Learning to program simulations in *Mathematica*.

by W. Garrett Mitchener

```
In[1]:= Date[]

Out[1]= {2005, 8, 9, 17, 17, 40.140521}
```

Basics

■ Definitions

Mathematica allows you to define things in three ways.

First: Direct assignment. This is done with the := symbol, and it creates a global definition. The pattern you give on the left of the := will be replaced with exactly the expression you give on the right.

Here's an example that defines y to be a linear expression with an x in it.

```
In[2]:= y := 5 x + 3

In[3]:= y

Out[3]= 3 + 5 x
```

To see all definitions associated with a symbol, you can enter a ? followed by the symbol's name.

```
In[4]:= ?y
        Global`y
        y := 5 x + 3
```

Now suppose we define x to be -8:

```
In[5]:= x := -8
```

Now when you ask for the *value* of y, you get the expression $3+5x$ evaluated with whatever other definitions *Mathematica* has been given, in this case, with $x := -8$.

```
In[6]:= y

Out[6]= -37
```

But if you ask to see the definition of *y*, it's still an expression:

```
In[7]:=  ?y
         Global`y
         y := 5 x + 3
```

So if you re−define *x* and ask for the value of *y*, you get something different than we got before.

```
In[8]:=  x := 10
In[9]:=  y
Out[9]=  53
```

The second kind of assignment is assignment with evaluation, which is done with the = symbol. This is very much like direct assignment, but the pattern on the left of the = is assigned to be the *value* of the expression on the right. In other words, *Mathematica* does work on the right−hand−side before making the assignment.

Let's define *z* like this.

```
In[10]:=  z = y^2 + x
Out[10]=  2819
```

Now the definition of *z* is:

```
In[11]:=  ?z
          Global`z
          z = 2819
```

So if we change *x*, it causes a change in the value of the expression *y*, but doesn't change the definition of *y*. But, it doesn't change the value of *z* because there is no *x* or *y* in the definition of *z*. *Mathematica* has already replaced all the *x*'s and *y*'s in the expression we used to define *z* with the value they had at the time we made the definition.

```
In[12]:=  x := 2
In[13]:=  y
Out[13]=  13
In[14]:=  ?y
          Global`y
          y := 5 x + 3
In[15]:=  z
Out[15]=  2819
In[16]:=  ?z
          Global`z
          z = 2819
```

You sometimes need to remove definitions because you made a mistake, or perhaps you've re−written something and an old definition is in the way.

You erase a definition by assigning "nothing" to a symbol, which is done with the =. operator:

```
In[17]:=  x =.
In[18]:=  ?x
          Global`x
```

Now watch what happens if we ask for the value of *y*. *Mathematica* does the best it can with evaluating *y*, but since *x* is no longer defined, it evaluates to itself rather than a number.

```
In[19]:=  y
Out[19]=  3 + 5 x
```

You can also get rid of unwanted definitions with the `Clear` function.

```
In[20]:=  Clear[y]
In[21]:=  ?y
          Global`y
```

The third kind of definition is through a rule table. *Mathematica* applies all the definitions you've given it through := and = whenever it has to evaluate an expression. But a rule table gives a list of definitions that only apply when you decide to use them. A rule table is a list of rules, and is entered as rules separated by commas between curly braces.

Here's an example of a rule table that defines *a* to be 2, *b* to be 3, and *c* to be 47. The arrow → is entered by typing −> and *Mathematica* magically replaces the − and > with an arrow symbol when you type the next character.

```
In[22]:=  {a → 2, b → 3, c → 47}
Out[22]=  {a → 2, b → 3, c → 47}
```

The definitions in a rule table aren't used until you apply them to an expression with the `/.` operator. Then, the expression is evaluated with those definitions.

Here's an example using the rule table from before. Without the rule table, this expression evaluates to itself because we haven't define *a*, *b*, or *c*.

```
In[23]:=  a + b^2 + c^3
Out[23]=  a + b² + c³
```

But if we apply the rule table, it evaluates to a number:

```
In[24]:=  a + b^2 + c^3 /. {a → 2, b → 3, c → 47}
Out[24]=  103834
```

Rules made with the arrow → behave like assignments made with = in that the right hand side is evaluated.

To illustrate this, notice that in this rule table, all the expressions on the right hand side of its arrows are evaluated.

```
In[25]:=  {u → x^2, v → 1 - y}
Out[25]=  {u → x², v → 1 - y}
In[26]:=  u - v /. {u → x^2, v → 1 - y}
Out[26]=  -1 + x² + y
```

Occasionally, you need rules that don't evaluate their right−hand−side. To do that, you enter :> instead of −> and it produces a different kind of arrow ⧴. These definitions behave like :=. You don't need these rules very often. I'll show an example later.

■ Functions

The concept of a function is central: Most of the work done in *Mathematica* takes place through functions. A function is a definition that defines an expression of the form $f[x, y, z, ...]$ to have a value that's an expression in x, y, z ...

> Note that this is the computational notion of the term *function*. In calculus, you use the analytical definition of the term *function*, which is a mapping between two sets with particular properties, and it may or may not be anything easy to write down or compute. Sorry about the confusing terminology but there's nothing I can do about it.

Here's how to define a function that gives two times a number plus 3. I've used x for the argument.

```
In[27]:=  f[x_] = 2 x + 3
Out[27]=  3 + 2 x
In[28]:=  f[10]
Out[28]=  23
```

A couple of things to note.

- *Mathematica* uses square brackets [] to indicate the arguments of a function. That's because it allows you to put two expressions next to each other with a space to indicate multiplication, as in $x\,y$. If you used parentheses, then the expression $s\,(x+y)$ could either mean the product of s and $x+y$ or the value of the function s at $x+y$.
- The left hand side of the assignment should be a pattern. A plain symbol like x in a pattern only matches itself. You have to indicate that a symbol is to be replaced by the argument's value by making it a pattern variable. There are several ways to do this, but the easiest is by adding an underscore _.

Here's what happens if you forget the _ :

```
In[29]:=  g[x] = 5 x - 8
Out[29]=  -8 + 5 x
```

So far so good:

```
In[30]:=  ?g
          Global`g
          g[x] = -8 + 5 x
In[31]:=  g[x]
Out[31]=  -8 + 5 x
```

But this is clearly wrong:

```
In[32]:=  g[8]
Out[32]=  g[8]
```

Here's the same problem using a rule table:

```
In[33]:=  h[8] /. {h[x] → 2 x^2}
Out[33]=  h[8]
```

What's happened is that by defining *g*[*x*] and *h*[*x*] we've given a definition that only applies when the argument is the symbol *x*. Instead, what we usually want is to make *x* a pattern variable, so that the definition applies no matter what the argument to the function is.

```
In[34]:= Clear[g]

In[35]:= g[x_] = 5 x - 8

Out[35]= -8 + 5 x

In[36]:= ?g

         Global`g

         g[x_] = -8 + 5 x

In[37]:= g[x]

Out[37]= -8 + 5 x

In[38]:= g[y]

Out[38]= -8 + 5 y

In[39]:= g[8]

Out[39]= 32

In[40]:= h[8] /. {h[x_] → x^2}

Out[40]= 64
```

Again, we can get rid of unwanted definitions by calling `Clear`:

```
In[41]:= Clear[f, g]
```

■ Exercises

Exercise 1

Define a function that gives the square of a number plus seven.

