Sparx

NAMES of AUTHORS

March 27, 2006

Contents

List of Tables

Chapter 1

Introduction

This is an introduction to SPARX.

1.1 Abstract

SPARX is ....

1.2 Goals

The goals of SPARX are ....

1.3 Adding Documentation

This documentation for SPARX is made using LaTeX (see the LaTeX homepage for more information). There are two files, *sparx.tex* and *TOC.tex*, which respectively define the basic the layout of the entire document and the order in which sections of the document are included.

When adding new documentation to this existing document, no preamble tags should be included in the new sections. Preamble information can removed either manually or using the python script rm_preamble.py (see 1.3.1 for more information).

The recommended steps for adding new documentation go as follows:

1. Create a new stand alone LaTeXdocument
2. Remove the preamble information from the new file (1.3.1)
3. Edit TOC.tex (1.3.2)
4. Run "make" to create the new documentation file

1.3.1 Removing Preambles

The preamble of a LaTeX document is used to specify various global parameters of the document. In a stand alone document, all commands coming before and including \begin{document} are part of the preamble. There is also a \end{document} at the very end of the document that should be removed.

Use the "%" character at the start of a line to comment it out.

The python script rm_preamble.py can usually be used to automatically remove the preamble and other tags that should not be used.

```
% python rm_preamble.py file1 file2 ...
```

The output will be placed in the *file1*.tex, *file2*.tex, ...

1.3.2 Editing TOC.tex

TOC.tex contains the table of contents information for this document. Add "\input{*new file name*}" in the appropriate chapter to insert a new documentation file.

A new chapter can be added using the "\chapter{*new chapter name*}" command.

Chapter 2

Golem

2.1 GOLEM documentation

Documentation of the GOLEM/EMAN2 system and contributions of the UTH group.

- **How to use GOLEM**
- **Manual**
- **Information for programmers**

2.2 GOLEM manual

Commands in GOLEM are written on one of three possible levels:

1. Level 1 - commands that directly wrap C++ code. These are usually simple image processing commands.

2. Level 2 - commands written as a mixture of python and Level 1 commands. They perform image processing tasks that require a number of distinct steps, such as 3-D reconstruction from projections.

3. Level 3 - image processing tasks written as python scripts. They perform EM-related tasks, such as 2-D alignment.

Classes of Level 1 and 2 commands:

- FUNDAMENTALS
  FFT, periodogram (power spectrum), cross-correlation, convolution, self-correlation
  center of gravity,

- STATISTICS
  FRC/FSC, moments, entropy, center of gravity (again?!), SSNR, principal component analysis, t-test, Mahalanobis distance, correlation
  coeff, clustering, discriminant analysis, power spectrum

- FILTER
  FOURIER: top hat, Gaussian, Butterworth, hyperbolic tangent
  SPATIAL: box convolution
  NONLINEAR:

- MORPHOLOGY
  dilation, erosion, skeleton, edge detection

Clases of Level 3 scripts:

- UTILITIES
  calculate average and variance files for an image series,

- 2-D ALIGNMENT

- 3-D PROJECTION ALIGNMENT

2.3 Developing with EMAN2

Contents

2.3.1 Introduction

The considerable flexibility of the EMAN2 library comes at a bit of a price: adding new functionality to EMAN2 can involve a lot of different steps.

In the above figure, the items on the left roughly correspond to code, while the items on the right roughly correspond to the libraries generated from the code. Developers working only within the Python layer need only worry about writing Python scripts and learning how to incorporate them into the EMAN2 python module, EMAN2.py. (Additionally, even these "high-level" developers might need to know a bit about Cmake, the EMAN2 packaging program which is used to build and install EMAN2, and Epydoc, which, along with Doxygen for C++, is used to automatically build programming-level documentation for the libraries.

Developers who work on the C++ layer, however, need to understand nearly all of this diagram. The following sections will go into this process in much more detail, but here is a brief outline of the process:

1. Write C++ code (in libEM/) that does something new or better. Documentation for this code should exist in the header (.h) file.
2. Write (or modify an existing) .pyste file (in libpyEM/), if needed, and run the EMAN2 Pyste build script to generate the corresponding libpyXXX.cpp boost.python source file. When the libpyXXX.cpp file is compiled it will generate the necessary Python wrapper to allow the C++ code to be used from Python.

3. Rebuild and reinstall EMAN2.
4. Test to make sure that the new code works.
5. Regenerate the documentation using Doxygen (and Epydoc if necessary).
6. Add new documentation to the Golem documentation.

