THE OPEN MICROSCOPY ENVIRONMENT MATLAB HANDLER: COMBINING A BIOINFORMATICS DATA & IMAGE REPOSITORY WITH A QUANTITATIVE ANALYSIS ENVIRONMENT

Tom J. Macura[1], Josiah N. Johnston[1], Douglas A. Creager[2], Peter K. Sorger[3], and Ilya G. Goldberg[1]

[1]Image Informatics and Computational Biology Unit, Laboratory of Genetics, National Institute on Aging, National Institutes of Health, 333 Cassell Drive, Baltimore MD 21224, USA {tmacur1, siah, igg}@nih.gov; [2]Oxford University Computing Laboratory, University of Oxford, Oxford, UK, douglas.creager@comlab.ox.ac.uk; [3]Dept. Of Biology, Massachusetts Institute of Technology, Cambridge, MA psorger@mit.edu

ABSTRACT

High-throughput scoring of image-based biological assays heavily depends on the extraction of quantitative numerical information from microscopy images. This paper describes how the Open Microscopy Environment (OME)'s MATLAB Handler combines a bioinformatics data and image repository – OME – with a numerical analysis environment – MATLAB. An OME/MATLAB coupling leverages the wealth of intellectual property invested in MATLAB algorithm implementations by making them accessible to OME managed data, and takes advantage of OME's features for organized analysis: workflow manager, provenance recording, result reuse, and distributed analysis. The MATLAB Handler is a general method for incorporating legacy MATLAB code with the OME analysis system. We used it to implement a complex image analysis workflow for calculating >800 numerical image descriptors as part of a general image classification technique.

1. INTRODUCTION

Novel bio-informatics technologies are necessarily being developed to support the new microscopy approach based on high-content image-based cellular screens (commonly referred to as High Content Screens, or HCS).

In HCS, fluorescence markers are monitored to observe the effects of compounds or genetic manipulation on cellular activities or morphology. Fluorescence markers monitor changes in a tagged protein's distribution and intensity. Examining subtle changes requires high-resolution imaging and therefore generates high information content. These screens are high-throughput because a whole class of compounds or genes must be exhaustively investigated in a systematic manner through tens of thousands of samples.

Key to extracting information from HCS are automated image processing and analysis techniques. These techniques offer the promise of (1) automation, (2) objectivity and reproducibility, and (3) potentially higher sensitivity than expert human observers.

1.1. Automated Image Classification

Our group has developed a computer-driven image classification system that can assign images into groups based on rules automatically inferred from a user-defined image training set. [1] In this classification system, each image is decomposed into a signature vector of continuous values (over 800) using a bank of filters. We use a large varied signature set because different HCS applications have different image particularities. After extraction, the signature vectors are treated with an entropy-based discretization method and used to train a Naïve-Bayesian network via the forward-selective method. During training, the classifier automatically determines which image descriptors are most relevant to the immediate classification problem. The resulting trained Bayesian network can classify previously unencountered images.

We are validating the efficacy of our classification approach using various microscopy and radiology image sources. We have already successfully applied our approach to the prediction of *C. elegans* muscle age, identification of sub-cellular organelles, and S2/RNAi screens for absence of centromeres, binucleate phenotype, loss of filopodia, etc. [1]

A typical classification problem involves four classes, 150 images per class (for training and validation) and generates 800 features per image. Thus, training the image classifier yields 480k unique numbers. Once training and validation is complete, the classifier is applied to a screen which may consist of 50k images, with perhaps a dozen unique numbers for each image selected for their classification power. In this way, well over a million individual values are associated with a single experiment.

1.2. The Open Microscopy Environment (OME)

The Open Microscopy Environment project is both a set of information and interchange standards for microscopy images [2] and an open-source software suite for managing and analyzing images and image information. [3] OME is being developed by an international consortium of academic groups and is intended to become a general-use informatics framework for biological research involving microscopy and imaging.

