13.3 Computing Sequence Values

13.3.1 Overview

When working with sequences, we often need to generate many sequence values. It is quite cumbersome to do this by hand. There would be a lot of repetition. Computers should be used to compute.

This section focuses on developing some basic skills in using computers to generate and plot sequences. Spreadsheets are one tool that can be used to compute and plot sequences. A spreadsheet is essentially a blank table; you create computational rules for individual entries within the table. For sequences, those rules correspond to the formulas for generating the sequence.

An alternative to spreadsheets is writing a computational script in a programming language. This might seem intimidating, especially if you have never done any programming. However, many modern scripting languages use commands very similar to the mathematical statements we already use. Scripts have the advantage of reducing the amount of work required to generate data quickly.

13.3.2 Sequences in Spreadsheets

One way to generate a sequence is with a spreadsheet. Common spreadsheet applications include Microsoft Excel, Apple Numbers, and Google Drive Sheets. A spreadsheet essentially starts as a giant blank table with rows and columns. The rows are numbered starting at 1 and the columns are labeled by letters. (After the first 26 columns, columns are labeled by pairs of letters.) Every cell in the table is identified by its column and row, called its **address**. So cell `B4` would be the cell in the fourth row of the second column.

Spreadsheets are designed to perform calculations based on the values of other cells. Suppose that `A1` contained the number 3 and `A2` contained the number 5. If you were to type in cell `A3` the formula `=A1+A2` (including the equal sign), then `A3` would show the value 8. However, it internally remembers the formula. If you were to change the values in either `A1` or `A2`, the value in `A3` would automatically be updated. We can take advantage of these calculations to compute the values of sequences.

For our first example, we look at how to use a spreadsheet to generate a sequence defined explicitly.

Example 13.3.1 Use a spreadsheet to generate values for the sequence defined by

$$x_n = \frac{3n}{4n+5}, \quad n = 0, 1, 2, \ldots.$$

Solution. Our sequence is defined explicitly by the map $n \mapsto x_n$. In our spreadsheet, we will use one column to define a subsequence of the values for the index. Then we will define a second column for the values of the sequence.

We will make the first column, `A`, contain the values for the index. The first entry in the column will be a label. So type `n` in the cell `A1`. Our first value for the index i $n = 0$, so type `0` in cell `A2`. The next value will be $n = 1$, so we type `1` in cell `A3`.

Now, it would be very tedious to type all of the values for the index once entry at a time. We take advantage of technology to have the spreadsheet apply a pattern to complete the rest of the column. Select the two cells we have already created, `A2` and `A3`. You should see a grab box, usually in the bottom right corner. If you drag that box down the column and then let go,

the spreadsheet will follow your pattern for all of the cells you select before releasing. You can make this column of index values as long as you desire, within the limits of the program you use.

We are now ready to create the column containing the values of the sequence. We start with a label for the column in `B1`, for example typing `x_n`. The rest of the column needs to use the map

$$n \mapsto x_n = \frac{3n}{4n+5}.$$

The value of input for this map, the index, is in column `A`. We want the value of the output for the map placed in column `B`. In cell `B2`, we want the output based on an input from `A2`, so we type `=(3*A2)/(4*A2+5)`.

We want to have this process repeated for the rest of the column, but we do not want to type a formula for each cell. We want the software to fill the column automatically. Notice that if you copy the formula from `B2` and paste it into `B3`, the formula is automatically adjusted to refer to `A3` instead of `A2`. This type of automatic modification of a formula is called **relative addressing**, where a pasted formula uses the relative position of a calculation. In this case, the relative position is to use the cell immediately to the left of the output.

Pasting the formula into every cell in the column is faster than typing a formula in every cell, but it is still too much work. We let the spreadsheet do all of the work by *filling* the remaining cells at once. We can do this by repeating the process earlier. Select the cell with a valid formula in `B2`, and you will again see the grab box in the corner. Click on the box and drag down the column. When you release the selection, the formula will be filled into all of the selected cells, adjusted to use relative addresses.

You now have two columns: the index in column `A` and the values in column `B`. You can create a graph by selecting the two columns of data and inserting a scatter plot. □

The second example illustrates how to use spreadsheet to compute the values of a recursively defined sequence.

Example 13.3.2 Use a spreadsheet to generate values for the sequence defined by

$$x_n = 2.3x_{n-1}(1 - 0.1x_{n-1})$$

with an initial value $x_0 = 1$.

Solution. We again want a column for the index values, which for convenience we will put in column `A`. The same steps as in the previous example apply. Put a label `n` in `A1`. Start with values `0` in `A2` and `1` in `A3`. Use the filling tool of the spreadsheet to extend the pattern for as far down the column as you desire.

In the next column `B`, we will put the values of the sequence. Put the label `x_n` in `B1`. Enter the initial value `1` in `B2` since $x_0 = 1$. For the rest of the values in the column, we will use the recursive map,

$$x_{n-1} \mapsto x_n = 2.3x_{n-1}(1 - 0.1x_{n-1}).$$

In the table, the previous value x_{n-1} corresponds to that value in the cell directly above the cell in question. Consequently, to compute x_1, we select `B3` and type `=2.3*B2*(1-0.1*B2)`. The value of x_1 should appear in the cell.

