Developer's Guide to *Flow*

Cliburn Chan

February 28, 2008

Introduction

Flow is a software package written in Python for exploratory data analysis, clustering and classification of flow cytometric data. While the base system provides the data pre-processing and data management tools, most of the interesting functionality of *Flow* is provided by plugins. Currently, *Flow* accepts four classes of plugins IO (input/output), Projections, Statistics and Visualization. This document describes how to develop plugins for each of these classes, as well as how to develop plugins in R and compiled languages like C/C++/Fortran. Familiarity with Python is necessary to follow from this guide.

Plugins

Plugins can be implemented either as a single file (for simple projects) or as a directory of files (for more complex projects). Each directory must contain a file named `Main.py` that is the entry point for the plugin. Each standalone plugin or `Main.py` program has a class that subclasses one of the provided IO, Projection, Statistics or Visualization classes and define certain mandatory attributes depending on the class of plugin it belongs to. This will be clear by following the specific examples provided.

In general, plugins will also have to interface with the Model class where the data is stored in order to do anything useful. The Model class provides several methods for data retrieval in its API (see Doxygen generated API).

Developing IO plugins

For our first plugin, we will add the capacity to read data from a comma-separated file (CSV). The basics of reading data from any source is essentially the same:

1. Extract a list of field names and convert the numerical data into a numpy array
2. Load the data into the current Group (creating it first if necessary)

It is trivial to implement the first step for CSV files, and the Model class provides a utility method for the second step. Start by creating a new file ReadCSV.py, which will become the new plugin. We start by importing some Python modules

```
from io import Io
import numpy
import os
```

All IO plugins need to subclass the base class `Io` to let the system know that we are defining a new IO class. The module `numpy` provides efficient numerical support for Python, and we will store our data as numpy array. The standard module `os` is used to manipulate filenames in a system-independent fashion.

Now we define the actual plugin class, which is specified as a subclass of IO and called `ReadCSV`. This has several class attributes and a single method, also called `ReadCSV` to be described following the code listing.

```
class ReadCSV(Io):
  """An IO plugin to read CSV files in which
  the first line consists of field names and
  subsequent lines consist of channel data."""

  type = 'Read'
  newMethods = ('ReadCSV', 'Load CSV data file')
  supported = 'CSV files (*.csv)|*.csv|All files (*.*)|*.*'

  def ReadCSV(self, filename):
    """Reads a CSV file and populates data structures."""

    # read and parse comma-separated header and data
    text = open(filename, 'rU').readlines()
    headers = text[0].strip('\n').split(',')
    arr = numpy.array([map(float, line.strip().split(','))
                        for line in text[1:]])

    # create a new group using the filename base as label
    basename = os.path.basename(filename)
    base, ext = os.path.splitext(basename)
    self.model.NewGroup(base)

    # put the data in the current group
    self.model.LoadData(headers, arr, 'Original data')
```

The `type` attribute specifies whether the plugin will read (import) or write (export) data, and can take the values `Read` and `Write`. In general, the `type` attribute specifies how the front-end will package the functionality (for example, menu placement or addition to contextual menus).

The code to read and parse CSV data should be clear to any Python programmer, and results in a string of field names stored in [headers] and the channel data stored as a numpy array in [arr].

Finally, we create a new Group (a Group is a non-terminal node in HDF5) to store the new data, give it the same name as the data filename (stripping off the file extension if present), and load it into the model using the provided hook `LoadData`.

If you've followed and written the code, you've just written your first plugin for *Flow*. To test the plugin, write a test CSV file like this

```
foo, bar, one, two, three
1,2,3,4,5
2,3,4,5,6
3,4,5,6,7
4,5,6,7,8
1.5,2.5,3.5,4.5,6.5
```

and save it as `testfile.csv`. Now move `ReadCSV.py` to the *plugins/io* subdirectory, start *Flow* and there will be a new menu item `File|Load CSV data file` that allows you to browse for and open `textfile.csv`. After loading the file, the Control Frame will show a new group under the root labeled `testfile`. Expand the group by clicking on the horizontal triangle to show the array `data`. Selecting `data` will show its associated metadata (in this case, only the field names are useful), and you can plot the data using one of the `Graphics` menu options etc.

