Bisque module integration

Andrew Predoehl

Summary

This document describes the key concepts involved in the process of integrating a new image analysis module into the Bisque platform. The scope of this document is restricted to batch-mode binary image analysis programs running in a Unix-style environment. Bisque offers a robust way to augment such a program with a graphical user interface, and share it with others. The present document focuses on an approach to such an augmentation using Python, a popular scripting language. Python offers many facilities that support this task, and the Bisque developers have created a convenient Python API wrapping the REST interface of Bisque.

Introduction to Bisque

In this first section, we briefly introduce Bisque, to provide motivation for the user considering integrating an analysis program as a new Bisque module. Please see the paper by Kvilekval et al. for a full presentation of the rationale for Bisque.

The domain we have in view is that of scientific computing where the input data are substantially images, and the output or outputs are semantically compact. In other words, the inputs are pictures, possibly very large files with two or three spatial dimensions, and perhaps a time dimension as well. But the outputs are expected to contain meaningful conclusions, rather than images. For example, a seed counter program might take an image showing seed kernels, and generate output consisting of a count of the number of kernels and the quintiles of the seed sizes. Another example: a root-tracing program might identify and trace the position of a plant root over time. The input would be a time-series of images, but the output would be a semantic representation of the root trace -- possibly as a polygonal path along the centerline, for example. Note that the inferred root trace in this case is essentially a curve, rather than a grid of pixels. Although the root trace could be represented in the form of an image, that is just one of many possible representations (such as a list of spline curve parameters). Images tend to carry information diffused unevenly among thousands or millions of pixels, but we are here considering systems where the results are more concentrated representations of meaning. To be colloquial: we have images, but images are a burden. Who has time to look at images? We prefer *answers*. A good analysis program can turn images into answers. That is why in the discussion below, we discourage image-based output.

Bisque is a web-based platform for sharing image data and algorithms among many users. Users can share, browse, and annotate images and other data easily, via a powerful web-browser interface. With some additional effort, users can also share image analysis algorithms, known as *modules*. Modules can work on images both singly and in batches. Via Bisque, modules can offer a friendly, web-based user interface that is decoupled from the analysis algorithm. A record of each module execution (abbreviated as *mex*) is automatically stored in Bisque's database. Finally, Bisque offers a structured format for graphical results (such as points, circles, and polygonal paths).

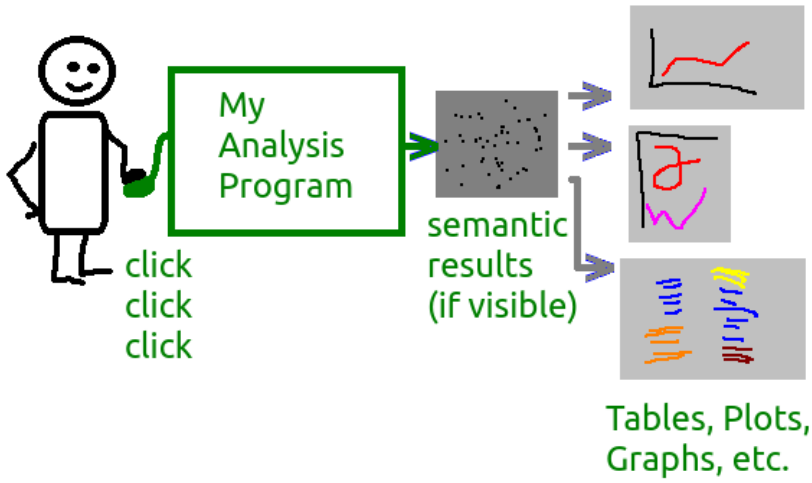
Architecture

From a high-level perspective, a new Bisque module consists of an analysis engine wrapped in some extra layers of software. The process of adapting an existing program into a Bisque module means creating a “wrapper” around the program, intervening in all the previous user interaction for both input and output. (This may necessitate some changes to the original analysis program as well.) A cartoon of the transformation is shown in Figure 1. The user is “replaced” by Bisque and, often, an additional script that helps wrap the analysis program. We call this additional script “middleware,” since it is located between Bisque and the original program. The task of the middleware wrapper is to translate the user inputs from Bisque into the form required by the analysis program, and also to translate the outputs from the analysis program into the form accepted by Bisque. In other words, Bisque only “sees” the wrapper script when it interacts with your module; whereas the analysis program only sees the wrapper script, not Bisque, providing its inputs and receiving its outputs.