Exercise 2

Define a function that takes the radius of a circle and gives its area. To get π, type Pi or click on the π symbol in the basic input palette.

```
In[42]:= Pi

Out[42]= π
```

A random walk simulation

Let’s say you want to simulate a random walk on the numbers 1, 2, ..., n. Imagine these numbers written out in a line and an ant that can crawl from one number to the next. At each instant in time, the ant can either step left, step right, or stay put, based on a random number. We want to simulate paths the ant takes.

■ First simulation

We’ll keep the first implementation very simple. The ant goes left or right or stays with probability $\frac{1}{3}$, and if it gets to the end a tries to walk off the end, it stays put instead. We’ll represent the ant’s location as a number between 1 and $n = 10$.

Conditional functions

We need to define a function that takes the ant from one step to the next. Mathematically, we imagine taking the current location of the ant k, and a random number u, uniformly distributed between 0 and 1. Our update function should do something like this:

$$f[k, u] = \begin{cases} \text{min}[k+1, 10] & \text{if } u < \frac{1}{3} \\ \text{max}[1, k-1] & \text{if } u > \frac{2}{3} \\ k & \text{otherwise} \end{cases}$$

So if $u < \frac{1}{3}$, the ant tries to move right, but if it tries to go past 10, it has to stay put. If $u > \frac{2}{3}$, the ant tries to move left, but if it tries to go past 1, it has to stay put. That gives us the transition rule we wanted. Now how do we explain this to *Mathematica*?

The function will take the current location k, and a number u between 0 and 1 as arguments. The number u is how we’ll decide whether the ant moves right or left or stays put. (We’ll give our function a random number in a minute.) We’ll also call it MoveAnt instead of f. The more descriptive the function name, the easier it is to remember what it’s supposed to do. This is what we have so far:

```
MoveAnt[k_, u_] :=
```

Now we have to express a decision. If $u < \frac{1}{3}$, then try to go right:

```
MoveAnt[k_, u_] := If[u < 1 / 3, Min[k + 1, 10], ...]
```

This introduces two built−in *Mathematica* functions, `Min` and `If`, and the comparison operators.

The `Min` function returns the smallest of its arguments. It also works on lists. Notice that if you give `Min` non−numeric arguments, it doesn't know quite what to do, so it remains unevaluated. There's also a `Max` function that returns the largest of its arguments.

```
In[43]:=  Min[1, 2, 3]
Out[43]=  1
In[44]:=  Min[-10, -11, -30]
Out[44]=  -30
In[45]:=  Min[{1, 2, -1}]
Out[45]=  -1
In[46]:=  Min[a, b, c]
Out[46]=  Min[a, b, c]
```

The `If` function is one way to express decisions. It takes three arguments: (1) a Boolean expression (that is, something that evaluates to `True` or `False`), (2) an expression to evaluate if the first argument is `True`, and (3) an expression to evaluate if the first argument is `False`.

```
In[47]:=  If[True, 4, 5]
Out[47]=  4
In[48]:=  If[False, 4, 5]
Out[48]=  5
```

You often use comparison operators as the first argument to `If`. These compare two numbers and return `True` or `False`. Less−than `<` and greater−than > are pretty obvious. If you want less−than−or−equal−to, type `<=` and when you type the next character, it will magically turn into `≤`. Same for `>=` which turns into `≥`. If you just want equal−to, you have to be careful. Remember that `=` makes a definition, so you can't use that. Instead, *Mathematica* has a separate notation, a double equals sign `==` that magically turns into `==`. This is the proper way to check for equality. For not−equal−to, you enter `!=`, which magically turns into `≠`.

```
In[49]:=       5 < 10
Out[49]=       True
In[50]:=       5 > 10
Out[50]=       False
In[51]:=       10 < 10
Out[51]=       False
In[52]:=       10 ≤ 10
Out[52]=       True
In[53]:=       Sqrt[2] < 2
Out[53]=       True
In[54]:=       10 == 10
Out[54]=       True
In[55]:=       10 == 9
Out[55]=       False
In[56]:=       10 = 9
From In[56]:=  Set::setraw : Cannot assign to raw object 1
Out[56]=       9
In[57]:=       10 ≠ 9
Out[57]=       True
```

So that gives us the first branch of the decision, that is, when to move right. But we have to handle another decision, which is when to move left. First, you should know that you can make your program more readable by hitting the Enter key and putting in some line breaks:

```
MoveAnt[k_, u_] := If[u < 1 / 3,
  Min[k + 1, 10],
  ...]
```

Now for when to move left:

```
MoveAnt[k_, u_] := If[u < 1 / 3,
  Min[k + 1, 10],
  If[u > 2 / 3,
   Max[k - 1, 1],
   ...]]
```

Notice that *Mathematica* will automatically indent things based on how they're nested. That also makes your program more readable. The last case is that the ant stays still, and we're done:

```
In[58]:= MoveAnt[k_, u_] := If[u < 1 / 3,
           Min[k + 1, 10],
           If[u > 2 / 3,
            Max[k - 1, 1],
            k]]
```

Let's test it out:

```
In[59]:= MoveAnt[3, 1 / 10]

Out[59]= 4

In[60]:= MoveAnt[3, 9 / 10]

Out[60]= 2

In[61]:= MoveAnt[3, 1 / 2]

Out[61]= 3

In[62]:= MoveAnt[10, 1 / 10]

Out[62]= 10

In[63]:= MoveAnt[1, 9 / 10]

Out[63]= 1
```

Uniform random numbers

We want the ant to move at random, so we need to pass our MoveAnt function some random numbers. Luckily, *Mathematica* has a built in function called `Random`.