Developing Statistics plugins

The process of developing statistics plugins is very similar, and only a trivial example will be shown here, as realistically, most statistics plugins will either be written in a compiled language (C/C++/Fortran) or interface with the R statistical libraries. Such foreign language plugins will be described later.

We will develop a statistics plugin to calculate the mean value of each column – while trivial, this demonstrates how to retrieve data from the Model, process it and append new results to the Model.

We begin by making the necessary imports

```
from plugin import Statistics
import numpy
```

As before, our plugin class needs to subclass `Statistics`, then it is a simple matter of calling the Model API to retrieve data, find its mean across columns and append the calculated statistic.

```
class Mean(Statistics):
  """Calculate channel averages and
  append statistic to group."""
  name = 'Average'
```

```python
    def Main(self, model):
        """Retrieve column data from Model and
        attach calculated column means."""

        # make a copy of the currently selected group's data
        data = model.GetCurrentData()[:]

        # calculate the mean per channel
        avg = numpy.mean(data, axis=0)

        # make a new array to store the calculated means
        model.NewArray('average', avg)
```

Copy this file to the *plugins/statistics* sub-directory, fire up *Flow*, and apply the new menu item `Statistics|Average` to the `testfile` data. This generages a new sub-group in the Control Frame with the label `average`. Right clicking and choosing `Edit` will show the calculated channel averages in a table.

Developing R plugins

Thanks to the `RPy` library, it is generally very simple to write an R plugin for *Flow*. We will illustrate how to find the independent components of the data using the R library `fastICA`. We assume that the user has a working R installation and has installed the `fastICA` library (see the R documentation for detaails at `http://www.r-project.org`).

Begin, as before, with the imports, and load the `fastICA` library

```python
from plugin import Projections
from rpy import r
import wx
import numpy
r.library("fastICA")
```

Now create a class that will do the necessary calculation and communication

```python
class Ica(Projections):
    """Uses the fastICA library to find
    independent components."""

    def Main(self, model):
        k = wx.GetNumberFromUser("ICA Dialog",
                                 "Enter number of components",
                                 "k", 1)
        data = numpy.array(model.GetCurrentData()[:])
        ica_data = r.fastICA(data, k)
        fields = ['Comp%d' % c for c in range(1, k+1)]
        model.updateHDF('ICA', ica_data['S'], fields=fields)
```

As can be seen, the class is almost trivial. We ask the user for the number of components k desired using a standard wxPython dialog, then pass a copy of the current data and k to R, and update the Model with the result and new appropriate field names. See the `fastICA` documentation for more details (`http://cran.r-project.org/web/packages/fastICA`) of its functionality.

Developing compiled language plugins

Sometimes, Python and R are just too slow and we need the speed of a compiled language like C, C++ or Fortran, but still want to use *Flow* to provide a frontend to these routines. Before plunging in, do check if Python optimization tricks will be enough – see guide at `http://www.scipy.org/PerformancePython` for examples.

We will not actually develop an example of a compiled language extension here, but merely suggests possible routes and resources. To allow *Flow* to interface with C or C++, a typical strategy is to compile a Python extension module, following the instructions at `http://www.python.org/doc/ext/intro.html`. Alternative and simpler methods include using Swig (http://www.swig.org), and for C++ only, using the Boost.Python library (`http://www.boost.org/libs/python`). For Fortran, we recommend using F2PY (`http://cens.ioc.ee/projects/f2py2e`). C++ examples can be found in the *c++* sub-directory.

Once an extension module is compiled into a shared library, it can be imported into Python like any other module and the subsequent plugin development is as already described for the various plugin classes.

Developing Visualization plugins

Writing a visualization plugin is rather more involved than the previous plugins, and requires knowledge of the specific GUI toolkits to be used, and will also not be described here. Interested developers will have to look at the provided source code examples in the *plugins/visualization* for now.