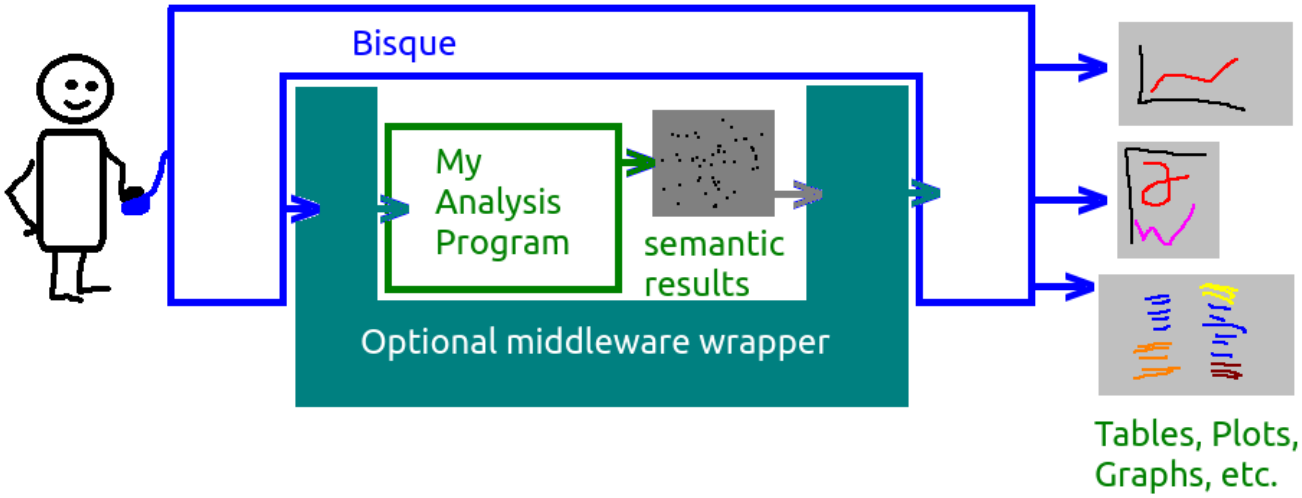
The discussion that follows assumes that analysis programs run in a Unix-like environment (such as Linux or BSD). I have no experience working with Bisque on Windows, or in the Matlab environment. Also, we assume that the program receives all its inputs either from its local filesystem, or from the console command line when the program is invoked. We regard console and mouse interaction as out of our scope. Finally, we expect that all program output is written to the console or to the local filesystem, not to a graphical window. Some graphical output is possible, as will be discussed below, but as discussed above, we frown on output in the form of images. The output should be semantically compact.

Furthermore, if the analysis program emits output showing its progress to the console, this output should somehow be easily separated automatically from the final results, since the middleware wrapper will be responsible for performing the separation. In particular, error output should be easily distinguishable from successful results.

My own experience with Bisque involves analysis code that was originally written in C and C++ by developers who had never heard of Bisque, and my approach was to wrap those binaries in a middleware wrapper script written in Python. With a moment's thought, it should become obvious that the approach described here could be adapted easily for an analysis program written in Python (e.g., using Python-Scitools).



(a)



(b)

Figure 1. (a) At the outset, your analysis program interacted with the user, generated results, and produced visualizations in the form of tables, plots, overlays, and other renderings. Not shown are image inputs from the file system. The user is possibly aware of "pure" semantic results that underlie the renderings. Debugging messages may be sent to a visible window.

(b) After adapting the program into a Bisque module. Your analysis program will be wrapped within one or two layers of additional software. The user will only interact with Bisque, both for input and output, via a web browser such as Chrome. Bisque will interact with the analysis program, perhaps through a middleware wrapper (a custom script) that surrounds it. The purpose of the wrapper is to adapt the inputs and outputs of the core analysis to the formats used by Bisque. Bisque can produce certain visualizations of the output, either directly, or through a JavaScript helper script. Debugging output, if any, is logged to a file.

