COMPUTATIONAL MODELS WITH PYTHON

Models with Arithmetic Growth

José M. Garrido
Department of Computer Science

January 2016

College of Computing and Software Engineering
Kennesaw State University

© 2015 J. M. Garrido
1 Mathematical Modeling

The three types of methods that are used for modeling are:

1. Graphical
2. Numerical
3. Analytical

Graphical methods apply visualization of the data to help understand the data. Various types of graphs can be used; the most common one is the line graph.

Numerical methods directly manipulate the data of the problem to compute various quantities of interest, such as the average change of the population size in a year.

Analytical methods use various forms of relations and equations to allow computation of the various quantities of interest. For example, an equation can be derived that defines how to compute the population size for any given year. Each method has its advantages and limitations. The three methods complement each other and are normally used in modeling.

1.1 Difference Equations

A data list that contains values ordered in some manner is known as a sequence. Typically, a sequence is used to represent ordered values of a property of interest in some real problem. Each of these values corresponds to a recorded measure at a specific point in time. In the example of population change over a period of five years, the ordered list will contain the value of the population size for every year. This type of data is discrete data and the expression for the ordered list can be written as:

\[ \langle p_1, p_2, p_3, p_4, p_5 \rangle \]

In this expression, \( p_1 \) is the value of the population of year 1, \( p_2 \) is the value of the population for year 2, \( p_5 \) is the value of the population for year 5, and so on. In this case, the length of the list is 5 because it has only five values, or terms.

Another example is the study of changes in electric energy price in a year in Georgia, given the average monthly price. Table 1 shows the values of average retail
price of electricity for the state of Georgia.\footnote{U.S. Energy Information Administration — Independent Statistics and Analysis. http://www.eia.gov/} The data given corresponds to the price of electric power that has been recorded every month for the last 12 months. This is another example of \textit{discrete data}. The data list is expressed mathematically as:

\[
\langle e_1, e_2, e_3, e_4, e_5, e_6, e_7, e_8, e_9, e_{10}, e_{11}, e_{12} \rangle
\]

Given the data that has been recorded about a problem and that has been recorded in a list, simple assumptions can be made. One basic assumption is that the quantities in the list increase at a constant rate. This means the increment is assumed to be fixed. If the property is denoted by \( x \), the increment is denoted by \( \Delta x \), and the value of a term measured at a particular point in time is equal to the value of the preceding term and the increment added to it. This can be expressed as:

\[
x_n = x_{n-1} + \Delta x
\]

(1)

Another assumption is that the values of \( x \) are actually always increasing, and not decreasing. This means that the increment is greater than zero, denoted by \( \Delta x \geq 0 \). At any point in time, the increment of \( x \) can be computed as the difference of two consecutive measures of \( x \) and has the value given by the expression:

\[
\Delta x = x_n - x_{n-1}
\]

(2)

These last two mathematical expressions, Equation 1 and Equation 2, are known as \textit{difference equations} and are fundamental for the formulation of simple mathematical models. We can now derive a simple mathematical model for the monthly average price of electric energy, given the collection of monthly recorded energy price in cents per kW-h of the last 12 months. This model is formulated as:

\[
e_n = e_{n-1} + \Delta e
\]

The initial value of energy price, prior to the first month of consumption, is denoted by \( e_0 \), and it normally corresponds to the energy price of a month from the previous year.

1.2 Functional Equations

A functional equation has the general form: \( y = f(x) \), where \( x \) is the independent variable, because for every value of \( x \), the function gives a corresponding value for \( y \). In this case, \( y \) is a function of \( x \).

An equation that gives the value of \( x_n \) at a particular point in time denoted by \( n \), without using the previous value, \( x_{n-1} \), is known as a functional equation. In this case the functional equation can also be expressed as: \( x = f(n) \). From the data given about a problem and from the difference equation(s), a functional equation can be derived. Using analytical methods, the following mathematical expression can be derived and is an example of a functional equation.

\[
x_n = (n - 1) \Delta x + x_1
\]  

This equation gives the value of the element \( x_n \) as a function of \( n \). In other words, the value \( x_n \) can be computed given the value of \( n \). The value of \( \Delta x \) has already been computed. The initial value of variable \( x \) is denoted by \( x_1 \) and is given in the problem by the first element in the sequence \( x \).

2 Models with Arithmetic Growth

Arithmetic growth models are the simplest type of mathematical models. These models have a linear relationship in the variable because the values of the variable increase by equal amounts over equal time intervals. Using \( x \) as the variable, the increase is represented by \( \Delta x \), and the difference equation defined in Equation 1:

\[
x_n = x_{n-1} + \Delta x
\]

Examples are time dependent models, in which a selected property is represented by a variable that changes over time.