2.3.2 Writing standalone Python programs

Writing a standalone EMAN2 program in Python is fairly simple, as long as you can figure out what functions/classes/methods to use (which is not so easy, since a lot of documentation is still lacking). Here's an overly simple example that prints out information about an image:

```
#!/usr/bin/env python

from EMAN2 import *
from sys import argv

info(argv[1])
```

The first two lines are going to be in every Python program that uses the EMAN2 library ("boilerplate"). The first line tells a unix-based system what program interpreter to use if the file is executed. (Don't forget to change the filemode with chmod +x filename first!) The from EMAN2 import * line makes all of the EMAN2 componenets accessible in the rest of the script. (For the third and fourth lines, see any decent book on Python and the Golem documentation, respectively.)

The script can be run with:

```
/path/to/imginfo.py /path/to/image_file
```

To have this script be included as one of the programs that gets installed during EMAN2 installation, one need only add the script to the programs/ directory. (Taking a look at the CMakeLists.txt file in that directory, it is reasonably intuitive that any file that ends in .py will be installed.)

2.3.3 Writing standalone C++ programs

Although likely to be uncommon, there may be times that you want to write a C++ main program that calls the C++ portion of the EMAN2 library directly, perhaps for debugging purposes.

The above program is a C++ version of the imginfo.py program from the previous section. To compile the program:

```
$ g++ -I$HOME/EMAN2/include -L$HOME/EMAN2/lib -o imginfo \
> imginfo.cpp -lEM2 -lm
```

Notice that the EMAN2 include and lib directories need to be specified, as do the EMAN2 (EM2) and math libraries. It is a good idea to check any EMAN2 header files that you #include to see if the file has any conditional compiler directives (such as which FFT library to use, as a specific example), and then include any desired directives in the compilation command.

Right now the EMAN2 framework lacks any obvious infrastructure for including compiled C++ main programs automatically, which means that to do so one would need to add the appropriate invocations to the CMakeLists.txt file. Doing so is non-trivial, and thus not covered here.

2.3.4 Adding to EMAN2's Python library

Adding a function or a class to EMAN2's python library is likely to be quite common, and it is fairly easy to do. In general, you want to add your function to the Golem module in libpyEM/Golem.

The first step is to decide which file in the Golem module your function belongs. See the Golem documentation for help there. For example, a wrapper for a circulant cross-correlation function:

```
def ccf(e, f):
  """
  Return the circulant cross-correlation function of images e and f.
  Input images may be real or complex.  Output image is real.
  1-D, 2-D, or 3-D images supported.
  """
  o = correlation(e, f, fp_flag.CIRCULANT)
  return o
```

is placed into the fundamentals.py file. The information in triple-quotes is called the "doc string", and it provides simple documentation for the function that can be accessed from the interactive environment by typing the name of the function followed by a "?", so in this case:

```
In [1]:ccf?
Type:         function
```

```
Base Class:   <type 'function'>
String Form:  <function ccf at 0x2aaaaab29938>
Namespace:    Interactive
File:         /home/grant/EMAN2/lib/Golem/fundamentals.py
Definition:   ccf(e, f)
Docstring:
  Return the circulant cross-correlation function of images e and f.
  Input images may be real or complex.  Output image is real.
  1-D, 2-D, or 3-D images supported.
```

Please make sure that you add a docstring for any functions you add. (Unfortunately, functions/classes/methods that are really C++ code with boost.python providing the wrapper lack docstrings, but hopefully that will get fixed eventually.)

Assuming that code was only added to exising files, then nothing more than a make install is needed in the build directory to make the new functionality accessible.

If a new .py file _is_ added to the Golem module, then it will still be installed automatically with make install in the build directory, but one additional step is needed first. The __init__.py file is used to expose all of the function names in the module, so if you add foo.py to the Golem module then you should also add a line like:

```
from foo import *
```

to __init__.py.

2.3.5 Adding to EMAN2's C++ library

Anything that needs to touch individual pixels/voxels of an image had better be written in C++, otherwise it's going to be awfully slow.

