. To prevent a model from running forever, a termination condition is required in `begin`, such as if `(ticks() > 100){stop()}`, or for a better technique, see Section 4.1.

\(^{12}\)Technically, it is not continuous, but computable time, however, this distinction is not of great importance for our purposes, so thinking of time being continuous will not affect your understanding.
In the original *Mining Example*, suppose we wanted to change how long it takes for a Miner turtle to extract resource, according to their ability. We could have the following:

```java
00 def ability = Math.random()
01 double timeTilCompletion
02 def currentTask = "walking" //This is either "walking" or "mining"
03 def updateCurrentTask(){
04   if (timeTilCompletion == ticks()){
05     if (currentTask == "mining"){
06       stockpile += patchHere().resource
07       patchHere().resource = 0
08       currentTask = "walking"
09       timeTilCompletion = ticks() + walkingTime
10     }
11   } else if (currentTask == "walking"){
12     if (patchHere().resource > 0){
13       timeTilCompletion = ticks() + \  
14       patchHere().resource / (1 + ability)
15       currentTask = "mining"
16     }
17   } else {
18     setHeading(getHeading() + Math.random()*60 - 30)
19     forward(walkingLength)
20     timeTilCompletion = ticks() + walkingTime
21   }
}
```

where `timeTilCompletion` is in terms of the ticks (so if `ticks()` is currently 24, and a walk is begun and takes 5 ticks, then `timeTilCompletion` would be 29). Another variable is introduced called `currentTask`, which is a string, and defines what task the miner is currently involved in ("walking" or "mining"). Line 4 asks the turtle if `timeTilCompletion` is the current time, and then defines actions to take if "mining" (lines 5 to 10) or "walking" (lines 11 to 21). If the turtle is currently "walking", then if its current patch has resources, it changes to "mining", and adjusts `timeTilCompletion`, which depends on its `ability` (which was define in line 0). By now, you should be able to read and understand the details. The new function `begin` is given by:

```java
01 def begin(){
02   while (ticks() < maxTicks){
03     tickAdvance(minOneOf(miners(),{it.timeTilCompletion} - ticks()))
04     ask (miners()){
05       updateCurrentTask()
06     }
07     ask (patches()){
08       setPcolor(scaleColor(105,resource,0,12))
09     }
10   }
11 }
```

where line 2 uses a `while` loop, executing lines 3 to 9 (the condition in the parentheses can also be when an event occurs, such as `while (sum(miners().stockpile) < maxGDP){}`. Line 3 is the key to “continuous” time models; it uses the primitive `tickAdvance()`, which
advances `ticks()` by the number in the parentheses. That number is the minimum of `timeTilCompletion` of all Miner turtles, less the current ticks (i.e. - `ticks()`). This advances time by that amount, and at that point, there will be at least one Miner turtle who will satisfy line 4 of the `updateCurrentTask()` above (by construction). So then we ask all Miner turtles to `updateCurrentTask()`, which updates their actions, if they have finished their current task. To see that ticks no longer advance at equally spaced, discrete time steps, between lines 6 and 7, include `print ticks()`, and you will see in Eclipse Console the ticks the model uses. You can think of this as continuous time, or more appropriately, event drive time.

When using continuous time, a display may not update until the stop condition has been reached. To overcome this, we can manually update the display by renaming and changing the `begin` function to `beginStep` function:

```python
01 def beginStep(){
02     def init_step = ticks().toInteger()
03     while (ticks().toInteger() - init_step < 1){
04         tickAdvance(minOneOf(miners(),{it.timeTilCompletion}).timeTilCompletion
05             - ticks())
06         ask (miners()){  
07             updateCurrentTask()
08         }
09     }
10     ;
11 }
```

and then we still have the event-driven time structure, but the model stops whenever `ticks()` increases to a new integer, and the display is then updated. This is useful when first creating a model to visualize output. To see when the model runs the `while` loop, we can add `print ticks()` between lines 5 and 6, and when the model is run, we see that after the first 5 or so steps, the model is updates off integer ticks, exactly as expected of Miner turtles with random ability.

Even better as a testing view is to update output whenever a Miner turtle’s `timeTilCompletion` changes from zero to a positive number, as then we can see the action of each turtle, whenever at least one Miner turtle does something.

**Exercise.** Adjust the code above so that the display gets updated whenever `timeTilCompletion` of a turtle changes from zero to a positive number (i.e. whenever `tickAdvance(·)` is called).

