DataFile Tutorial

This tutorial will show you how to use the “DataFile” component in Introspect IESP. The DataFile component provides an easy way to access files with floating-point data – e.g. the data measured by some other Test or some external measurement. It handles all the file reading/writing transparently behind the scenes.

Reading Files

Let’s start with a simple example. Suppose you have a text file named “measurements.txt” with the data from some external measurement. Suppose that this file has two floating-point numbers per line, separated by a space or tab. You can create your own data file for use in this tutorial, or use the one that is in the same folder as this document.

Create a new Test in Introspect ESP and then remove the “globalClockConfig” component if one has automatically been created. (We will not make use of the IESP hardware at all – this allows you to do this tutorial without access to the hardware.) Save this Test as “DataFileTutorial”.

Then use the “Add” button to add an instance of “DataFile”.

Set the “fileName” property of dataFile1 to “measurements.txt” (without the quotes). Change the “numFields” property to 2 (since there are 2 values per line of the file).

The “parentFolder” property is “Params” indicating that the data file is expected to be in the “Params” sub-folder of the Test folder. So use Windows Explorer to copy the data file to the “Params” sub-folder of this Test. (An alternative is to leave the data file where it is and change the “parentFolder” property to “Other” and set the “otherFolderPath” property to the full path to the data file.)

Add the following lines into the Test Procedure:

```
for dataEntry in dataFile1:
      print "(%g, %g)" % (dataEntry[0], dataEntry[1])
```

Your Test should now look like Figure 1.

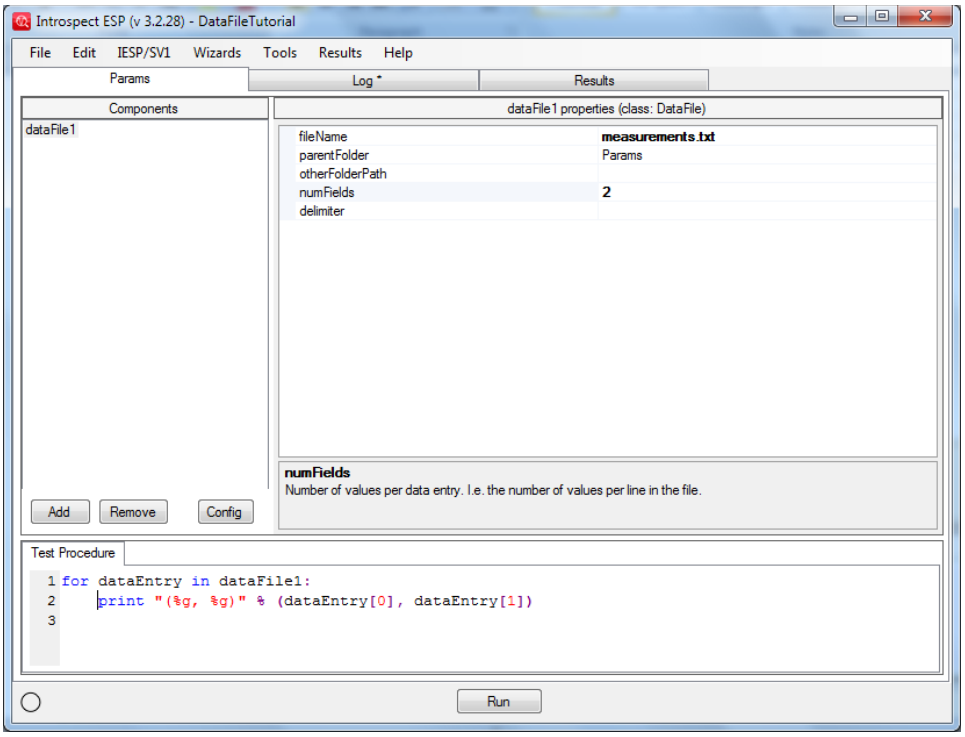

Figure 1 View of Test with data file.