Adding a simple utility

A relatively simple case is adding a quadratic interpolation utility function. Again, the first step is to decide where to put it, and this time the place to look is the EMAN2 autogenerated documentation, which you can read via firefox $HOME/EMAN2/src/eman2/doc/html/index.html. (If you can't find that file, run makedoc.sh in $HOME/EMAN2/src/eman2.) This documentation may be helpful, or it may not be. For example, should the quadratic interpolation utility be in util.cpp, or emutil.cpp? I chose util.cpp, but the choice wasn't clear. In any event, create the C++ code (in libEM/util.cpp, in this case):

```
float Util::quadri(float x, float y, int nxdata, int nydata,
           EMData* image, int zslice) {
  // sanity check
  if (image->get_ysize() <= 1) {
      throw ImageDimensionException("Interpolated image must be at least 2D");
  }
  MArray3D fdata = image->get_3dview(1,1,1);
  // periodic boundary conditions
  if (x < 1.0)
      x += (1-int(x)/nxdata)*nxdata;
  if (x > float(nxdata) + 0.5)
      x = fmodf(x - 1.0f, (float)nxdata) + 1.0f;
  if (y < 1.0)
      y += (1 - int(y)/nydata)*nydata;
  if (y > float(nydata) + 0.5f)
      y = fmodf(y - 1.0f, (float)nydata) + 1.0f;
  int i = int(x);
  int j = int(y);
  float dx0 = x - i;
  float dy0 = y - j;
  int ip1 = (i + 1) % nxdata + 1; // enforce ip1 in [1, nxdata]
  int im1 = (i - 1) % nxdata + 1;
  int jp1 = (j + 1) % nydata + 1;
  int jm1 = (j - 1) % nydata + 1;
  float f0 = fdata[i][j][zslice];
  float c1 = fdata[ip1][j][zslice] - f0;
  float c2 = (c1 - f0 + fdata[im1][j][zslice])*.5f;
  float c3 = fdata[i][jp1][zslice] - f0;
  float c4 = (c3 - f0 + fdata[i][jm1][zslice])*.5f;
  float dxb = dx0 - 1;
  float dyb = dy0 - 1;
  // hxc and hyc are either +1 or -1
  float hxc = (float)((dx0 >= 0) ? 1 : -1);
  float hyc = (float)((dy0 >= 0) ? 1 : -1);
  int ic = int(fmodf(i + hxc, float(nxdata)) + 1);
  int jc = int(fmodf(j + hyc, float(nydata)) + 1);
  float c5 = (fdata[ic][jc][zslice] - f0 - hxc*c1
              - (hxc*(hxc - 1.0f))*c2 - hyc*c3
              - (hyc*(hyc - 1.0f))*c4) * (hxc*hyc);
  float result = f0 + dx0*(c1 + dxb*c2 + dy0*c5)
               + dy0*(c3 + dyb*c4);
  return result;
```

Anything that isn't obvious to an average programmer, assuming that said average programmer has the necessary equations handy, should be commented. Actual usage documentation, however, belongs in the header (libEM/util.h here) file:

```
static float triquad(double r, double s, double t, float f[]);

  /** Quadratic interpolation (2D).
   *
   *  Note:  This routine starts counting from 1, not 0!
   *
   *    This routine uses six image points for interpolation:
   *
   *@see \url{http://www.cl.cam.ac.uk/users/nad/pubs/quad.pdf}
   *
   *@verbatim
             f3    fc
             |
             | x
      f2-----f0----f1
             |
             |
             f4
   *@endverbatim
   *
   *    f0 - f4 are image values near the interpolated point X.
   *    f0 is the interior mesh point nearest x.
   *
   *@li        f0 = (x0, y0)
   *@li        f1 = (xb, y0)
   *@li        f2 = (xa, y0)
   *@li        f3 = (x0, yb)
   *@li        f4 = (x0, ya)
   *@li        fc = (xc, yc)
   *
   *    Mesh spacings:
   *@li              hxa -- x- mesh spacing to the left of f0
   *@li              hxb -- x- mesh spacing to the right of f0
   *@li              hyb -- y- mesh spacing above f0
   *@li              hya -- y- mesh spacing below f0
   *
   *    Interpolant:
   *      f = f0 + c1*(x-x0) + c2*(x-x0)*(x-x1)
```

```
         *              + c3*(y-y0) + c4*(y-y0)*(y-y1)
         *              + c5*(x-x0)*(y-y0)
         *
         *    @param[in] x x-coord value
         *    @param[in] y y-coord value
         *    @param[in] nxdata Size of image along x.
         *    @param[in] nydata Size of image along y.
         *    @param[in] image Image object (pointer)
         *    @param[in] zslice Which z slice to interpolate (counting starts at 1)
         *
         *    @return Interpolated value
         */
        static float quadri(float x, float y, int nxdata,
                            int nydata, EMData* image, int zslice = 1);
};
```

A couple things to note: First, the Util::quadri function is really a static method of the Util class, not an ordinary function, and that's by request of the Baylor folks. The second thing to note is the extensive commenting before the method declaration, and that it starts with /** and ends with */. That javadoc-style commenting is used by Doxygen to autogenerate the EMAN2 documentation. Please write detailed documentation, and especially include a reference (the @see statement above) for the algorithm being used.