### 2.8 Stochastic Elements

In some instances, you may want one part of the code to use a random number generator, but for the random numbers in this section to be constant across all runs. In the Mining Example, we may want the three resource areas to be fixed for all runs, but initial positions of agents, changes in angles when moving, and abilities being random across runs.

Suppose we want all patches to be instantiated in a certain way. Then in the setup procedure, before setting `randomSeed(123456)` where 123456 is a fixed integer that we want to use to seed out random numbers that do not change across runs, we define `def seed = newSeed()`, which gives a new seed. Then we would include `randomSeed(123456)`, and after the block of code, we would include `randomSeed(seed)` to create a new random seed. That is:
Exercise. In the Mining Example, implement this so that whenever setup is pressed, the center of the three patches remains in the same location, but both the color of the patches and the location of the Miner turtles is stochastic (and changes every time you press setup).
3 Testing the Model

Don’t ever think that your program has no bugs, and that it works exactly as you want it. Once a model works approximately as you expect (i.e. doesn’t give you ridiculous results, like infinity for a variable that you know must be finite), you need to test it, and test it heavily. For this, a few tools should be used.

3.1 Data Sets

In a working model, we run the model (Arrow 10 of Figure 1.1.1), and after the Java window opens (Figure 3.1.1) but before pressing the “Initialize Run” button (Arrow 1 of that same figure), we go to the “Scenario Tree” tab in the bottom-left corner (Arrow 2). You will see a tree structure of files. Right click on the “Data Sets” file (Arrow 5), and click on “Add Data Set”.

![Figure 3.1.1: The initial screen after running the model (this is what will be referred to as the Java Screen).](image)

You will see a window as in Figure 3.1.2. In the “Data Set Id” dialogue box, type what you wish to call this data set (e.g. Average Heading, or Data Set 1), and in the “Data Set Type” box, select the type of data set you wish to create — is it an aggregate (e.g. mean income of all consumers) or non-aggregate (e.g. income of each consumer turtle) data set?
Testing the Model

3.1.1 Aggregate Data Sets

Click “Next” on Figure 3.1.2 after selecting “Aggregate”. In your output, you usually want ticks (and possibly run number and random seed), so select which you need. Then on the “Method Data Sources” tab, click the “Add” button and you should get something like Figure 3.1.3.

Double click on the cell below “Agent Type” (“BasePatch” in Figure 3.1.3), and then choose from which agent your aggregate variable applies to. Say it’s “Base Turtle” (choose it) and say we want the mean heading of all BaseTurtles. Double click on the adjoining “Methods” cell, and choose the variable of interest (getHeading in our example). Then in the adjoining “Aggregate Operation” cell, double click and select “Mean”. If you clicked the “Next” button at this point, you will get an error message; you must give this data source a name, so double click in the (empty) “Source Name” cell, and type in an appropriate name (e.g. “Mean Turtle Heading”). Now click “Next”.

You will now be in a screen similar to Figure 3.1.4. Here, you can tell RePast when to start the data source (“Start Time”), the priority in adding the data to the data set (“Priority”), how long you want the data collected (“Duration”; leaving this blank mean indefinite duration), how many times you want this action performed (“Frequency”), and lastly, every how many intervals you want the data collected (“Interval”). Typically, all the settings in this window are what is required, so you can just press “Finish”.

![Figure 3.1.2: The data set editor screen.](image)

![Figure 3.1.3: The Method Data Sources tab of the data set editor screen.](image)
In the Scenario Tree, you will see a new branch added from “Data Sets”, which is the data set you have just created. Before pressing “Initialize Run”, click the save icon (this will save the fact that you want that data set, even if you close the window and re-run the simulation).

**Obtaining Globals Data** Suppose the data set you want is to follow is an aggregate variable you made (e.g. the GDP of an economy). A reporter for this global variable is needed (outlined in Section 2.4.1), as we cannot use it directly if it was created by `addGlobal("global_variable01")` in UserGlobalsAndPanelFactory. Then the variable is just accessed from UserObserver in “Agent Type” in Figure 3.1.3, and then the appropriate variable is selected. You can choose “Sum” or “Mean” as there is only one such variable each period.