Every time you call `Random`, it produces a number between 0 and 1 that is for all intents and purposes random. It takes no arguments, just creates a number, and it's different each time. The brackets [] are necessary.

```
In[64]:= Random[]
Out[64]= 0.0305317
In[65]:= Random[]
Out[65]= 0.453709
In[66]:= Random[]
Out[66]= 0.283504
```

So when you evaluate this expression, it simulates the ant moving at random:

```
In[67]:= MoveAnt[5, Random[]]

Out[67]= 4

In[68]:= MoveAnt[5, Random[]]

Out[68]= 5

In[69]:= MoveAnt[5, Random[]]

Out[69]= 5
```

We can define a new function that moves the ant at random:

```
In[70]:= MoveAntRandom[k_] := MoveAnt[k, Random[]]
```

This is a place where you need the :=. Watch what happens if you use =. Essentially what goes wrong, is that the `Random`[] gets evaluated before the definition is made, which means the same *u* will always be used, which makes the ant always move the same direction. And that's not what we want.

```
In[71]:= Wrong[k_] = MoveAnt[k, Random[]]
Out[71]= Max[1, -1 + k]
In[72]:= ?Wrong
         Global`Wrong
         Wrong[k_] = Max[1, -1 + k]
```

Simulating many steps

The real power of a computer is that it can automatically do something over and over without getting bored. So how do we encode repetition? And in particular how do we keep up with the location of the ant?

There are lots of ways to do this. One is with the *Mathematica* function `NestList`.

The `NestList` function takes a function, and a starting point, and a count. It then creates a list consisting of your starting point, then the function applied to the starting point, then the function applied to that, and so on.

```
In[73]:=  NestList[f, x, 3]
Out[73]=  {x, f[x], f[f[x]], f[f[f[x]]]}
In[74]:=  f[x_] = x + 1
Out[74]=  1 + x
In[75]:=  NestList[f, 10, 3]
Out[75]=  {10, 11, 12, 13}
In[76]:=  Clear[f]
```

So let's try this:

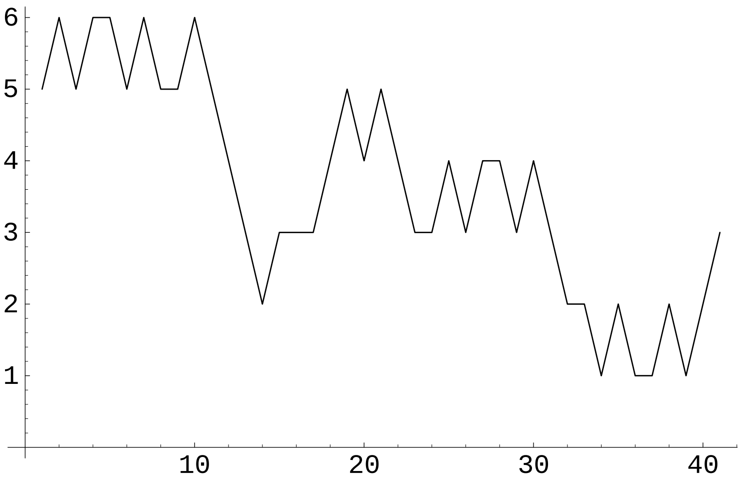
```
In[77]:=  NestList[MoveAntRandom, 5, 10]

Out[77]=  {5, 6, 7, 7, 6, 5, 6, 7, 7, 8, 7}
```

Let's see if we can get a picture. We'll move the any many steps now.

```
In[78]:=  antPath = NestList[MoveAntRandom, 5, 40]

Out[78]=  {5, 6, 5, 6, 6, 5, 6, 5, 5, 6, 5, 4, 3, 2, 3, 3, 3, 4, 5, 4,
           5, 4, 3, 3, 4, 3, 4, 4, 3, 4, 3, 2, 2, 1, 2, 1, 1, 2, 1, 2, 3}
```

The simplest way to get a picture is with `ListPlot`, which has the unfortunate side effect of drawing the picture sideways. I'd really like to see the ant moving left and right, with time going up, but we'll leave that for later.

```
In[79]:=  ListPlot[antPath, PlotJoined → True]
```



```
Out[79]=  - Graphics -
```

Using anonymous functions

Here's a very useful thing to know: You can define anonymous functions in *Mathematica*. That may not seem very useful, after all, how can you use a function if it has no name. But, very often you need to pass a function as an argument to another function, as in `NestList`, and you need to specify some of its arguments.

Here's how to create an anonymous function that takes a number x and returns $x^2 + 1$. The first item is a list of argument names, and the second is an expression for the return value.

```
In[80]:=  Function[{x}, x^2 + 1]
Out[80]=  Function[{x}, x² + 1]
```

Suppose the function you want to iterate takes extra arguments. Then, this is how you can call `NestList` and specify a particular value for that extra argument. You use `Function` to create a function that takes one value, and passes it to h, but the other arguments to h are fixed. This construction is really useful when you want to define a function that uses `NestList` and you want to pass in that extra argument. In this example, hIter takes an initial x, the value of c to use when calling h, and how many steps to iterate. You will write functions like hIter all the time when using functions like `NestList` and `Map`.

```
In[81]:=  h[x_, c_] = x^2 + c
Out[81]=  c + x²
In[82]:=  NestList[Function[{x}, h[x, 1/4]], 0, 3
Out[82]=  {0, 1/4, 5/16, 89/256}
In[83]:=  hIter[xInitial_, c_, nSteps_] :=
           NestList[Function[{x}, h[x, c]], xInit
In[84]:=  hIter[0, 1/4, 3]
Out[84]=  {0, 1/4, 5/16, 89/256}
In[85]:=  Clear[h, hIter]
```

Exercises

Exercise 3

Write a new MoveAnt function that takes two additional arguments, p_{left} and p_{right}, so that the ant moves left with probability p_{left} and right with probability p_{right}.

Exercise 4

Extend your MoveAnt function from the previous problem so that it also takes n, the number of positions our ant can occupy, as an argument.

Exercise 5

Using your MoveAnt function, plot a sample path of an ant that moves left with probability $\frac{1}{2}$, right with probability $\frac{1}{3}$, walking on the numbers 1 to 50. Plot 50 steps. Try to do this using NestList and an anonymous function, as in the example.

■ Second simulation

Let's try something different. Instead of the previous simulation, where the ant moved left or right with some probability, let's make it so that the ant is facing left or right, and at each time step, it either moves forward or turns around. This means that the ant has state in addition to its location, so we have to keep up with more information.

Representing multi−part data (records)

Records are a useful concept that most programming languages support. A record allows you to combine several pieces of information into a single place. A classic example is that a simple address consists of a person's name, a street with a house or building number, a city, a state, and a zip code. To represent an address in a computer, you need to keep up with these five pieces of information, so it's common to use some feature of the language to keep all the parts of an address together.

Records are called "structures" in C and C++ and related languages.