Wrapper Details

Execution platform

From a systems perspective, your Bisque module will run in a Bisque installation, either a complete Bisque server, or a reduced-complexity system called an *engine server*. Either way, the platform that launches the code must ordinarily have access to the internet in order to support web-based sharing of analysis options. It is a nontrivial task to configure such a system, since it must run a web-server such as Nginx or Apache, and if firewalls are used, the machine needs at least one port open through the firewall. See the Bisque wiki for documentation about how to set up the right kind of Bisque server.

Module Definition File

Everything that Bisque knows about your module's inputs and outputs comes from the required *module definition file* that sits in the same directory as the module files. The module definition file is an XML description of the module's interface: not only its inputs and outputs, but associated descriptive information such as the title, description, version number, location of the help file (which you should write), and more. Look to the Bisque wiki for complete documentation on the format of the module definition file, which can in some cases be quite complicated. Nevertheless, we add a few notes here about the file structure.

The module definition file is arranged as top level <module> . . . </module> element, under which are found a hierarchy of <tag> . . . </tag> elements. Bisque recognizes certain values of the name attribute of the tags. The most important are the names *inputs* and *outputs*. In other words, there will be two children of the module element that look like so:

<tag name="inputs"> . . . </tag>

<tag name="outputs"> . . . </tag>

. . . and these elements generally have children (sub-elements) too. The structure of this part of the XML document tree determines the inputs available to your program, and the outputs it may generate. Template sub-elements of the inputs tag will prompt Bisque to generate an HTML form interface for your module. Here is an example:

```
    <tag name="inputs">
        <tag name="image_url" type="image" >
            <template>
              <tag name="require_geometry">
                 <tag name="t" value="stack" />
                 <!-- maybe a single z plane or maybe a z stack, dunno -->
                 <tag name="fail_message" value="Input must be a timeseries
of X,Y images or a timeseries of X,Y,Z stacks." />
              </tag>
            </template>
        </tag>
        <tag name="spot-size-min" type="number" value="2">
            <template>
                <tag name="label" value="Tip size minimum" />
                <tag name="minValue" value="0.1" type="number" />
                <tag name="maxValue" value="50" type="number" />
                <tag name="allowDecimals" value="true" type="boolean" />
                <tag name="decimalPrecision" value="1" type="number" />
                <tag name="step" value="0.1" type="number" />
                <tag name="showSlider" value="false" type="boolean" />
                <tag name="units" value="microns" />
                <tag name="description" value="Width of smallest detection
                of a growing pollen tube." />
            </template>
        </tag>
      . . .
```

The above example shows two input tags. Each surrounds a <template> sub-element, the first of which creates a button to launch an image requester. The second <template> creates an HTML form element for numeric input. There are many more examples in the default modules provided by Bisque, and more information is found at the Bisque developers' wiki.

The module definition file also enumerates the outputs that Bisque will expect from your module, in the form of XML. In other words, Bisque will require your module to provide XML output whose tree structure matches that of the XML in the relevant section of the module definition file. (The relevant section is the <tag> element that is a child of the top-level <module> element, and which has its name attribute set to “outputs.”

Graphical output (such as an annotation on an input image) is organized as a child element of a <tag> element having a type attribute of "image." For example:

```
<tag name="outputs">
      <tag name="TrackedImage" type="image">
          <template>
              <tag name="label" value="Tracked pollen tubes" />
          </template>
          <gobject name="pollen_tube_tracker">
              <template>
                <tag name="plot" value="true" type="boolean"/>
                <tag name="export_csv" value="true" type="boolean" />
                <tag name="export_xml" value="true" type="boolean" />
                <tag name="export_gdocs" value="true" type="boolean" />
                <tag name="preview_movie" value="true" type="boolean" />
              </template>
          </gobject>
      </tag>
 </tag>  <!-- END of tag name="outputs" -->
```

Notice that the <gobject> element contains a template tag. This causes Bisque to create an HTML interface allowing the graphical output to be plotted, saved, and so on as described by the template.