Type make in the build directory to see if the new stuff compiles. Assuming that it does, then one needs to decide if the new stuff should be exposed to the Python layer (and it probably should). Look in libpyEM/ for the appropriate .pyste file (utils.pyste, in this case). If the new stuff that has been added returns a reference or a pointer (Util::quadri does not), then the .pyste file will need to be edited. In this case, however, the .pyste file just needs to be used by pyste to generate the boost.python C++ wrapper file. The libpyEM/create_boost_python script does this, but it has to be run explicitly. Once the new boost.python wrapper (libpyUtils2.cpp here) is generated, make && make install in the build directory will install the python wrappers for the new stuff.

As an aside, it may occasionally be useful to write small test objects in c++ and then use pyste and boost.python to create a loadable python module. If foo.cpp is the c++ code, and foo.pyste contains the pyste instructions, then the loadable module can be created by:

```
$ pyste.py -I/usr/include/python2.4 --module=libpyfoo foo.pyste
$ g++ -fPIC -shared -I. -I/usr/include/python2.4 -o libpyfoo.so \
> foo.cpp -lboost_python -lpython2.4
```

although you should use the correct version number for Python for your system (if it is not 2.4), and you might well need to link to additional libraries.

Adding a polymorphic (object-oriented) method

Filters (processors), reconstructors, projectors, etcetera are defined as part of an object-oriented heirarchy and used via a factory design pattern. We'll consider the concrete example of a linear ramp filter (RampProcessor), but the actual process is fairly general.

The basic idea about the polymorphic business is that code that is common to many different flavors of something should exist in a base class, while the more specific code exists in specialized classes. For the linear ramp filter, the base class is the Processor class, which implements a variety of methods (functions), but expects the specialized classes to implement New, get_name, get_desc, get_param_types, and process functions, so that's what RampProcessor provides (except for get_param_types, since the ramp filter doesn't take any parameters). Figuring out exactly where new functionality should fall in the heirarchy of classes is a highly inexact science, unfortunately.

To complicate matters, the expectation is that users will not use Processors directly, but instead they are invoked through the EMData::process method. (Add the factory stuff that needs to be changed, along with other stuff, but I'm out of energy.)

Debugging with gdb

Debugging a pure compiled C++ code is reasonably straightforward, but how does one debug C++ code that is called from Python? The first step is to make sure that both Python and EMAN2 are compiled with debugging symbols (-g), and preferably with no optimization (-O0). Once that's done, then start gdb with Python as the object file:

```
$ gdb /usr/bin/python
```

Now, even though gdb won't know about it, set a breakpoint in whatever source file needs to be debugged:

```
(gdb) br emdata.cpp:39
(gdb) run /path/to/python/script
```

and run the python script (or the path to the sparx script for an interactive session). It works surprisingly well, except for the fact that boost multi_arrays (which are created through fancy C++ templates) are nearly impossible to access from within gdb.

2.3.6 CVS – Adding (or obtaining) code to the official repository

Here's the simple version. To check out the latest code:

```
$ cd $HOME/EMAN2/src/eman2
$ cvs up -dP
```

To add a new file:

```
$ cvs add filename
```

To check in a single file:

```
$ cvs ci -m "message about what's changed" filename
```

To check in all the new stuff:

```
$ cd $HOME/EMAN2/src/eman2
$ cvs ci -m "message about what's changed"
```

A fairly good CVS tutorial is the Gentoo CVS tutorial, although the short section on how to install CVS should be ignored.

FUNDAMENTALS

Level 1 commands

- welch_pw2()

Purpose: Calculate power spectrum of a micrograph using Welch's overlapped strategy.
Usage: welch_pw2(image,win_size,overlp_x,overlp_y,edge_x,edge_y)

ramp()
Purpose: Fits a least-squares plane to the picture, and subtracts the plane from the picture. A wedge-shaped
overall density profile can thus be removed from the picture.
Usage: ramp(image)
window2d()
Purpose: window a section from a micrograph.
Usage: window2d(inputimage,ix,iy,isize_x,isize_y)

Index