In *Mathematica*, there are a couple of ways to represent a record. The simplest is to take advantage of the fact that if *Mathematica* encounters an expression for which there are no definitions, that expression evaluates to itself. So you can represent an address as an expression (that looks like function application) whose head is "Address" and whose arguments are strings of characters that represent the parts of the address.

```
In[86]:=  Address["Elmo", "123 Sesame St.", "New York", "NY", "10023"]

Out[86]=  Address[Elmo, 123 Sesame St., New York, NY, 10023]
```

Since an ant now has to have a direction as well as a location, we need to use a record with two pieces of information. We can use the symbols Left and Right for the orientation. So this, for example, represents an ant on space number 5, facing left.

```
In[87]:=  Ant[5, Left]
Out[87]=  Ant[5, Left]
```

Now we have to encode the rule for moving an ant: It takes a step with probability 1/2, that is, if the random number u is less than 1/2, otherwise it turns around. Again we have to use `Max` and `Min` to make sure it doesn't step out of bounds.

$$f[\mathrm{Ant}[k, \mathrm{Left}], u] = \begin{cases} \mathrm{Ant}[\mathrm{Max}[1, k-1], \mathrm{Left}] & \text{if } u < \frac{1}{2} \\ \mathrm{Ant}[k, \mathrm{Right}] & \text{otherwise} \end{cases}$$

$$f[\mathrm{Ant}[k, \mathrm{Right}], u] = \begin{cases} \mathrm{Ant}[\mathrm{Min}[10, k+1], \mathrm{Right}] & \text{if } u < \frac{1}{2} \\ \mathrm{Ant}[k, \mathrm{Left}] & \text{otherwise} \end{cases}$$

Again, we'll call the update function MoveAnt because *f* is such an uninformative name. You can see here that we use Left instead of Left_. That's because we want to give a definition that applies when the ant's second item is the symbol Left, so we don't want to use Left as a pattern variable. We also leave the underscore off Right for the same reason.

Also, you can see that we can give *Mathematica* several definitions for MoveAnt. It remembers the pattern we used in each definition, and that's how it keeps them straight.

```
In[88]:=  MoveAnt[Ant[k_, Left], u_] := If[u < 1 / 2
            Ant[Max[1, k - 1], Left],
            Ant[k, Right]]

In[89]:=  MoveAnt[Ant[k_, Right], u_] := If[u < 1 /
            Ant[Min[10, k + 1], Right],
            Ant[k, Left]]
```

Let's test our new definition of MoveAnt:

```
In[90]:=   MoveAnt[Ant[5, Left], 0.1]
Out[90]=   Ant[4, Left]
In[91]:=   MoveAnt[Ant[1, Left], 0.1]
Out[91]=   Ant[1, Left]
In[92]:=   MoveAnt[Ant[8, Right], 0.1]
Out[92]=   Ant[9, Right]
In[93]:=   MoveAnt[Ant[8, Right], 0.9]
Out[93]=   Ant[8, Left]
```

Exercises

Exercise 6

Write a new MoveAnt function that takes two additional arguments, p_{step} and n, so that the ant steps forward with probability p_{step} and turns around with probability $1 - p_{\text{step}}$. The argument n is the number of positions our ant can occupy.

Exercise 7

Write a new definition for MoveAntRandom that takes an ant, p_{step}, and n, and moves the ant at random.

Exercise 8

Create a path with 20 random steps of an ant that moves with probability $\frac{2}{3}$ and turns with probability $\frac{1}{3}$ on the numbers 1 to 15. (Consider using an anonymous function.)

Solution (hidden)

Transforming lists of data

We now have a way to produce a list of all the states our ant has been in. But we can't just plot this list because it's a list of Ant[*k*, direction], not a list of numbers. Let's try to plot a list of just the positions of the ant.

So we'll have something to work with, here's a sample path I created.

```
In[99]:= randomAntPath = {Ant[5, Right], Ant[6, Right],
           Ant[7, Right], Ant[8, Right], Ant[8, Left], Ant[7, Left],
           Ant[6, Left], Ant[6, Right], Ant[6, Left], Ant[6, Right],
           Ant[7, Right], Ant[8, Right], Ant[9, Right], Ant[10, Right],
           Ant[11, Right], Ant[12, Right], Ant[12, Left],
           Ant[11, Left], Ant[10, Left], Ant[9, Left], Ant[8, Left]}
```

```
Out[99]= {Ant[5, Right], Ant[6, Right], Ant[7, Right],
          Ant[8, Right], Ant[8, Left], Ant[7, Left], Ant[6, Left],
          Ant[6, Right], Ant[6, Left], Ant[6, Right], Ant[7, Right],
          Ant[8, Right], Ant[9, Right], Ant[10, Right],
          Ant[11, Right], Ant[12, Right], Ant[12, Left],
          Ant[11, Left], Ant[10, Left], Ant[9, Left], Ant[8, Left]}
```

Usually we don't want to see the long output of a calculation like this. *Mathematica* can be told not to show the output: Just end the command with a semi−colon (;) Later, when we do some very long simulations, you'll definitely want to use the semi−colon. Otherwise your notebook will be full of useless output.

```
In[100]:= randomAntPath = {Ant[5, Right], Ant[6, Right],
            Ant[7, Right], Ant[8, Right], Ant[8, Left], Ant[7, Left],
            Ant[6, Left], Ant[6, Right], Ant[6, Left], Ant[6, Right],
            Ant[7, Right], Ant[8, Right], Ant[9, Right], Ant[10, Right],
            Ant[11, Right], Ant[12, Right], Ant[12, Left],
            Ant[11, Left], Ant[10, Left], Ant[9, Left], Ant[8, Left]};
```

We need a function that will take an ant and give just its position.

```
In[101]:= AntPosition[Ant[k_, dir_]] := k
```

To go from a list of Ant records to a list of positions, we need to apply AntPosition to each entry in the list. The best way to do that is to use the `Map` function. `Map` takes a function and a list, and returns the list you get by applying that function to each item in the list.

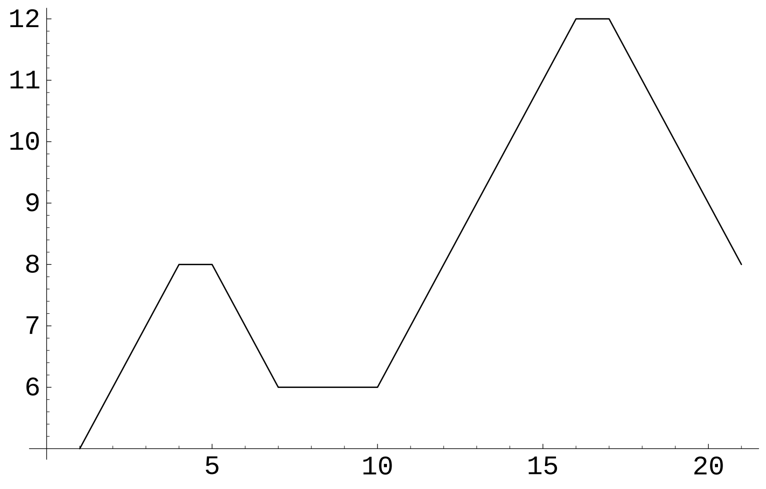
```
In[102]:=  Map[f, {a, b, c}]
Out[102]=  {f[a], f[b], f[c]}
```

So here's how to get just the positions out of our list of ants:

```
In[103]:=  Map[AntPosition, randomAntPath]
Out[103]=  {5, 6, 7, 8, 8, 7, 6, 6, 6, 6, 7, 8, 9, 10
```

And this we know how to plot:

```
In[104]:=  ListPlot[Map[AntPosition, randomAntPath], PlotJoined → True]
```



```
Out[104]=  - Graphics -
```

You'll notice that this ant behaves very differently from our first ant. It tends to walk in one direction for a while, then turn around and walk the other way for a while.