Plots as Output

Users often want to see results presented in the form of a plot (such as a line chart or a bar chart). This can be accomplished through the use of a JavaScript rendering program for the plot. The basic idea is that the module itself must generate the numerical results, and provide them to Bisque in XML format, as described above. Then the module description file lists one or more JavaScript rendering programs, which Bisque will launch when the module finishes executing. The JavaScript rendering programs use Bisque's so-called *statistical service*, which can extract selected results from the module output. One of the most useful techniques for doing so is via *xpath* queries. (An xpath query is a way of pattern-matching elements of an XML tree. The xpath language is a useful XML-related standard that is independent of Bisque and JavaScript.) The details of generating such plots are beyond the scope of this document, but for an example, see the JavaScript used in the standard Bisque module called RootTipMulti.

Python Bisque API

Bisque is designed for HTTP-based interaction between its major components, designed to use something called a *REST* interface. (Do not worry if this is an unfamiliar concept.) It is technically possible to write a Python module wrapper using a REST interface as well. However, for convenience, the Bisque developers have provided an alternative that I consider easier to use: an interface to Bisque that presents as a Python object, made available through a Python module called *bq.api*. It is implemented using the REST interface.

Using the Python bq.api, your wrapper script can access all the module inputs described in the module definition file. This includes the input image. Also it lets your script hand over semantic output, when the analysis completes successfully. Finally, it lets your script report errors and status messages, and terminate the module execution (with normal status or failure status). Here is a list of some of the most useful bq.api functions and methods:

```
init_mex
update_mex
fail_mex
finish_mex
load
util.fetch_image_planes
```

Useful Python Concepts

There are a number of features in Python that are very useful in a wrapper script, but which a novice Python programmer might not know of. Yet there is no need to reinvent the wheel.

Exceptions

I recommend that error conditions in the module wrapper be handled using exceptions. In particular, raise an exception whenever something goes wrong. The top-level code of my wrapper script looks something like the Python-style pseudocode below. Note that *bqsession* below represents the bq.api session object. The block structure below elegantly shows how to handle all paths of execution, whether or not an error occurs.

```
try:
     setup_things()

     # launch the analysis engine
     # ------------------------------------------------------------
     outputs = run_cpp_analysis(bqsession, image_stuff, other_stuff)
 except AnalysisErr as e:
     logging.exception(str(e))
     bqsession.fail_mex(str(e))
 except:
     erm = 'failure during inference'
     logging.exception(erm)
     bqsession.fail_mex(erm)
 else:
     bqsession.finish_mex(tags=[outputs])
```

Notice the setup and analysis are wrapped in a “try” block, and any number of exception-catching clauses (in this case, two) follow. Each such clause ends in a call to the fail_mex() method of the bq.api session object. However, if nothing bad happens, the “else” clause executes, and the analysis outputs are handed to Bisque as the module execution finishes.

Logging

Debug, warning, and error messages generated by the module are likely never to be seen, if the programmer simply uses print. However, Python offers a nice library called logging, which supports different levels of verbosity for diagnostic output.

Subprocess

The Python subprocess library lets your script run the binaries (if any) of the analysis code in the form

of a subprocess, and to capture its stream output and return value. This library should be preferred over older Python features such as *os.pipe*.

Regular expressions

Parsing and interpreting text-format output can be a tedious task, but it can be made much easier via the use of regular expressions (regexes), which are a potent pattern-matching language, although it has a well-deserved reputation for being infamously obscure. Python has many of the regex features found in Perl. It is difficult to overstate the value and importance of regexes.

xml.etree.ElementTree

The ElementTree library is a good way to generate XML, especially when generating <gobject> structures.

Conclusion

Integrating an image-analysis program into Bisque as a module is a nontrivial task. The Bisque system can turn a console-driven application into a robust, web-available application with a sophisticated HTML interface. It should come as no surprise that the wrapping process requires thoughtful construction of an adapter script and associated interface specifications. Since the Bisque system is still in development, the procedures are likely to continue evolving after this document is completed. Those who wish to wrap an application will do well to study the Bisque documentation, examine the example modules distributed with Bisque, and consult with the Bisque developer and user online community when necessary.