The rest of the column can be calculated by selecting `B3`, where we just typed the formula, and using the applications fill down feature. Notice that if you change the value for the initial condition in `B2`, every subsequent entry in the column is automatically updated. If you create a graph using columns `A` and `B`, the graph also is updated whenever the initial value is updated. □

The final example illustrates using parameters in our calculations.

Example 13.3.3 Use a spreadsheet to generate values for an arbitrary arithmetic sequence defined by

$$x_n = x_{n-1} + \beta$$

with an initial value $x_0 = a$, where a and β are parameters. Include the explicit and recursive calculations side-by-side in the table.

Solution. When we have parameters, we need to use part of our table to enter those values. You could put them anywhere in the table that is convenient. We will put them to the left of the generated table. There are two parameters: β and a. In cell `A1`, type the label `beta`. Then enter a value in the neighboring cell `B1` such as `3` for $\beta = 3$. In cell `A2`, type the label `a`. Then enter a value in the neighboring cell `B2` such as `8` for $a = 8$. The labels are primarily for our convenience to remember the meaning of the parameter values.

The parameters occupy part of the first two columns. You could start the remainder of your table below the parameters, but I find it more convenient to keep values in their own columns. We will skip column `C` to create a gap between our parameters and our sequence table. Column `D` will have the index values, column `E` will have the explicitly computed sequence values, and column `F` will have the recursively computed sequence values. Start by putting the labels in the first row. You might use `explicit` in `E1` and `recursive` in `E2`.

Create the values for the index in column `D` in the same way as described above. The explicit formula for our arithmetic sequence is given by

$$n \mapsto x_n = a + \beta n.$$

In the table, the index n is always found using relative table location. That is, the index used in `E2` will be found to the left in `D2`. The values for the parameters, however, will always be found in the same table positions.

Our parameters need to use **absolute addresses**, which spreadsheet indicate by putting a dollar sign in front of the column and row. To create the formula in `E2`, we will type `$B$1` to represent β and `$B$2` to represent a. This means our spreadsheet entry in `E2` is `$B$2+$B$1*D2`. If you copy and paste this into `E3`, you should see that only the cell representing the index is updated to `$B$2+$B$1*D3`. Selecting one of these cells and filling the rest of the column will finish generating the explicitly calculated values.

To create the column with recursively calculated values, we start by putting the initial value in `F2`. Because that is our parameter a, stored in `B2`, we enter the simple formula `=$B$2` to use that initial value. To apply the recursive relation

$$x_{n-1} \mapsto x_n = x_{n-1} + \beta,$$

we enter `=F2+$B$1` in `F3`. The rest of the column can be automatically filled with this formula.

If you did this correctly, you should see that the explicit and recursive columns contain the same values even though they were calculated in different ways. □

13.3.3 Computer Programming

Spreadsheets are useful, but typing a script can be even more efficient. In a spreadsheet, the use of cell references is a little awkward. There are some advanced techniques where you can reference cells by names instead of reference. However, if you want to adjust the size of your table and change the number of rows, you essentially need to repeat the dragging and filling steps.

Writing computer scripts in a programming language is one of the most efficient approaches. A free online tool called SageMath uses a scripting language based on the Python programming language. The online version of this text has interactive cells where you can try the scripts directly. Otherwise, you can use the following website and type the scripts: https://sagecell.sagemath.org/.

The idea of a scripting language is that the computer will store values in memory associated with names of your choosing. You can includes commands in your script to display the values or even to create graphs. To create a sequence of values, there are two fundamental ideas to understand: memory assignment and looping.

First is the idea of memory assignment. In a script, we tell the computer to perform a computation and then to store the result somewhere in memory. That memory location is assigned a variable name. The pattern for this step is the form `name = calculation`, where `name` is replaced by whatever name you want associated with the memory and `calculation` is replaced by the expression used in the calculation. The calculation can use variable names for any memory previously saved.

Below is a very short script. It will store a value of 3 in memory associated with the name `x`. It will then calculate $x^2 - 5x$ and store the result in memory associated with the name `y`. Finally, it will show the values of x and y as results. You will notice extra lines that begin with the # symbol. These are called comments and the script ignores them. We use comments in scripts to remind ourselves or to explain to others what is happening.

```
# Store the value for x
x = 3
# Calculate x^2-5x and store the value as y
y = x^2 - 5*x
# Show the results
show(x,y)
```

```
3
-6
```

For a sequence, we repeat the same process of calculation many times. Repeating a computation is called a **loop**. In a script, a loop is usually associated with a variable (a named memory location) that is associated with a given list of values. The basic scripting pattern in Python and Sagemath to repeat a computation for each value in a list is as follows.

```
for name in list:
    repeated block
```

The `name` is replaced by whatever variable name you want for the associated memory location. The `list` is replaced by a list of values that will be used for `name`. The `repeated block` is a collection of scripting commands, indented exactly four spaces to the right of the `for` statement. The script will take the values from the `list`, one at a time and in order, and will then go through the `repeated block` immediately after the value has been placed in the memory associated with `name`.