If you Run this Test, you will see the values from the data file printed in the Log tab. That by itself is not so useful, but it serves to illustrate how easy it is to access the data using the DataFile component. Before we go on to make use of the DataFile component in a more interesting way, let's explore some of its capabilities.

In the code above, we looped over all values in the file by treating 'dataFile1' as if it were a list that contained the values. The 'dataEntry' variable gets assigned a numpy array with the values from each line of the file and we access the individual values via array indexing (index 0 is the first value, index 1 is the second value).

We could do the same thing in a different way by making use of some of the methods supplied by the DataFile class (see full documentation in "svt.html"). The method 'getNumDataEntries' tells you how many data entries (how many lines) are in the file. The method 'getDataEntry' returns the i'th data entry (for an index 'i'). So a longer-winded way of doing the same thing would be:

```
numEntries = dataFile1.getNumDataEnties()
for i in range(numEntries):
      dataEntry = dataFile1.getDataEntry(i)
      print "(%g, %g)" % (dataEntry[0], dataEntry[1])
```

But instead of using the 'getDataEntry' method, we could make use of the fact that we can treat 'dataFile1' as if it were a list, so we can access the i'th data entry via list indexing:

```
numEntries = dataFile1.getNumDataEnties()
for i in range(numEntries):
      dataEntry = dataFile1[i]
      print "(%g, %g)" % (dataEntry[0], dataEntry[1])
```

And instead of using the method 'getNumDataEntries', we could just take the 'len' of dataFile1:

```
numEntries = len(dataFile1)
for i in range(numEntries):
      dataEntry = dataFile1[i]
      print "(%g, %g)" % (dataEntry[0], dataEntry[1])
```

Another way of doing the same thing would be to use the method 'getData' which returns a numpy array with the data entries:

```
data = dataFile1.getData()
for i in range(len(data)):
      dataEntry = data[i]
      print "(%g, %g)" % (dataEntry[0], dataEntry[1])
```

So there are lots of different ways that you can access the data that is in the file represented by a DataFile component. There is no difference in efficiency between these different ways so choose between them according to which suits your style or the needs of the occasion. For example, if you need to pass the data array to some other function, there is no need to create a new array and fill it in with the data entries via a loop – you can just use 'getData' and pass the result of that. By the way, all of the data is read from the file the first time you access any of it and then it is cached, so future accesses don't require re-reading of the file.

Now let's do something a little more interesting with the data from our file – let's plot the data, supposing that the first value in each line is the 'x' value, and the second value is the 'y' value. The PlotCreatorBasic component has properties 'xValues' and 'yValues' which are expected to be lists (or arrays) of floating-point numbers. The array that we get from the 'getData' method is of shape (N, 2) where N is the number of data lines in our file and 2 is the number of values per line. Hence, to get the arrays of x and y values for plotting, we need to take the transpose of the array from the 'getData' method:

```
data = dataFile1.getData()
(dataXValues, dataYValues) = transpose(data)
```

Remove all the lines from your Test Procedure and then add the above two lines. Then use the "Add" button to add an instance of PlotCreatorBasic to your Test. Set the 'xValues' property of plotCreatorBasic1 to "dataXValues" (without the quotes), and set the 'yValues' property of plotCreatorBasic1 to "dataYValues" (without the quotes). Your Test should look like Figure 2.

If you Run the Test, you should get a plot that looks like Figure 3 (if you are using the "measurements.txt" data file that is in same the folder as this document).