From the given data of a real problem, to decide if the model of the problem will exhibit arithmetic growth, the given data must be processed by computing the differences of all consecutive values in the data. The differences thus consist of another list of values. If the differences are all equal, then the model exhibits arithmetic growth, and therefore it is a linear model.

Equation 1 and the simplifying assumption of constant growth can be applied to a wide variety of real problems, such as: population growth, monthly price changes of electric energy, yearly oil consumption, and spread of disease.

Using graphical methods, a line chart or a bar chart can be constructed to produce a visual representation of the changes in time of the variable \( x \). Using numerical
methods, given the initial value $x_0$ and once the increase $\Delta x$ in the property $x$ has been derived, successive values of $x$ can be calculated using Equation 1. As mentioned before, with analytical methods, a functional equation can be derived that would allow the direct calculation of variable $x_n$ at any of the $n$ points in time that are included in the data list given. This equation is defined in Equation 3:

$$x_n = x_1 + \Delta x (n - 1)$$

3 Using the Python Language and NumPy

The *NumPy* library includes functions two of which that are needed to implement computational models with arithmetic growth and these functions are: *diff* and *linspace*. The first function, *diff* computes the differences of a vector (sequence) of data values given.

Calling function *diff* requires one argument, which is the specified vector. A second argument is optional and it specifies order of the differences of the array. The function produces another array that has the values of the differences of the values in the given vector.

```python
>>> import numpy as np
>>> e = np.array ([1.25, 2.15, 4.55, 3.2, 1.05, 2.45, 3.85, 1.15, 2.75, 3.55])
>>> e
array([ 1.25, 2.15, 4.55, 3.2 , 1.05, 2.45, 3.85, 1.15, 2.75, 3.55])
>>> de = np.diff(e)
>>> de
array([ 0.9 , 2.4 , -1.35, -2.15, 1.4 , 1.4 , -2.7 , 1.6 , 0.8 ])
```

### Table 1: Average price of electricity (cents per kW-h) in 2010.

<table>
<thead>
<tr>
<th>Month</th>
<th>Jan</th>
<th>Feb</th>
<th>Mar</th>
<th>Apr</th>
<th>May</th>
<th>Jun</th>
<th>Jul</th>
<th>Aug</th>
</tr>
</thead>
<tbody>
<tr>
<td>Price</td>
<td>10.22</td>
<td>10.36</td>
<td>10.49</td>
<td>10.60</td>
<td>10.68</td>
<td>10.80</td>
<td>10.88</td>
<td>10.94</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Month</th>
<th>Sep</th>
<th>Oct</th>
<th>Nov</th>
<th>Dec</th>
</tr>
</thead>
<tbody>
<tr>
<td>Price</td>
<td>11.05</td>
<td>11.15</td>
<td>11.26</td>
<td>11.40</td>
</tr>
</tbody>
</table>

Listing 1 shows the Python program that computes the differences and the approximate price of electricity for 12 months. The array for the monthly price is
denoted by $e$, and the array for months is denoted by $m$. The array of differences is denoted by $de$ and is computed in line 17 of the program. The data is taken from Table 1.

In the program, the number of measurements is the total number of months, denoted by $N$ and has a value of 12. To compute the average value of the increments in price of electric energy in the year we can use the general expression for calculating average:

$$\Delta e = \frac{1}{n} \sum_{i=1}^{n} de_i$$

Computing the average is implemented in Python with the *mean* NumPy function that computes the average of the values in a vector. Using $d$ to denote the average increment ($\Delta e$), the following statement illustrates the call to function *mean* and is included in line 20 of the Python program.

```
deltax = np.mean(de)
```

Listing 1: Program for computing the differences in price of electricity.

```
1 # Program: priceelect.py
2 # Python program for computing monthly price for electric energy
4
5 import numpy as np
6
7 N = 12;   # 12 months
8 e = np.array([10.22, 10.36, 10.49, 10.60, 10.68, 10.80,
                10.88, 10.94, 11.05, 11.15, 11.26, 11.40])
9 mm = np.arange(N) # month array
10 m = mm + 1
11 print "Monthly price of electricity\n"
12
13 # Array Monthly price for electric energy
14 print e
15 # differences in sequence e
16 print "\nDifferences of the given data\n"
17 de = np.diff(e)
18 print de
19 # average of increments
20 deltax = np.mean(de)
```