Exercises

Exercise 9

Create a path with 50 random steps of an ant that moves with probability $\frac{2}{3}$ and turns with probability $\frac{1}{3}$ on the numbers 1 to 15. Plot just the positions.

Exercise 10

Write a function AntOrientation that takes an ant and gives just its orientation.

Exercise 11

Using the path you created, find a way to plot the just the ant’s orientation. This means you’ll need to come up with a graphical representation for Left and Right.

■ Third simulation

We’ve seen our ant move around on a line, but what about on a plane? Let’s write a new simulation where the ant is facing some orientation. It steps forward with some probability p_{step}, and turns with probability $p_{\text{turn}} = 1 - p_{\text{step}}$. If the ant decides to turn, it turns left with probability p_{left} and right with probability $p_{\text{right}} = 1 - p_{\text{left}}$. The ant is moving around on an $n \times n$ grid.

As before we need to keep track of the ant’s position and orientation. It’s best to represent the ant’s location as an ordered pair $\{x, y\}$ because this is the form that points should be in for `ListPlot` and other built−in function. The orientation can’t be just left and right anymore. Let’s use the symbols North, South, East, and West. So in this new simulation, ants will be represented as Ant[$\{x, y\}$, direction].

```
In[105]:=  Ant[{3, 7}, South]
Out[105]=  Ant[{3, 7}, South]
```

Exercises

Since this simulation is more complicated, we’ll break up the MoveAnt function into smaller parts. If you try to write the MoveAnt function all at once, you’ll find yourself drowning in all the different cases.

Exercise 12

Write a function TakeStep that takes a point $\{x, y\}$ and a direction North, South, East, or West, and returns the point $\{x_{\text{next}}, y_{\text{next}}\}$ you get by taking a step of size 1 in that direction. (Don’t worry about the boundary. We’ll handle that in the next exercise.)

Exercise 13

Write a function EnforceBoundary that takes a point $\{x, y\}$ and n and returns a point $\{x_{\text{fixed}}, y_{\text{fixed}}\}$. The new point must satisfy $1 \le x_{\text{fixed}} \le n$ and $1 \le y_{\text{fixed}} \le n$. If $\{x, y\}$ is already inside the box, then you shouldn’t change it. But if $\{x, y\}$ is outside the box, you should return the point on the boundary closest to $\{x, y\}$. Your function should satisfy the properties that

EnforceBoundary[$\{n + 1, y\}$, n] = $\{n, y\}$

EnforceBoundary[$\{0, y\}$, n] = $\{1, y\}$

etc.

The idea is that when we write MoveAnt, there will be a case where the ant must take a step. To write that case, we'll first have the ant try to take a step, and use the EnforceBoundary function to make sure it doesn't step outside the box.

Exercise 14

Write a function Turn that takes an orientation (North, South, East, or West), and a direction (Left or Right), and returns the new orientation the ant gets by making a turn in the given direction. For instance, if it's facing north and has to turn right, it will now be facing east.

Exercise 15

Now write the MoveAnt function. It should take an ant, p_{step}, p_{left}, and the box size n. How many random numbers should it take? Be sure to use your TakeStep, EnforceBoundary, and Turn functions, as they will make this function much shorter and easier to write.

Exercise 16

Write the MoveAntRandom function to go with your new MoveAnt function.

Exercise 17

Write a MakeRandomPath function that takes a record representing the initial position and orientation of the ant, p_{step} and p_{left}, the box size n, and the number of random steps for the ant to take. It should return a list of ant records for the history of the ant's steps.

Exercise 18

Make a random path where $p_{\text{step}} = \frac{2}{3}$, $p_{\text{left}} = \frac{1}{2}$, the box size is 100, and the ant takes 10000 steps. (Hint: You *really* don't want to see all the ant records on the screen, so end the command that creates the path with a semi−colon.)

Now plot the path.

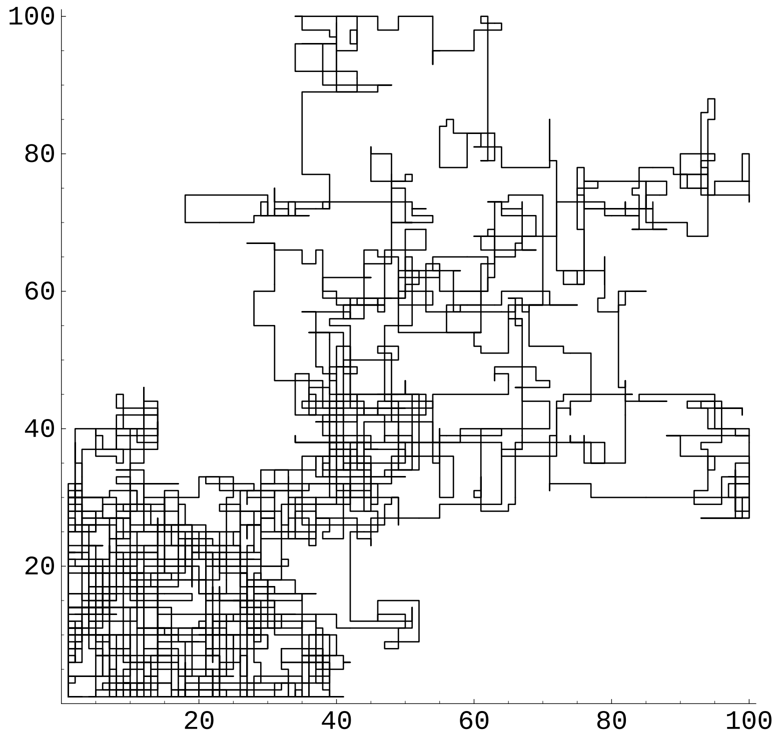
Solution (hidden)

Example random ant path

Here's a picture from my solution. This is pretty ant−like. (Well, ant's don't move on grids, but this is pretty good.) I used the PlotRange option so that the entire box is visible, and that there's a little bit of a gap between the box and the axes and the edges of the picture. I used the AspectRatio option to make the plot square, that way, the grid isn't squished into a TV−shaped rectangle.

```
In[122]:= antPathInPlane =
            MakeRandomPath[Ant[{4, 4}, North], 2 / 3, 1 / 2, 100, 10000];
```

```
In[123]:= ListPlot[Map[AntPosition, antPathInPlane], PlotJoined → True,
            PlotRange → {{0, 101}, {0, 101}}, AspectRatio → 1]
```



```
Out[123]= - Graphics -
```

A stratified population model in discrete time.