**Obtaining Data of Non-Standard Functions of Aggregate Variables** Suppose that we want to apply an arbitrary function to a turtle-specific variable for all Walker01 turtles, and then sum the result. Clearly, in Figure 3.1.3, we have a limited set of functions (basic ones, such as mean, standard deviation, etc.), so this can be overcome in a number of ways. We could create a new turtle variable which is the output of the function of interest, and then use the above procedure (simply using sum). Creating the new turtles variable amounts to adding the code `def variable_of_interest = 0` to Walker01.groovy, and then in the `go` function, adding the code:

```groovy
def go(){
    ... ask (Walker01s()){
        ... variable_of_interest = arbitrary_function(turtle_specific_variable)
        ... }
    }
```

where `arbitrary_function` is a function that you apply to the `turtle_specific_variable` that you require. Finally, to plot the result, we would just use the procedure described above to sum this new variable across turtles.

**Example.** Suppose a turtle class, called `consumer`, has a variable `income`. But we want to find out the average disposable income of all turtles. Firstly, we need to find the disposable
(after tax) income of each consumer, and then average this over all consumers. In most countries, income tax is progressive, in that the marginal rate of tax changes at certain levels of income. Suppose that in this economy, net income is given by the function:

\[
ny(y) = \begin{cases} 
y & \text{if } 0 \leq y < 10 \\
10 + 0.8(y - 10) & \text{if } 10 \leq y < 30 \\
26 + 0.5(y - 30) & \text{if } 30 \leq y < 50 \\
36 + 0.4(y - 50) & \text{if } 50 \leq y
\end{cases}
\]

To find net income of each agent, in the Walker01 class, add the following function:

```java
01 static def netIncome = 0
02 :
03 def f_net_income(inc){
04     if ((0 <= inc) && (inc < 10)){
05         netIncome = inc
06     }
07     else if ((10 <= inc) && (inc < 30)){
08         netIncome = 10 + 0.8 * (inc - 10)
09     }
10     else if ((30 <= inc) && (inc < 50)){
11         netIncome = 26 + 0.5 * (inc - 30)
12     }
13     else if (50 <= inc){
14         netIncome = 36 + 0.4 * (inc - 50)
15     }
16 }
17 :
18 def step(){
19     :
20     L = []
21     ask (consumers()){
22         :
23         L.add(f_net_income(income))
24         :
25     }
26     meanNY = mean(L)
27 }
```

where lines 3 to 16 implement the \( ny \) function, and then line 20 creates an empty list, line 23 adds disposable income of all consumers to the list, and line 25 finds the mean net income.

### 3.1.2 Non-Aggregate Data Sets

Firstly, it is assumed you have read Section 3.1.1 on aggregate data sets. Now, when adding a data set, we choose “Non-Aggregate” as the “Data Set Type” in Figure 3.1.2. Again, check the appropriate boxes in the “Standard Sources” tab, and then in the “Method Data Sources” tab, click “Add”. From the drop-down menu, choose the class from which you want to obtain the non-aggregate data (e.g. Walker01), then double click on the appropriate cell under the “Method” heading, and choose the data you want. Because there are usually many of a particular turtle (or patch), it is also best to include the identifier for the turtle.
(or patch) in the output. Say we added `netIncome` for consumer turtles, then in our output, we also want to identify which turtle had that net income. This is done by simply clicking “Add” again, choosing that same turtle type, and then selecting `getWho` (which returns the identifier for the turtle).

Then click “Next” and “Finish” (the last screen is the same as for aggregate data sets) to create the data set.

### 3.1.3 Exporting Data Sets

Once a data set of interest has been created, it can be exported by using a text sink. Before initializing the run, in the “Scenario Tree” tab, after creating a data set, right click on “Text Sink” and select “Add File Sink” (Arrow 6 of Figure 3.1.1); you should get something like the following figure:

![Figure 3.1.5: The initial window of the file sink editor.](image)

Name the file something appropriate (e.g. Mean Headings over Time). From the “Data Set ID” drop-down menu, choose which of the created data sets you wish to use data from. Then click on the specific elements you want in you file sink from the left box, and add them (using the arrows) to the right box. After clicking “Next” you should get the following:
3 Testing the Model

Figure 3.1.6: The final window of the file sink editor.

In the “File Name” box, change the name to something appropriate (e.g. MeanHeadings.txt), which saves it in the Project folder, or alternatively, browse the location you wish to save the file, and just type the file name in the browse box; then press “Finish”. Click the save icon on the RePast Simphony window to save this text sink.

Now every time you run the model, a text sink will be produced, which you can open with a spreadsheet and analyze the results.