OME is designed as a client-server architecture where images and data are stored on a centralized OME server. Users access OME data with either a lightweight web-based interface

or a Java client. The OME server is composed of the OME Image Server (OMEIS) and the OME Data Server (OMEDS). OMEIS is an interface to a repository where image pixels, original image files, and other large binary objects are stored [6]. OMEDS stores all meta-data and derived data about the images in a database. The data-server is made up of a PostgreSQL database with a dynamically generated schema along with a collection of Perl classes providing the middle-ware for accessing the data in a structured object-oriented manner.

OME supports a wide range of microscopy and radiology multi-dimensional image formats. Image meta-data is automatically extracted from these formats and automatically entered into the OME database, but can also be imported into OME using Excel® or tab-delimited text files. These image and meta-data files serve as the primary entry point for data into OME.

Derived data is produced by applying algorithms to initial data. Each datum is a logically distinct object, called an attribute, and is an instance of a semantic type. Each semantic type defines several fields, or semantic elements (SE), which specify the simple values that make up the data of the type. OME's underlying data model is highly flexible. Semantic types, which are essentially ontological terms, are defined in XML and imported into a live OME system, thus expanding OME's dynamically generated schema.

All data in OME is stored in these attributes and the analysis module is the sole mechanism by which new attributes are created. Analysis modules usually represent computational algorithms, though they are also used to represent user input or parsing of external files. The inputs and outputs of modules are semantically typed, and thus modules serve as declared transformations between sets of semantically typed attributes.

The OME Analysis Engine (AE) delegates the implementation specific logic about executing an analysis module's algorithm and managing I/O to *handlers* e.g. MATLAB Handler. This makes it easier to add support for modules implemented in other programming languages.

Analysis modules can be connected, via their inputs and outputs, to form actionable workflow plans called analysis chains. These chains are constructed using the ChainBuilder interactive user interface [2] or by directly writing XML [4]. When the AE executes analysis chains against a dataset of images, it does provenance recording [5], reuses results of modules it has determined have been executed before, and distributes analysis module execution across multiple CPUs on the network.

2. OME / MATLAB COUPLING

MATLAB is a high-level, interactive programming environment for numerical computation. With more than 1000 built-in mathematical, statistical, and engineering functions, including statistical, image analysis, and machine learning tools, MATLAB is especially useful for quantitative bioinformatics.

Our lab uses MATLAB: (1) as an interactive environment for data analysis and visualization leading to algorithm development; and (2) as a programming language in which we can implement numerical analysis algorithms much faster than with traditional programming languages such as C, C++, or Fortran. We designed the OME/MATLAB coupling to supporti both usage methods: (1) we wrote a MATLAB binding for the OMEIS client library that makes pixels stored in OMEIS accessible in MATLAB like regular image files; and (2) we wrote a MATLAB handler for the AE that allows algorithms, implemented in MATLAB and wrapped in a simple XML syntax, to be executed as OME Analysis Modules by the AE.

3. OMEIS-LOCAL IMAGE FORMAT

The OME Image Server (OMEIS) is built to store millions of images and allow efficient access to them for reading and writing over a universal interface. [6] It has features such as scheduled compression and purging, transparent inflation of compressed files and pixels, and transparent recovery of purged pixels from their original files. OMEIS is usually run on a high-performance computer with a high-capacity disk array and optimized bandwidth to the network. Our labs routinely use it to store thousands of images comprising hundreds of gigabytes.

It is more efficient and convenient to access OMEIS pixels directly in the MATLAB environment than to access them through temporary file-system intermediaries.

We defined an OMEIS-Local file-format for managing this interaction. OMEIS-Local files stored on the local file system contain references to pixels managed by OMEIS. These files, which must be constructed manually or programmatically, contain a magic string identifying them as OMEIS-Local files, the URL for a particular OMEIS, and a unique identifier for the pixels object on the specified OMEIS.

Since the OMEIS-Local file-format is registered in MATLAB, it is treated as a natively supported image format by the standard MATLAB image I/O functions. The appropriate pixels are seamlessly retrieved via http from the server and placed into the MATLAB workspace memory.