■ A model with adults and children

Let's suppose we want to model a population consisting of children and adults, with time being discrete. Each time step will represent a year. So we might say that each year, $\frac{1}{16}$ of all children are old enough to become adults, and each adult has a $\frac{1}{20}$ chance of becoming a parent. To keep things simple, parents will just have one child per year at most. We'll also include mortality: $\frac{1}{50}$ of the children die each year, and $\frac{1}{30}$ of the adults die each year. (*By the way, I'm making these numbers up.) This leads to the following discrete time model:

$x_1[t] =$ the number of children at time t

$x_2[t] =$ the number of adults at time t

$x_1[t+1]$ = (the number from last year) +
(the number of newborns) − (the number that grow up) − (the number that die)

$x_2[t+1]$ = (the number from last year) + (the number of children who grew up) − (the number that die)

$$x_1[t+1] = x_1[t] + \frac{1}{20} x_2[t] - \frac{1}{16} x_1[t] - \frac{1}{50} x_1[t]$$
$$x_2[t+1] = x_2[t] + \frac{1}{16} x_1[t] - \frac{1}{30} x_2[t]$$

You can represent this model in terms of matrices and vectors. We'll represent the population as a vector:

$$x[t] = \begin{pmatrix} x_1[t] \\ x_2[t] \end{pmatrix}$$

The transition from one year to the next is a linear transformation of x_1 and x_2 so we can represent it as a vector:

$$M = \begin{pmatrix} 1 - \frac{1}{16} - \frac{1}{50} & \frac{1}{20} \\ \frac{1}{16} & 1 - \frac{1}{30} \end{pmatrix}$$

In *Mathematica*, vectors are represented as lists with two items. Matrices are represented as lists of lists, where the entries in the matrix are grouped into rows. The special `MatrixForm` function displays a matrix nicely laid out.

```
In[124]:=  A = {{a, b}, {c, d}}

Out[124]=  {{a, b}, {c, d}}


In[125]:=  MatrixForm[A]

    Out[125]//MatrixForm=
```

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix}$$

Thus, to define the linear transformation for going from one year to the next, we have two options. One is to define M as a list of lists, and the other is to use the matrix button on the Basic Input palette. When you click it, it inserts a matrix with four open slots:

$$\begin{pmatrix} \square & \square \\ \square & \square \end{pmatrix}$$

You must replace the empty boxes with expressions. You can also go to the Input menu, and choose Insert Table/Matrix/Palette. That opens up a dialog box where you can specify the number of rows and columns.

In the end, either way produces the same definition of M.

```
In[126]:=  M = {{1 - 1 / 16 - 1 / 50, 1 / 20}, {1 / 16, 1 - 1 / 30}}
```

Out[126]= $\left\{\left\{\frac{367}{400}, \frac{1}{20}\right\}, \left\{\frac{1}{16}, \frac{29}{30}\right\}\right\}$

```
In[127]:=  MatrixForm[M]
```

Out[127]//MatrixForm=

$$\begin{pmatrix} \frac{367}{400} & \frac{1}{20} \\ \frac{1}{16} & \frac{29}{30} \end{pmatrix}$$

In[128]:= MUsingPalette = $\begin{pmatrix} 1 - 1/16 - 1/50 & 1/20 \\ 1/16 & 1 - 1/30 \end{pmatrix}$

Out[128]= $\left\{\left\{\frac{367}{400}, \frac{1}{20}\right\}, \left\{\frac{1}{16}, \frac{29}{30}\right\}\right\}$

The transition rule can now be represented as matrix multiplication. In *Mathematica*, you have to be careful about multiplication and lists. If you just put two terms for a matrix product next to each other, it treats them as lists rather than matrices and you get strange results. This example shows that we get something horribly wrong from the expression $A\,x$. It's not even a column, which is what we wanted. Instead, matrix multiplication is handled with the dot operator that is, just a period (.) So the expression $A\,.\,x$ gives us what we want.

```
In[129]:=  x = {x1, x2}

Out[129]=  {x1, x2}

In[130]:=  MatrixForm[x]
```

Out[130]//MatrixForm=

$$\begin{pmatrix} x1 \\ x2 \end{pmatrix}$$

```
In[131]:=  A x

Out[131]=  {{a x1, b x1}, {c x2, d x2}}

In[132]:=  MatrixForm[A x]
```

Out[132]//MatrixForm=

$$\begin{pmatrix} a\,x1 & b\,x1 \\ c\,x2 & d\,x2 \end{pmatrix}$$

```
In[133]:= A.x

Out[133]= {a x1 + b x2, c x1 + d x2}
```

```
In[134]:= MatrixForm[A.x]

Out[134]//MatrixForm=
```

$$\begin{pmatrix} a\,x1 + b\,x2 \\ c\,x1 + d\,x2 \end{pmatrix}$$

Let's say that we start with a population of 20 adults and 5 children. What does the population look like the next year?

```
In[135]:= M.{5, 20}
```

Out[135]= $\left\{\frac{447}{80}, \frac{943}{48}\right\}$

Hmm. Now is a good time to introduce the *N* function, which evaluates an expression down to decimal numbers and gets rid of all the fractions.

```
In[136]:= N[M.{5, 20}]

Out[136]= {5.5875, 19.6458}
```

What if we wanted the state after two years? We just apply the transition matrix twice:

```
In[137]:= N[M.M.{5, 20}]

Out[137]= {6.10882, 19.3402}
```

What about after 50 years? We need to take M^{50} but the power operator (^) doesn't work like we want: It returns a matrix consisting of powers of the entries. Instead, we have to use the `MatrixPower` function.

```
In[138]:= MatrixForm[A^2]

Out[138]//MatrixForm=
```

$$\begin{pmatrix} a^2 & b^2 \\ c^2 & d^2 \end{pmatrix}$$

```
In[139]:= MatrixForm[A.A]

Out[139]//MatrixForm=
```

$$\begin{pmatrix} a^2 + b\,c & a\,b + b\,d \\ a\,c + c\,d & b\,c + d^2 \end{pmatrix}$$

```
In[140]:= MatrixForm[MatrixPower[A, 2]]

Out[140]//MatrixForm=
```

$$\begin{pmatrix} a^2 + b\,c & a\,b + b\,d \\ a\,c + c\,d & b\,c + d^2 \end{pmatrix}$$

Back to our example, if we want to see the population in 50 years, this is what we should do.

```
In[141]:= N[MatrixPower[M, 50].{5, 20}]

Out[141]= {11.3224, 19.416}
```

Let’s say we want to plot the trajectory of the population over time. We’d like the number of children as a function of time, and the number of adults as a function of time.