3.2 Real-time Plots

Sometimes, however, you are just testing and want the plots to appear as the model runs. Two kinds of real-time plots are available in RePast Simphony, time series plots and histograms. Other plots can be obtained by outputting data (as a text file), and then analyzing it externally. Further, the plots obtained in real time shouldn’t be used for publication purposes. Rather externalize the data, and use that (this is because if anyone else wants your data, you will have it).

3.2.1 Time Series Plots

Once you have a model that runs, we can obtain a plot of certain variables (sometimes you need to modify or add a new variable in your code so that you get the desired output). Suppose we want the mean direction of all turtles, plotted at each time step. Create a data sink. Before initializing the simulation, right click on “Charts” in the “Scenario Tree” and choose “Add Time Series Chart”; you should get Figure 3.2.1.
3 Testing the Model

Figure 3.2.1: The initial screen of the time series chart creator.

Give the Chart an appropriate name, and choose the “Data Set” you wish to plot. After clicking “Next”, you should get something like Figure 3.2.2; choose the variables from the data set you wish to plot on the one chart. You should only choose those variables that have approximately the same range, as if we plot a variable with a range [900, 1000] and another with a range [0, 1] on the same chart, then both will not be visually appropriate and will not show the precise features you want to observe.

Figure 3.2.2: The second screen of the time series chart creator.

Click “Next”. In this last window, you can change properties of the chart (labels, colors, etc.). The “Title” of the chart will be the title of a new tab in the RePast Simphony window after you initialize the run. After clicking “Finish”, click “Save” in the RePast Simphony window.

Initializing the run, you should notice a new tab which plots, in real time, the data set you chose.

3.2.2 Histograms

To create a real-time histogram chart, right click “Chart” in the Scenario Tree and select “Add Histogram Chart” (Arrow 7 in Figure 3.1.1), resulting in something like the following figure:
Choose an appropriate name, then in the “Data Set”, choose your desired non-aggregate data set, and then choose the data you want for the histogram. After clicking “Next”, you get to choose the number of bins (“Bin Count”). Initially, choose a large number, and then gradually find a number that allows you to see the distribution without having too many or too little (it’s a bit hit and miss). Next, if you know the approximate range that the variable you want the histogram of can take, then choose static from the “Histogram Type” drop-down menu, and fill in the minimum and maximum boxes; otherwise choose dynamic. After clicking “Next” you get edit the chart properties (title, color etc.). Click “Finish” and then initialize the run. Notice the tab with name the same as the title of the chart you entered.

3.3 Statistical Testing

When testing a model (or submodel), we need to know that what we expect a model to do is actually what it does. That is, we ideally want to test each command in our model to see that it does what we expect it to. We can assume things like `Math.random()` actually gives a random number between 0 and 1 (although, it’s interesting to read some early random number generators were not random, so it’s always good to keep that in the back of your mind so as you don’t get too cocky about the results of your model).

For this end, we may want to output a text file away from the Java screen. In the setup function, near the top, we would add:

```java
01 File file_name_used = new File("C:/Jakub’s C Drive Documents/ReLogo Projects/test_file01.txt")
```

where this creates a new .txt file in the directory specified, called `test_file01`. So as to prevent the file growing each time you run the program (which you will be doing quite a lot whilst making and testing it), in the `setup` function, add the code `file_name_used.delete()`, so as to delete the contents of this file.

Then suppose we wish for each line to represent a tick, so that the first number of each line is the tick count, then, separated by a comma, another variable of interest, and separated by another comma, another variable of interest. The code would looks something like:
3 Testing the Model

```bash
01  file_name_used << ticks() + ", " + variable_of_interest01 + ", " + variable_of_interest02 + "\r\n"
```

where the the notation `<<` essentially says that anything on the right side gets appended to the file `file_name_used`. We also have `", "`, which also adds a comma and whitespace (for readability and for export to spreadsheet applications). The same pattern is followed for adding the other two variables, but for the end of a line, we don’t want a comma, we use a new line, which is obtained by "\r\n". The `+` signs are is just “adding” all these variables together, to be appended onto the file `file_name_used`.

A related topic is to append onto a line a list, say some attribute of the patch a turtle is currently on, and the attributes of the patches in the Moore neighborhood. This can be easily done by the code

```bash
01  file_name_used << ticks() + ", " + patchHere().variable_of_interest01 + ", " + patchHere().neighbors().variable_of_interest01 + "\r\n",
```

however when exporting to some spreadsheets, the square brackets, denoting a list, will not be removed. To overcome this, what we just use a loop over the values:

```bash
01  f1 << ticks() + ", " + patchHere().variable_of_interest01 + ", "
02  for (i in patchHere().neighbors()){
03      f1 << i.variable_of_interest01 + ", "
04  }
05  f1 << "\r\n"
```

where line 1 is as described above, lines 2 to 4 are the loop across the neighboring patches (in no particular order), and adding to the same line of the text file, their value of `variable_of_interest01`.