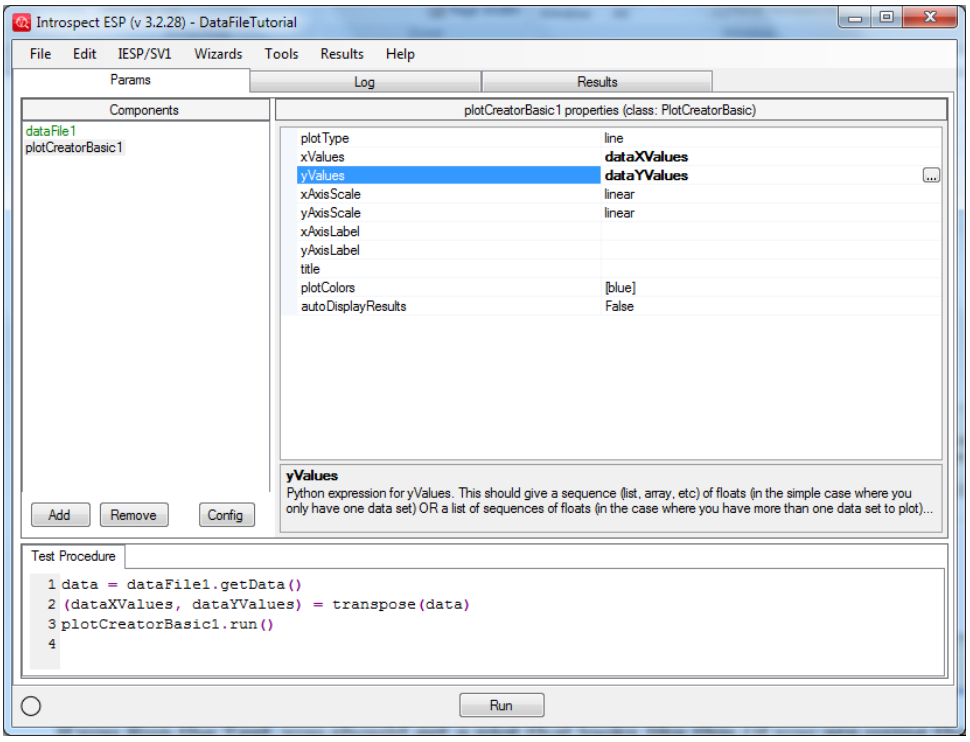

Figure 2 Latest view of Test.

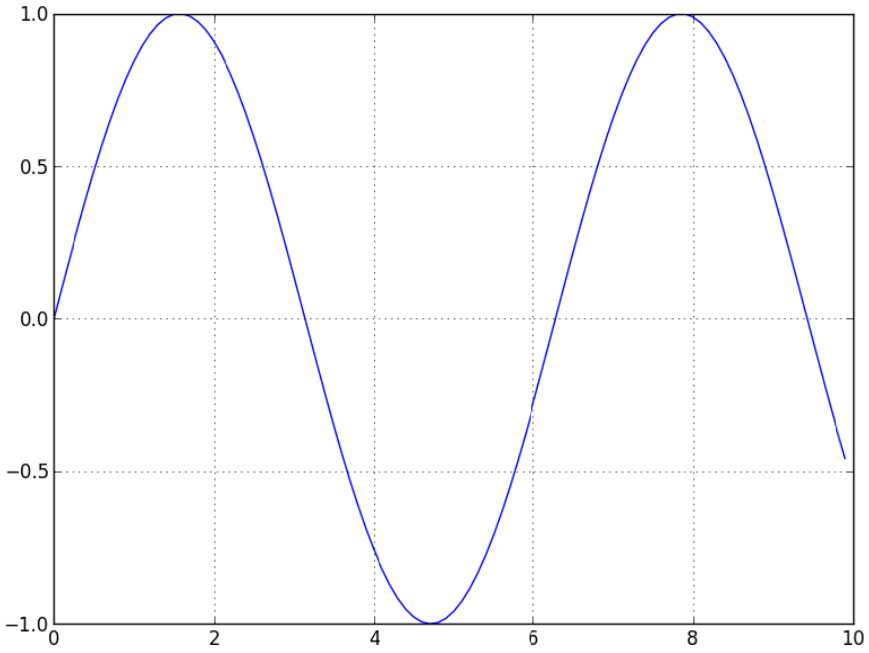

Figure 3 Plot resulting from the execution of the Test Procedure.