First, we’ll generate a list of population states. Let’s start by doing this with the `Table` function. Table takes an expression and a range for a variable, then produces a list of all values of that expression for different values of the variable. The range is specified as {variable, first, last}. You can also specify {variable, last} to mean that the first is 1, and {variable, first, last, size} to mean that it should take steps of the given size, rather than just 1.

```
In[142]:= Table[f[j], {j, 10}]

Out[142]= {f[1], f[2], f[3], f[4], f[5], f[6], f[

In[143]:= Table[f[j], {j, 2, 10}]

Out[143]= {f[2], f[3], f[4], f[5], f[6], f[7], f[

In[144]:= Table[f[j], {j, 0, 10, 2}]

Out[144]= {f[0], f[2], f[4], f[6], f[8], f[10]}
```

So to get our list of states, we want a list of $M^t\ x_0$ where t goes from 0 to 50. But we’d like to eventually plot x and y as functions of time, so rather than just the value of $M^t\ x_0$, we’ll make the entries in our table pairs of $\{t,\ M^t\ x_0\}$. It’s a good idea to try out functions with a small number of iterations to be sure you’re getting it right, then do a bigger calculation and use a semi−colon to keep it from being displayed.

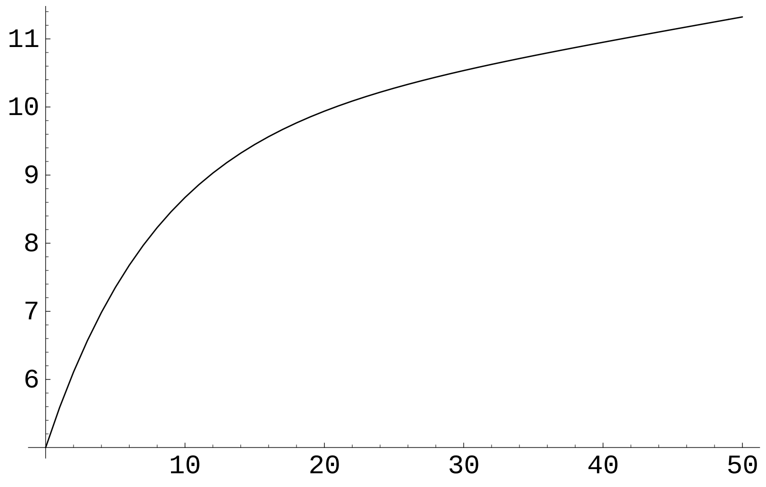
```
In[145]:= x0 = {5, 20}

Out[145]= {5, 20}


In[146]:= Table[N[MatrixPower[M, t].x0], {t, 0, 10}]

Out[146]= {{5., 20.}, {5.5875, 19.6458}, {6.10882, 19.3402},
           {6.57185, 19.0773}, {6.98354, 18.8521},
           {7.35001, 18.6602}, {7.67664, 18.4976}, {7.9682, 18.3608},
           {8.22886, 18.2468}, {8.46232, 18.1529}, {8.67182, 18.0767}}


In[147]:= Table[{t, N[MatrixPower[M, t].x0]}, {t, 0, 10}]

Out[147]= {{0, {5., 20.}}, {1, {5.5875, 19.6458}}, {2, {6.10882, 19.3402}},
           {3, {6.57185, 19.0773}}, {4, {6.98354, 18.8521}},
           {5, {7.35001, 18.6602}}, {6, {7.67664, 18.4976}},
           {7, {7.9682, 18.3608}}, {8, {8.22886, 18.2468}},
           {9, {8.46232, 18.1529}}, {10, {8.67182, 18.0767}}}
```

```
In[148]:= populationStates =
           Table[{t, N[MatrixPower[M, t].x0]}, {t, 0, 50}];
```

Now that we have the states, how do we get a list of $\{t, \ x_1[t]\}$ so we can plot the number of children? We'll write a function that takes a data point $\{t, \{x_1[t], x_2[t]\}\}$ and gives $\{t, x_1[t]\}$. Then we can use the `Map` function to apply it to our list of population states.

```
In[149]:= TimeAndNumChildren[{t_, {x1_, x2_}}] = {t, x1}

Out[149]= {t, x1}


In[150]:= TimeAndNumChildren[{0, {x1[0], x2[0]}}]

Out[150]= {0, x1[0]}


In[151]:= childrenInTime = Map[TimeAndNumChildren, populationStates]

Out[151]= {{0, 5.}, {1, 5.5875}, {2, 6.10882}, {3, 6.57185},
           {4, 6.98354}, {5, 7.35001}, {6, 7.67664}, {7, 7.9682},
           {8, 8.22886}, {9, 8.46232}, {10, 8.67182}, {11, 8.86023},
           {12, 9.03006}, {13, 9.18355}, {14, 9.32264},
           {15, 9.44907}, {16, 9.56434}, {17, 9.66981}, {18, 9.76665},
           {19, 9.8559}, {20, 9.93847}, {21, 10.0152}, {22, 10.0867},
           {23, 10.1537}, {24, 10.2168}, {25, 10.2763}, {26, 10.3328},
           {27, 10.3866}, {28, 10.438}, {29, 10.4874}, {30, 10.5349},
           {31, 10.5809}, {32, 10.6255}, {33, 10.6689}, {34, 10.7113},
           {35, 10.7528}, {36, 10.7934}, {37, 10.8334}, {38, 10.8728},
           {39, 10.9118}, {40, 10.9502}, {41, 10.9883}, {42, 11.0262},
           {43, 11.0637}, {44, 11.101}, {45, 11.1382}, {46, 11.1752},
           {47, 11.2121}, {48, 11.2489}, {49, 11.2857}, {50, 11.3224}}
```

In[152]:=
```
ListPlot[childrenInTime, PlotJoined → True]
```



Out[152]=
```
- Graphics -
```

Exercises

Exercise 19

Plot the number of adults as a function of time.

Exercise 20

Make a picture with two curves, one for the number of children as a function of time, and one for the number of adults.