In this way, we can create and add to multiple files at once, where we just label them differently. This text file output can now be used for statistical and other testing.
4 Experimentation

Only read this section once you have a model that works as expected, and you have debugged and tested it to a reasonable extent. This section looks at how sensitive certain results of your model are to changes in underlying parameters.

4.1 Basics

Once you have a model, you want to be able to efficiently run the program multiple times, and then obtain output of each run, so as to be able to test the full extent of the model and analyze its properties statistically. To do this, we need to do a few things:

1. specify what it is to be a single “run” of the mode (i.e. scheduling),
2. specify what parameters we want to vary across (i.e. the parameter space we wish to explore),
3. specify how many times the model is run for each point in the parameter space.

For the following explanation, we suppose our model is running as expected.

4.1.1 Defining a “Run” (Scheduling)

To satisfy Point 1 above, we need to code into the program the specific timing of events and when the events should stop (i.e. the schedule of events). To do this, we import the following package in the UserObserver class:

```java
import repast.simphony.engine.schedule.ScheduledMethod;
import repast.simphony.engine.environment.RunEnvironment;
```

We then need to define which methods should be scheduled. Specifically, we want `setup` and `go` to be scheduled explicitly. Thus the line before each of these functions, add `@ScheduledMethod(·)` where inside the parentheses, we include specific values. So for the `setup` and `go` functions, we would have something like:

```java
01 @ScheduledMethod(start = 0d)
02 def setup(){
03    ;
04    }
05
06 @ScheduledMethod(start = 1d, interval = 1d)
07 def go(){
08    ;
09    }
```

where in line 1, `start = 0d` means that the following method is scheduled at tick 0, but as the tick needs to be a double precision number (not an integer or any other type), with the `d` in 0d defining this. In line 6, the function following (i.e. `go`) is scheduled to start at tick 1 (i.e. `start = 1d`), with this method being scheduled at intervals of 1 tick (i.e. `interval`
Experimentation

You have to be careful when you run the program manually (i.e. by using the “Go” button in the user interface), as then ticks will run at double speed (once because of the @ScheduledMethod and once because you pressed the “Go” button), even though the visual output will be running at normal speed (as will the tick counter in the visual interface). However, when running batch experiments, everything will run at normal speed.

So far, this has defined the schedule of the events, but not when to stop the run. Suppose we want to stop it after a certain number of ticks (say 1000). Inside the setup function, we would write

```
RunEnvironment.getInstance().endAt(1000)
```

which tells the environment to run, and end at tick 1000. Clearly, we can define a global variable (say called endTime) which replaces the 1000.

Alternatively, suppose we wish to end a run when some particular event occurs (say all agents of one particular type die), but still have a finite end to the program (as some agents may never die!). In this case, it would be easiest to not have `RunEnvironment.getInstance().endAt(1000)` in setup. Rather, in the go function, the following code would be added as the second last thing executed:

```

```}

where the `||` in line 3 is the groovy operator OR. Notice that we use `ticks() >= 999` if we want 1000 ticks, as ticks start at zero. So line 3 says that if the number of Walker01 turtles is zero, or the ticks exceed 999, then execute line 4 (which ends the run).

4.1.2 Defining the Parameters to Vary

To address Point 2 (i.e. the parameters to vary for an experiment), suppose in UserGlobalsAndPanelFactory, we have a global variable (which does not change within each run), added using `addGlobal(“risk_free_rate”,0.03)` (i.e. risk_free_rate is the variable that is constant for each run, but we wish to vary it over different runs of the model).

Now, in the your_project_name.rs folder in the Package Explorer, open parameters.xml and click on the source tab (see the following figure).

![parameters.xml file](image)

Figure 4.1.1: The parameters.xml file.

---

1 The last thing is `tick()`. 

46
Notice that you have a few blocks of source code that look similar to:

```xml
<parameter name="default_observer_maxPycor" displayName="default_observer_maxPycor"
  type="int" defaultValue="16"/>
```

each which may be written on one or multiple lines. This is written in Java rather than Groovy. But it should still be quite readable. Line 1 defines the name, display name and type for a parameter (the name is not the name of the parameter in UserGlobalsAndPanelFactory, but something similar; e.g. add the text _Param to the variable in the code). Line 2 defined the default value.