OMEIS-Local stubs allow MATLAB functions (including compiled proprietary MEX files) that use image filenames as input parameters to work without alteration with pixels managed by OMEIS. These benefits come at the cost that users must write the stub files. Alternatively, we provide GetPixels/SetPixels

```
<AnalysisModule
   ModuleName="Haralick Features 2D"
   ModuleType=
    "OME::Analysis::Handlers::MatlabHandler"
   ProgramID="HaralickFeaturesRI"
   ID="urn:lsid:openmicroscopy.org:Module:7704">
  <Description> ... </Description>
  <Declaration>
    <FormalInput  Name="Pixels Plane Slice"
(A)         SemanticTypeName="PixelsPlaneSlice"         Count="!"/>
    <FormalInput  Name="Texture Distance"
          SemanticTypeName="HaralickTextureDistance"  Count="?"/>

    <FormalOutput Name="Angular Second Moment"
(B)         SemanticTypeName="CoOcMat_ASM"      Count="!"/>
    <FormalOutput Name="Contrast"
          SemanticTypeName="CoOcMat_Contrast" Count="!"/>
       ...
  </Declaration>

  <ExecutionInstructions ExecutionGranularity="I" ...>
    <FunctionInputs>
(C)   <Input><PixelsArray FormalInput="Pixels Plane Slice"
          ConvertToDatatype="uint8"/></Input>
      <Input><Scalar InputLocation="Texture Distance.Distance"/></Input>
    </FunctionInputs>
    <FunctionOutputs>
(D)   <Output><Vector
          DecodeWith="Haralick_Avg_and_Range_Output_Vector"/></Output>
    </FunctionOutputs>

    <VectorDecoder ID="Haralick_Avg_and_Range_Output_Vector">
(E)   <Element Index="1"  OutputLocation="Angular Second
          Moment.ASM_avg"/>
      <Element Index="2"  OutputLocation="Contrast.Contrast_avg"/>
        ...
    </VectorDecoder>
  </ExecutionInstructions>
</AnalysisModule>
```

Figure 1: An XML wrapper allows a MATLAB implemented algorithm to be executed by the OME analysis engine via the MATLAB Handler.

MATLAB functions that load pixels to/from OMEIS based on input parameters. These function are easier to use but don't have legacy support.

4. MATLAB HANDLER

A wrapper for a MATLAB algorithm is a set of XML elements that define, according to the Analysis Module Library schemata [7], how the MATLAB algorithm ought to be executed by the AE:

- **ModuleName**: serves mostly for user benefit.
- **Category**: Modules belong to a hierarchical structure of categories. This is a simple organization scheme that aids users by grouping modules by similarity.
- **ModuleType**: refers to the Perl class that implements the handler for this module.
- **ProgramID**: refers to the name of the MATLAB function that implements the module.
- **ID**: a globally unique Life Science Identifier (LSID) [8]
- **Description**: is free-text that informs potential users as to the module's algorithm and implementation.
- **Declaration**: defines the names, semantic types, and arity of a module's inputs and outputs.
- **Execution Instructions**: are instructions that describe how inputs and outputs to the algorithm implementation correspond to OME semantic types. These are interpreted by an AE handler and are therefore handler specific.

Since MATLAB analysis modules have access to all data that is managed by OME, there must be an established equivalence between MATLAB data-types and OME data-types. This is summarized in online documentation. [9]

The MATLAB handler does strong data-type checking for all MATLAB inputs and outputs. Mismatches between the class of the MATLAB variable and the semantic element data-type defined in XML are considered errors. Explicit type-casting can be used in all input/output execution instructions via the `ConvertToDataType` XML tag. This tag specifies the MATLAB class the module's input/output should be converted to in the MATLAB environment. An example of explicit casting is illustrated in Figure 1c.

The `<ExecutionInstructions>` element has two parts: `<FunctionInputs>` and `<FunctionOutputs>` (Figure 1c and 1d). These blocks are a set of `<Input>`/`<Output>` elements that describe each of the function's inputs/outputs and can either be a scalar, a pixels array, or a vector. Their order in XML corresponds to the order they will be passed/retrieved from the function. The MATLAB handler doesn't yet support MATLAB functions that return a variable number of outputs.