© 2015  J. M. Garrido
With the value of average increment of price of electricity, the functional equation of the model is applied to compute the price of electric energy for any month. The new array, $ce$, is defined and contains all the values computed using the functional equation and the average value of the increments. This is shown in line 23 of the program. The following listing shows the command that starts the Python interpreter and the results produced.

```
$ python priceelect.py
Monthly price of electricity
[ 10.22 10.36 10.49 10.6 10.68 10.8 10.88 10.94 11.05 11.15 11.26 11.4 ]
Differences of the given data
[ 0.14 0.13 0.11 0.08 0.12 0.08 0.06 0.11 0.1 0.11 0.14]
Average difference: 0.107272727273
Calculated prices of electricity:
[ 10.22 10.32727273 10.43454545 10.54181818
  10.64909091 10.75636364 10.86363636 10.97090909
  11.07818182 11.18545455 11.29272727 11.4 ]
Data for Plotting
1 10.22 10.22
2 10.36 10.3272727273
3 10.49 10.4345454545
4 10.6 10.5418181818
5 10.68 10.6490909091
6 10.8 10.7563636364
```
4 Producing the Charts of the Model

Two lists of values for the price of electric energy are available as arrays. The first one is the data given with the problem and is denoted by $e$, the second list was computed in the program using the functional equation for all 12 months, and the list is denoted by $ce$. The array with data representing the months of the year is denoted by $m$.

*Gnuplot* is a software tool and is used to produce charts with these three data lists or arrays. Two line charts are generated on the same plot, one line chart of array $e$ with array $m$, the second chart using array $ce$ and array $m$. The Gnuplot commands that create and draw the charts are stored in the script file: *priceelect.cgp* and are shown as follows:

```plaintext
set title "Plot of Montly Price of Electricity vs time"
set xlabel "Time (secs)"
set ylabel "Monthly price of electricity"
set size 1.0, 1.0
set samples 12
plot "priceelect.gpl" u 1:2 with linespoints, "priceelect.gpl"
    u 1:3 with linespoints
```

Figure 1 shows the line chart with the original data given in Table 1 and the line with computed values of the monthly price of electricity.

5 Validation of a Model

Validation of a model is the analysis that compares the values computed with the model with the actual values given. For example, starting with the first value of the monthly consumption of electric energy, the model is used to compute the rest of the monthly values of consumption. These values can then be compared to the values given.

If the corresponding values are close enough, the model is considered a reasonable approximation to the real system.
Figure 1: Given and computed values of monthly price of electricity.

6 File I/O

Computational models typically deal with a large amount of data, which is conveniently stored in data files. The programs discussed and shown previously read and write data to the console, which on a personal computer is the keyboard and the screen, this is also known as standard input and output.

Python provides basic functions and methods necessary to manipulate files. The programs most of the file manipulation using file objects.

6.1 Types of Files

There are two general types of file: text or binary. A text file is typically structured as a sequence of text lines, each being a sequence of characters.

A text line is terminated by a special control character, the EOL (End Of Line) character. The most common line terminator is the \n, or the newline character. The backslash character is used to specify control characters and indicates that the next character will be treated as a newline.

A binary file is basically any file that is not a text file. The main advantage of text files is that no data conversion is necessary.
6.2 Opening and Closing Text Files

A file has to be opened before data can be read or written to it. Opening a file creates a file object and is carried out by calling Python’s built-in `open` function, which returns a file object. The first argument in the call is a string with the file name, the second argument is the access mode for the file, the third argument is optional and indicates the buffering as an integer. The following table indicate the access modes allowed for files in Python.

<table>
<thead>
<tr>
<th>Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>r</td>
<td>Opens a file for reading only. The file pointer is placed at the beginning of the file. This is the default mode.</td>
</tr>
<tr>
<td>rb</td>
<td>Opens a file for reading only in binary format. The file pointer is placed at the beginning of the file. This is the default mode.</td>
</tr>
<tr>
<td>r+</td>
<td>Opens a file for both reading and writing. The file pointer will be at the beginning of the file.</td>
</tr>
<tr>
<td>rb+</td>
<td>Opens a file for both reading and writing in binary format. The file pointer will be at the beginning of the file.</td>
</tr>
<tr>
<td>w</td>
<td>Opens a file for writing only. Overwrites the file if the file exists. If the file does not exist, creates a new file for writing.</td>
</tr>
<tr>
<td>wb</td>
<td>Opens a file for writing only in binary format. Overwrites the file if the file exists. If the file does not exist, creates a new file for writing.</td>
</tr>
<tr>
<td>w+</td>
<td>Opens a file for both writing and reading. Overwrites the existing file if the file exists. If the file does not exist, creates a new file for reading and writing.</td>
</tr>
<tr>
<td>wb+</td>
<td>Opens a file for both writing and reading in binary format. Overwrites the existing file if the file exists. If the file does not exist, creates a new file for reading and writing.</td>
</tr>
</tbody>
</table>

© 2015 J. M. Garrido
### Mode Description

- **a**: Opens a file for appending. The file pointer is at the end of the file if the file exists. That is, the file is in the append mode. If the file does not exist, it creates a new file for writing.