Now, `risk_free_rate` is the parameter we wish to vary in experiments. First, what we need to do is copy and paste below a block of code as above, but with some added parts and some other parts changed:

```xml
<parameter name="risk_free_rate_Param" displayName="Risk Free Rate" type="double"
  defaultValue="0.03"
  isReadOnly="false"
  converter="repast.simphony.parameter.StringConverterFactory$DoubleConverter"/>
```

line 3 defines whether the parameter is read only (="true") or read and writable (="false"). Line 4 may be a little confusing, but the converter essentially converts a string into whatever type you defined the variable to be (notice in line 2, the number is written in quotes ", thus it is a string, but we require it to be a double precision number, from type in line 1).

Lastly, in UserObserver, import the following:

```java
import repast.simphony.parameter.Parameters;
```

Then in the setup function, the first thing after `clearAll()`, should be the following code:

```java
def setup(){
  Parameters params_of_interest = RunEnvironment.getInstance().getParameters()
  risk_free_rate = params_of_interest.getValue("risk_free_rate_Param")
}
```

where line 2 defines the parameters which RePast can read and (later) make variable across runs. I called this set `params_of_interest`, but you can give any name. Line 3 defines the global variable to be a `param_of_interest`, and refers to the parameter "risk_free_rate_Param" which we defined above. This overrides the value of the parameter set in UserGlobalsAndPanelFactory, if it was set. To define other such variables (e.g. `num_banks`), we just add another line similar to 3, but first defining the variable in UserGlobalsAndPanelFactory and also in the parameters.xml file.

### 4.1.3 Batch Runs

Now we are ready to run our experiment. Say we wish to see how sturdy our model is to changes in the `risk_free_rate` over the range \{0.025, 0.026, \ldots, 0.034, 0.035\}, and

---

2The website http://repast.sourceforge.net/docs/api/repast_simphony/repast/simphony/parameter/StringConverterFactory.html gives other converters which can be used, depending on the type of variable required.
Experimentation

num\_banks over the set \{10, 100, 1000\}. What we would do is run the model, but before initializing the run (Arrow 1 in Figure 3.1.1), we add a data sink for a variable or a set of variables of interest (say GDP, unemployment, and inflation), and also a file sink from that data sink (see Chapter 3 for details).

In the bottom left of the Java window, you should see a “Parameters” tab (Arrow 3 in Figure 3.1.1); open it.\(^3\) There should be three buttons, but the one of interest is “Parameter Sweep”. Clicking this opens a window (Figure 4.1.2).

![Figure 4.1.2: The initial window when setting up a parameter sweep.](image)

The “Repeat Count” dialogue box is the number of replicants you wish at each point in the parameter space (i.e. given a set of parameters, this is the number of independent times the code is run). This is used for statistical analysis.

Next, we have a number of parameters we can vary, and notice, we also have the parameters we created in the parameters.xml file. On the drop-down menu of risk\_free\_rate choose “Number”, and you will get something that looks like Figure 4.1.3. In the “From” dialogue box, write 0.025, in the “To” box, write 0.035, and in the “Step” box, write 0.001. This will vary the risk\_free\_rate from 0.025 to 0.035 at increments of 0.001 (exactly as we want above).

![Figure 4.1.3: Part of the initial window when setting up a parameter sweep over the global variable risk\_free\_rate from 0.025 to 0.035 in steps of 0.001.](image)

Now, for num\_banks, this variable does not have regular increments, so instead, from its drop-down menu, choose “List” and the type 10 100 1000 into the “Blank Separated” dialogue box. Notice that the list of numbers is separated by blank spaces and not commas.

\(^3\)It may be that the “Parameter Sweep” button is not visible, so what you want to do is stretch the parameters window.
In a similar fashion, you can change any of the other parameters. Once satisfied with the experiment you wish to run, click the “Save Run/Parameter/Execution Configuration”, and then click “Submit For Local Execution”. You should get a window similar to that shown in Figure 4.1.4, if all goes well.

Figure 4.1.4: If all goes well, you should end up with a window like this. If an error occurred, then this will usually show up here (e.g. if there is a RunTime error in your model).

Further, you will have created an output .txt file wherever your project is saved. If you open this in a spreadsheet program (e.g. MS Excel), you can analyze the experiment at your leisure. You can also “sync” statistical packages if you so wish.