Eigenvalue solution

Computing powers of a matrix is much easier if we put the matrix into Jordan canonical form (that is, diagonalize it). That’s because powers of a diagonal matrix are easy to take:

In[153]:=
```
MatrixForm[MatrixPower[(a 0
                        0 b), n]]
```

Out[153]//MatrixForm=

$$\begin{pmatrix} a^n & 0 \\ 0 & b^n \end{pmatrix}$$

And with diagonalization, the following happens:

$$A = P\,J\,P^{-1}$$
$$A^n = (P\,J\,P^{-1})\,(P\,J\,P^{-1})\,...\,(P\,J\,P^{-1})$$
$$A^n = P\,J^n\,P^{-1}$$

To compute the Jordan canonical form $A = P\,J\,P^{-1}$, we need the eigenvalues and eigenvectors of the matrix A. Then we stack the eigenvectors to create P and make a diagonal matrix out of the eigenvalues for J.

$$A\,v_1 = \lambda_1\,v_1$$
$$A\,v_2 = \lambda_2\,v_2$$
$$A \begin{pmatrix} \uparrow & \uparrow \\ v_1 & v_2 \\ \downarrow & \downarrow \end{pmatrix} = \begin{pmatrix} \uparrow & \uparrow \\ v_1 & v_2 \\ \downarrow & \downarrow \end{pmatrix} \begin{pmatrix} \lambda_1 & 0 \\ 0 & \lambda_2 \end{pmatrix}$$

Here are some functions you might find useful. *Mathematica* can compute eigenvalues and eigenvectors through the functions `Eigenvalues` and `Eigenvectors`. The function that computes both is `Eigensystem`. There's also a `JordanDecomposition` function that assembles the eigenvectors into the matrices P and J.

In[154]:= **`{{lambda1, lambda2}, {v1, v2}} = Eigensystem[M]`**

Out[154]=
$$\left\{\left\{\frac{2261+\sqrt{21481}}{2400}, \frac{2261-\sqrt{21481}}{2400}\right\}, \left\{\left\{-\frac{232}{15}+\frac{1}{150}\left(2261+\sqrt{21481}\right), 1\right\}, \left\{-\frac{232}{15}+\frac{1}{150}\left(2261-\sqrt{21481}\right), 1\right\}\right\}\right\}$$

In[155]:= **`MatrixForm[Simplify[M.v1]]`**

Out[155]//MatrixForm=
$$\begin{pmatrix} \frac{-18653+367\sqrt{21481}}{60000} \\ \frac{2261+\sqrt{21481}}{2400} \end{pmatrix}$$

In[156]:= **`MatrixForm[Simplify[lambda1 v1]]`**

Out[156]//MatrixForm=
$$\begin{pmatrix} \frac{-18653+367\sqrt{21481}}{60000} \\ \frac{2261+\sqrt{21481}}{2400} \end{pmatrix}$$

In[157]:= **`{P, J} = JordanDecomposition[M]`**

Out[157]=
$$\left\{\left\{\left\{\frac{1}{150}\left(-59-\sqrt{21481}\right), \frac{1}{150}\left(-59+\sqrt{21481}\right)\right\}, \{1, 1\}\right\}, \left\{\left\{\frac{2261-\sqrt{21481}}{2400}, 0\right\}, \left\{0, \frac{2261+\sqrt{21481}}{2400}\right\}\right\}\right\}$$

```
In[158]:= MatrixForm[P]
```

Out[158]//MatrixForm=

$$\begin{pmatrix} \frac{1}{150}\left(-59-\sqrt{21481}\right) & \frac{1}{150}\left(-59+\sqrt{21481}\right) \\ 1 & 1 \end{pmatrix}$$

```
In[159]:= MatrixForm[J]
```

Out[159]//MatrixForm=

$$\begin{pmatrix} \frac{2261-\sqrt{21481}}{2400} & 0 \\ 0 & \frac{2261+\sqrt{21481}}{2400} \end{pmatrix}$$

Just to check that everything's right:

```
In[160]:= MatrixForm[M]
```

Out[160]//MatrixForm=

$$\begin{pmatrix} \frac{367}{400} & \frac{1}{20} \\ \frac{1}{16} & \frac{29}{30} \end{pmatrix}$$

```
In[161]:= MatrixForm[Simplify[P.J.Inverse[P]]]
```

Out[161]//MatrixForm=

$$\begin{pmatrix} \frac{367}{400} & \frac{1}{20} \\ \frac{1}{16} & \frac{29}{30} \end{pmatrix}$$

Suppose we write our population state as $x = a\,v_1 + b\,v_2$. Then $M\,x = a\,M\,v_1 + b\,M\,v_2 = a\,\lambda_1\,v_1 + b\,\lambda_2\,v_2$. Continuing inductively, we see that:

$$M^t\,x = a\,M^t\,v_1 + b\,M^t\,v_2 = a\,\lambda_1{}^t\,v_1 + b\,\lambda_2{}^t\,v_2$$

Let's see what the numerical values of the eigenvalues are.

```
In[162]:= N[{lambda1, lambda2}]

Out[162]= {1.00315, 0.881015}
```

Since $\lambda_1 > 1$ and $\lambda_2 < 1$, the power $\lambda_2{}^t \to 0$ as $t \to \infty$, which means that the v_2 component of the population state is disappearing. However, the v_1 component of the population is growing. So that means that over time, the population state will look more and more like a multiple of v_1.

Let's see what v_1 looks like numerically. This means that in the limit there will be about 0.6 children for each adult in the population.

```
In[163]:= N[v1]

Out[163]= {0.58376, 1.}
```

Exercises

Suppose we have a population with four age groups: children, adolescents, adults, and elderly. Children are born at a yearly rate of $\frac{1}{50}$ per adolescent (for a few teenage pregnancies), and $\frac{1}{20}$ per adult. Each year, $\frac{1}{14}$ of the children become old enough to become adolescents, $\frac{1}{4}$ of the adolescents become adults, and $\frac{1}{40}$ of the adults become elderly. Each year, $\frac{1}{50}$ of the children die, $\frac{1}{40}$ of the adolescents die, $\frac{1}{50}$ of the adults die, and $\frac{1}{5}$ of the elderly die.

Exercise 21

Starting from an initial population of 20 children, 10 adolescents, 50 adults, and 4 elderly, plot the number of each age group in the population as a function of time for 100 years.

Hint: Instead of writing TimeAndNumChildren with patterns, try using the `Part` function, or the equivalent notation $v[[n]]$. This will allow you to write a single function TimeAndNthGroup, rather than 4 functions TimeAndChildren, TimeAndAdolescents....

```
In[164]:=  z = {1, 2, 3, a, b, c}
Out[164]=  {1, 2, 3, a, b, c}
In[165]:=  Part[z, 3]
Out[165]=  3
In[166]:=  Part[z, 6]
Out[166]=  c
In[167]:=  z[[3]]
Out[167]=  3
In[168]:=  z[[6]]
Out[168]=  c
```

Exercise 22

Using eigenvectors, what proportions will the population tend to as time increases?

Now let’s make the problem a little more interesting. Suppose that with probability p_{epidemic} there’s an epidemic of the flu in any given year. When there’s an epidemic, the death rates of children and elderly are tripled, and the death rates of adolescents and adults are doubled. Now we can’t just use M^t because there’s a different matrix for years when there’s an epidemic, and a year has an epidemic or not based on a random number.

Exercise 23

Simulate the population with $p_{\text{epidemic}} = \frac{1}{10}$.