- **ab**: Opens a file for appending in binary format. The file pointer is at the end of the file if the file exists. That is, the file is in the append mode. If the file does not exist, it creates a new file for writing.

- **a+**: Opens a file for both appending and reading. The file pointer is at the end of the file if the file exists. The file opens in the append mode. If the file does not exist, it creates a new file for reading and writing.

- **ab+**: Opens a file for both appending and reading in binary format. The file pointer is at the end of the file if the file exists. The file opens in the append mode. If the file does not exist, it creates a new file for reading and writing.

The following example opens two files: `lengthpart.dat` is opened for writing and the file object created is `outfile`. The second file, `mydata.dat`, is opened for reading and the file object created is `infile`.

```python
outfile = open("lengthpart.dat", "w")
infile = open("mydata.dat", "r")
```

After a file has been used in a program, it should be closed. Method `close` of a file object flushes any unwritten data and closes the file object, after which no more writing can be performed. The following example closes the two files that were opened previously.

```python
outfile.close()
infile.close()
```

### 6.3 Writing Data to a File

Writing data to a file involves calling method `write` of the file object and it writes a string to the open file. Method `write` does not add a newline character (`\n`) to the end of the string.

The following example is a short Python program that opens file `mtest.dat` for writing. In line 3, the string "Testing output data" is written to the file. In line 4 defines variable `v1` with a value of 56. This value has to be converted to a string before it is written to the file. Function `str` is called with variable `v1` as the
The new string variable \texttt{v1str} is created. The statement in line 6 writes the string variable to the file. In a similar manner, \texttt{v2} is a floating-point variable and its value is also converted to a string then written to the file in lines 8–9. The file is closed in line 12.

```python
1 # writing to a data file
2 mfile = open("mtest.dat", 'w')
3 mfile.write("Testing output data\n")
4 v1 = 56
5 v1str = str(v1)
6 mfile.write(v1str)
7 v2 = 10.45
8 v2str = str(v2)
9 mfile.write("\n")
10 mfile.write(v2str + "\n")
11 mfile.write(v1str + " + v2str +"\n")
12 mfile.close()
```

The following lines are stored in file \textit{mtest.dat} when the Python interpreter runs the program.

```
Testing output data
56
10.45
56 10.45
```

### 6.4 Reading Data from a File

As mentioned previously, a text file stores data in string form. Reading data from a text file involves reading lines of text from a file. Method \texttt{read} reads a string from an open file. This method starts reads from the beginning of the file and tries to read as much as possible, or until the end of file. An optional argument can be used that indicates the number of bytes to read.

Method \texttt{readline} reads a single line from the file and returns a string containing characters up to \texttt{\n}. This method is useful for reading data from a file line by line rather than reading the entire file in at once.

```python
infile = open('mydata.txt', 'r')
# Read a line from the open file
line = infile.readline()
```
Because several data values can be contained in a line, the `split` string method can be used to separate substrings in the line that separated by a space character. This method returns a list of substrings. An optional argument is the actual separator character to apply. The following example illustrates the use of the `split` string method. It is first called with no arguments, so the default separator is one or more space characters and the list created is `cols`. The second time it is called with a comma as the argument to be used as the substring separator.

```python
>>> line = "1234 23.59 part 27134"
>>> cols = line.split()
>>> cols
['1234', '23.59', 'part', '27134']
>>> line2 = "1234, 23.59, part 27134"
>>> cols2 = line2.split(",")
>>> cols2
['1234', '23.59', 'part 27134']
```

To read multiple lines from a file, a loop can be used over the file object. This is memory efficient, fast, and leads to simple code. For numerical variables either integer or floating-point, their values are converted from string to the numeric type using functions `float` and `int`. The following short Python program illustrates the general technique of reading lines of data, separating the various data fields from the line, converting each data field to the required type.

```python
1 # Open input file program: gen_filein.py
2 infile = open('mydata.dat', 'r')
3
4 # Read and ignore a header line
5 headstr = infile.readline()
6
7 # Loop over lines and extract variables
8 for line in infile:
9    line = line.strip()  # remove \n char
10   columns = line.split()  # split line into columns
11   print columns
12   var1 = columns[0]
13   var2 = float(columns[1])
14   j = int(columns[2])
15   print var1, var2, j
```

The following data file `mydata.dat` is used for testing the program.
Testing data
sequence1 34.56 88
sequence2 10.45 79
sequence3 85.56 45

Starting the Python interpreter to run the program, produces the following output:

$ python gen_filein.py
['sequence1', '34.56', '88']
sequence1 34.56 88
['sequence2', '10.45', '79']
sequence2 10.45 79
['sequence3', '85.56', '45']
sequence3 85.56 45