An alternative way of getting the 'x' and 'y' values would have been to use the 'getField' method. The following two lines are the equivalent of what we did above:

```
dataXValues = dataFile1.getField(0)
dataYValues = dataFile1.getField(1)
```

If we use the 'getField' method, then we don't need the variables 'dataXValues' and 'dataYValues' since we can enter the expressions from the right-hand side of the '=' in the 'xValues' and 'yValues' property slots of the PlotCreatorBasic component. Try this now – remove all the lines from your Test Procedure except the call to the 'run' method of plotCreatorBasic1 and then change the 'xValues' property of plotCreatorBasic1 to "`dataFile1.getField(0)`" (without the quotes) and change the 'yValues' property to "`dataFile1.getField(1)`" (without the quotes). Your Test should look like Figure 4. If you Run the Test, you should get the same plot as before.

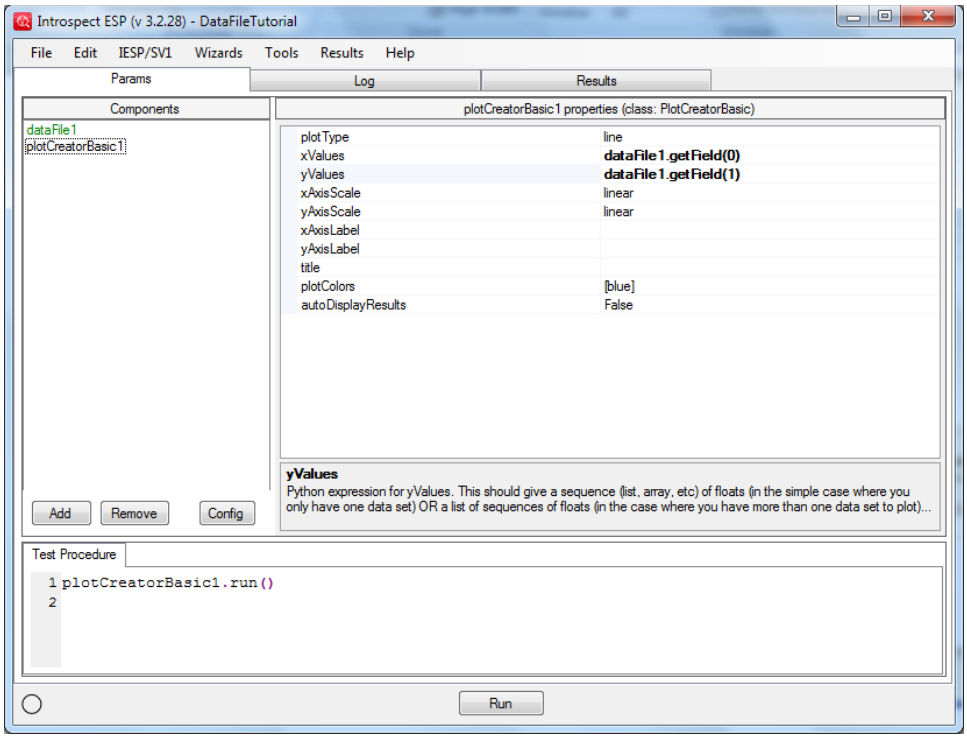

Figure 4 Simplifying the Test Procedure by calling the Data File fields directly.

Writing Files

We've seen how the DataFile component can be used to read the data from an existing file. Now, let's look at how to use it to write data to files. Add a new DataFile component to your Test – it will be called 'dataFile2'. Set the 'fileName' property of dataFile2 to "squares.txt". Leave the other properties as they are. In particular, note that we will only have one data value per line in the file ('numFields' is 1). The file "squares.txt" does not exist yet. It will get created (in the "Params" sub-folder of the Test folder) when we Run the Test.

Add a Function component to the Test (it will be called 'function1') and then rename it to 'calcSquares'. Enter the following code for the 'calcSquares' function:

```
for i in range(10):
      value = i**2
      dataFile2.appendDataEntry(value)
```

This code calculates the squares of the integers 0 to 9 and appends each in turn to the file represented by the component 'dataFile2'.