4.1. Scalar Inputs and Outputs

Scalar input definitions associate values of individual semantic elements - or constants - with inputs to MATLAB functions. Output definitions indicate how MATLAB results will be stored in OME.

- `<Input><Scalar InputLocation="FormalInput.SE"/></Input>`: the handler passes the value of the formal input's semantic element as an input into the function.
- `<Output><Scalar OutputLocation="FormalOutput.SE"/></Output>`: the function's output value is stored as a new attribute's semantic element.

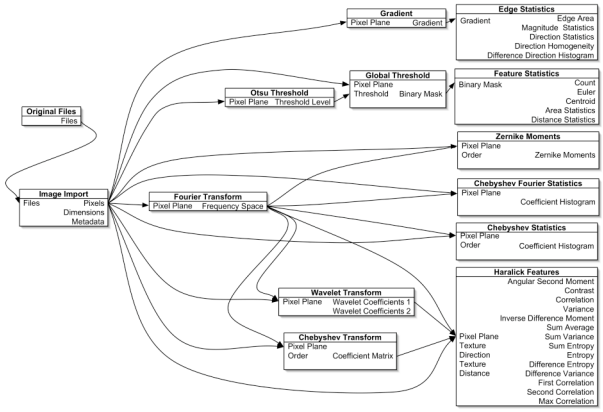

Figure 2: This OME analysis chain is a workflow of 12 MATLAB analysis modules that computes an image's signatures.

- `<Input><ConstantScalar Value="3.14"/></Input>`: is a mechanism by which an input to the MATLAB function can be hard-coded in XML.

4.2. Pixels Inputs and Outputs

The MATLAB handler makes it easy to take pixels from OMEIS and pass them as a 5D XYZCT array into a MATLAB function.

- `<Input><PixelsArray FormalInput="FormalInputName"/></Input>`
- `<Output><PixelsArray FormalOutput="FormalOutputName"/></Output>`

4.3. Vector Outputs

VectorDecoder syntax allows us to use a vector output from MATLAB to define OME attributes. We have not encountered a use case that requires a complementary VectorEncoder.

In the `<FormalOutput>` block, `<Output><Vector DecodeWith="VectorName"/></Output>` associates an XML VectorDecoder identified by `DecodeWith` with a particular function output. A VectorDecoder maps the vector's components to new attributes' semantic elements using a series of `<Element Index="i" OutputLocation="FormalOutput.SE"/>` tags. Figure 1e is an example of a VectorDecoder.

4.4. Implementation

Mathworks® provides the MATLAB Engine and MX Array Manipulation libraries as an external interface by which MATLAB can be called from within C programs. We used the Perl XS interface to define Perl bindings for the Mathworks® C-libraries. It is through these bindings that the MATLAB Handler, which is a Perl class, executes analysis modules implemented in MATLAB.

5. VALIDATION & EFFICIENCY

Executing an analysis chain of OME MATLAB modules must produce exactly the same results as running it natively in the

MATLAB environment. We tested the OMEIS & OMEDS connections independently and also did a comprehensive test by executing analysis chains with known results.

OMEIS enforces, through the use of SHA1 digests, a uniqueness constraint on all pixels. We exploited this constraint to test the OMEIS –MATLAB connection. We wrote a MATLAB script that would connect with OMEIS, download each pixels set to serve as a template, and replicate this template as a new pixels set. The replication was designed to use the maximum number of OMEIS MATLAB functions. When we ran this script against our local OMEIS installation (10,000 images, 100+ GB) we observed that the PixelsIDs returned were the same as the input PixelIDs, indicating that the pixels SHA1 digests were always identical.

The underlying PostgressSQL database stores real and double precision according to the IEEE Standard 754 for Binary Floating-Point Arithmetic. MATLAB, on the other hand, uses denormal numbers to store very small numbers at lower precision. This arcane point implies that for the few applications that use very small floating-point numbers, OME and MATLAB results differ.