In Python or SageMath, we create a list of consecutive integers using the `range` function. The command `range(integer)`, where `integer` is replaced by any expression representing an integer, creates a list of consecutive integers starting at 0 and ending at the integer before the `integer` used. For example, `range(5)` creates the list $(0, 1, 2, 3, 4)$ and `range(3)` creates the list $(0, 1, 2)$. The following two scripts are equivalent, but the looped version is much more

efficient.

```
# Unlooped
n=0
x=3*n+1
show(n)
show(x)
n=1
x=3*n+1
show(n)
show(x)
n=2
x=3*n+1
show(n)
show(x)
```

```
# Looped
for n in range(3):
    x=3*n+1
    show(n)
    show(x)
```

We are almost ready to create a script that generates a table of values. The last step is creating a table with two values on the same line. We can do this with the `print` command and value formatting. In place of the `show` commands, we will use `print(format % (values))`, with `format` being a format string and `values` being a comma-separated collection of expressions to be formatted. An integer uses format string `%d` while a decimal value (a floating-point) uses format string `%f`. We can include a tab character using `\t`. Our improved script to generate a table with twenty values is now given.

```
# Loop to calculate a table of values.
for n in range(20):
    x=3*n+1
    print("%d\t%f" % (n,x))
```

The script can be quickly modified to generate a table as large or small as desired. If we want to create a graph of these sequence values, we need to create a table in memory. SageMath expects a table to be graphed as a list of points. We can modify our script to create an empty table before the loop, and then add (append) the individual points one at a time in the loop. In SageMath, a graphic is itself an object stored in memory, so we give it a name and then show it. A scatterplot is created using the `list_plot` command, which has an option to label the axes with a given list of names.

```
# Create an empty list named "dataPoints"
dataPoints = []
# Loop to calculate a table of values.
for n in range(20):
    x=3*n+1
    print("%d\t%f" % (n,x))
    # Append the current point to our list.
    # Each point is itself a list with two entries.
    dataPoints.append( [n,x] )
# Create the scatter plot with labels on the axes
myGraph = list_plot(dataPoints, axes_labels=["$n$","$x_n$"])
# Show the resulting figure with given width/height
```

```
show(myGraph, figsize=[4,3])
```

Once you have a script, such as the one above, you can just modify it for a new problem. A table for any explicit sequence can be calculated using the script above simply by modifying the first line in the repeated block to match the explicit formula. More values can be generated by modifying the `range` command. If the table is not wanted, just remove the `print` command.

To create a table for a recursive sequence, we need to make another modification. A recursive sequence uses the previous value of a sequence to compute the next value. In a script, we can use a memory assignment command using the variable name in the expression and then storing the result back into the original memory location. The old value is replaced by the new value.

The following script illustrates how to generate a table and a graph for a recursively defined sequence with recurrence

$$x_n = 1.05x_{n-1} - 10$$

and initial value $x_0 = 400$.

```
# Create an empty list named "dataPoints"
dataPoints = []
# Set the initial value.
# We use the name "x" for the currently-stored sequence value
x = 400
# Loop to calculate a table of values.
for n in range(20):
    # Because the loop starts at n=0, we print and append
        data first.
    print("%d\t%f" % (n,x))
    dataPoints.append( [n,x] )
    # Before we end the repeat block, we update for the next
        value.
    # The formula on the right uses the old value.
    # The answer replaces what was in memory for the next
        loop block
    x=1.05*x-10
# Create the scatter plot with labels on the axes
myGraph = list_plot(dataPoints, axes_labels=["$n$","$x_n$"])
# Show the resulting figure with given width/height
show(myGraph, figsize=[4,3])
```

When we have parameters in a model, we just need to add a few memory assignment commands at the beginning of the script. The following script is a generalization of the previous script for a recursive model

$$x_n = (1 + r)x_{n-1} - w$$

where r and w are parameters. When using variables in scripts, remember that the symbols must exactly match. Uppercase and lowercase letters are not the same—`w` and `W` are different.

```
# Assign parameters
r = 0.05
w = 10
# Create an empty list named "dataPoints"
dataPoints = []
# Set the initial value.
# We use the name "x" for the currently-stored sequence value
```

```
x = 400
# Loop to calculate a table of values.
for n in range(20):
    # Because the loop starts at n=0, we print and append
        data first.
    print("%d\t%f" % (n,x))
    dataPoints.append( [n,x] )
    # Before we end the repeat block, we update for the next
        value.
    # The formula on the right uses the old value.
    # The answer replaces what was in memory for the next
        loop block
    x=(1+r)*x-w
# Create the scatter plot with labels on the axes
myGraph = list_plot(dataPoints, axes_labels=["$n$","$x_n$"])
# Show the resulting figure with given width/height
show(myGraph, figsize=[4,3])
```