It uses the DataFile method 'appendDataEntry' which expects a list or array of 'numFields' values or (if 'numFields' is 1 as in this case) a single value.

Add the following lines to your Test Procedure after the call to 'plotCreatorBasic.run':

```
calcSquares()
print "numSquares: %d" % len(dataFile2)
```

Then add another instance of PlotCreatorBasic to the Test – it will be called 'plotCreatorBasic2'. Set the 'yValues' property of plotCreatorBasic2 to "`dataFile2.getField(0)`" (without the quotes).

Your Test should now look like Figure 5.

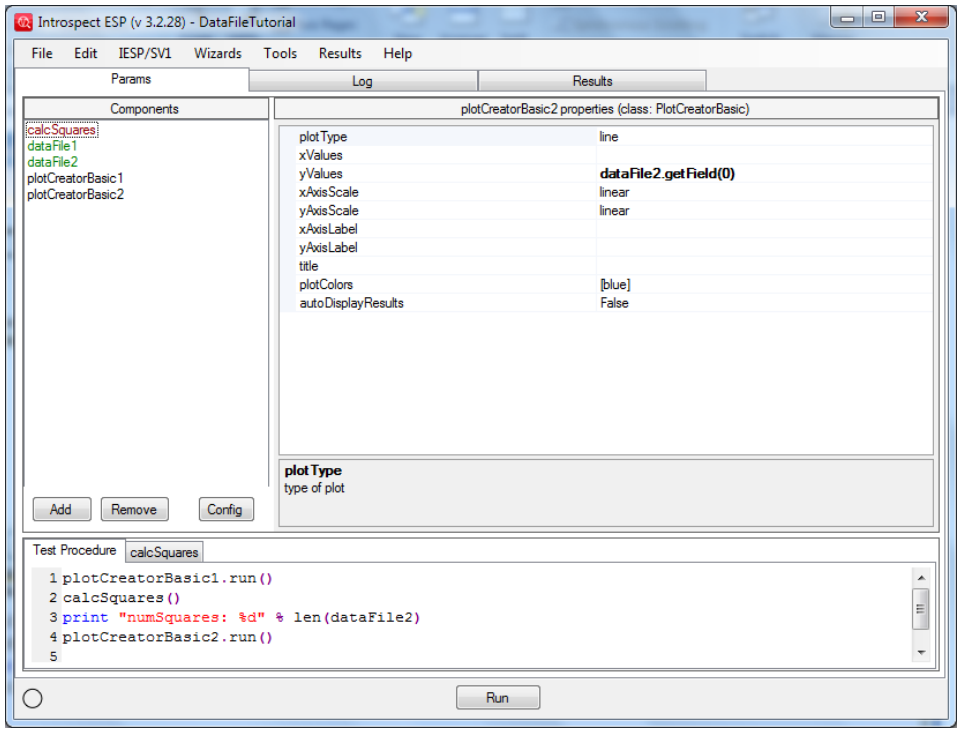

Figure 5 Final view of the Test Procedure.

Now Run the Test. You should see a line in the Log that says "numSquares: 10". And there should be one icon in the Results and double-clicking on that icon should show the sine plot you saw earlier, plus a plot of the squares.

Note that if you Run this Test a second time, you will see a line in the Log that says: "numSquares: 20" – this is because the data file is persistent and it already has 10 lines in it from the first Run. The

plot of the squares will also look different since it is plotting all of the values from dataFile2.

If you want to have the data file emptied at the start of each Test Run, add the following line to your Test Procedure at an appropriate point:

```
dataFile2.clear()
```

It is also possible to populate the data file with values all at once (instead of appending values one at a time) if you have an array (or list) with the data entries. You would use the 'setData' method to do this – for example, the following lines would make dataFile2 have a copy of the data from dataFile1:

```
data = dataFile1.getData()
dataFile2.numFields = 2
dataFile2.setData(data)
```