We wrote a "glue" MATLAB script to mimic an OME analysis chain by connecting individual MATLAB modules. We compared the results of this script, where intermediate results are always kept within MATLAB, to the equivalent chain computed with the OME analysis engine, where intermediate results pass through OMEIS and the database. All 800 results varied by less than 10^{-12}.

The overhead due to OME managed execution depends on the number of MATLAB modules and how many inputs/outputs they each have. Although it is probably better from a data-modeling and modular design perspective to have many smaller modules, it increases overhead in an OME-based implementation. Currently 75% of the total real-world execution time is spent within the MATLAB modules and the other 25% is OME overhead. We measured this by executing the signature chain (Figure 2) on a dataset containing 20 16-bit 200x200 images. It ran 6 minutes per image.

6. DISCUSSION

The MATLAB Handler has been developed as a solution to a specific problem: porting our automated image classification system, implemented in MATLAB, into an OME chain of analysis modules that could be executed by the analysis engine.

Designing the execution instructions has been a compromise between the simplicity of the XML syntax and generality. Syntax complexity is also highly correlated with the cost of developing the handler implementation. There are innumerable valid use-cases for conversion of inputs and outputs from MATLAB functions into OME attributes.

Modularizing the image classifier resulted in 12 diverse MATLAB analysis modules. We found that there are a handful of common patterns for specifying inputs and outputs to MATLAB functions and that we can support most MATLAB functions in OME without modification: approximately 20% of MATLAB functions will require MATLAB wrappers that do additional data-conversion. Alternatively, a Perl class based off of the MATLAB Handler can be written to implement any module-specific interface.

Although motivated by a specific application, the MATLAB Handler is a general method for integrating analysis routines implemented in MATLAB with biological data stored in OME. Our use-case validates that OME managed analysis execution doesn't corrupt data and that execution instructions are sufficiently expressive to wrap around real-world image analysis routines. We found that integration of a module has a shallow learning curve.

Other projects, such as the microarray analysis platform *GenePattern*, also have integrated MATLAB with a data store. [10] Our MATLAB handler is markedly different. Firstly, our approach is very general–the required interface is specified per MATLAB module in XML—so most legacy applications are supported without alteration. Secondly, since the handler is based on the OME extensible data model, it is not limited to genomic/proteomic data but supports any textual, numerical, and visual data.

Planned future work involves adding support for accessing OME meta-data in the MATLAB interactive environment and further optimizations to lower OME analysis engine and MATLAB Handler over-head.

8. REFERENCES

[1] N. Orlov, J. Johnston, C. Wolkow, and I.G. Goldberg, "Image Similarities in Classification problems for Microscopy Applications" *Submitted to 2006 IEE International Symposium on Biomedical Imaging*.

[2] I.G. Goldberg, et al., "The Open Microscopy Environment (OME) Data Model and XML File: Open Tools for Informatics and Quantitative Analysis in Biological Imaging." *Genome Biology*, Biomed Central, London, pp. 6:R47, 2005.

[3] J. Swedlow, et al.,"Informatics and quantitative analysis in biological imaging." *Science* **300,** 100 –102, 2003,

[4] http://www.openmicroscopy.org/api/xml/

[5] R. Bose and J. Frew, "Lineage Retrieval for Scientific Data Processing: A Survey." *ACM Computing Surveys*, 37(1), 1-28, 2005

[6] http://www.openmicroscopy.org/api/omeis/

[7] http://www.openmicroscopy.org/XMLschemas/AnalysisModule/latest/AnalysisModule_xsd/

[8] S. Martin, M.M. Hohman, T. Liefeld, "The Impact of Life Science Identifier on informatics data." *Drug Discovery Today*, 10(22):1566-72, 2005

[9] http://www.openmicroscopy.org/api/xml/AML/MATLAB.html

[10] T. Liefeld, et al., "GeneCruiser: a web service for the annotation of microarray data", *Bioinformatics*, 21(18): 3681-2, 2005