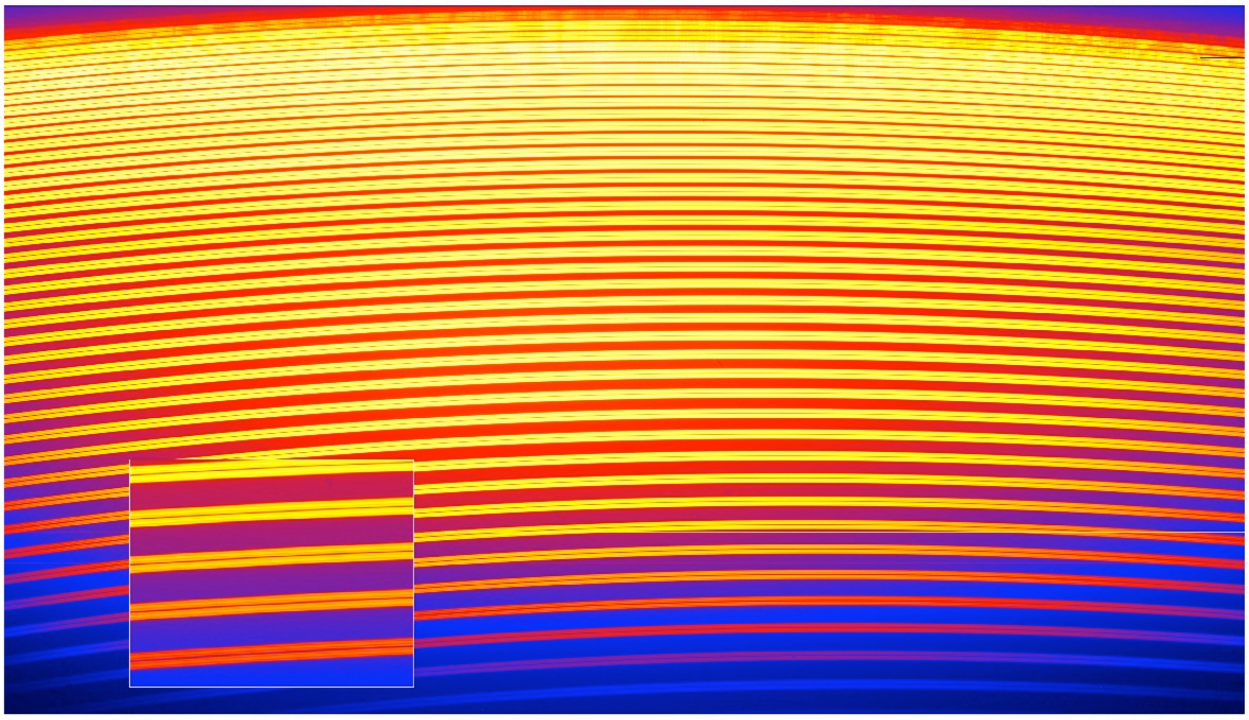
Canada-France-Hawaii Telescope Corporation
65-1238 Mamalahoa Hwy, Kamuela, Hawaii 96743 USA

Société du Télescope Canada-France-Hawaii
Telephone (808) 885-7944 Fax (808) 885 -7288

OPERA PIPELINE PROJECT

OPERA Image Classes and Operators

Tutorial



DOUG TEEPLE
Canada France Hawaii Telescope
Waimea, HI October 2012

Contents

1. Introduction

This document shows readers, astronomers or software developers, how to use the major classes and operators developed as part of the OPERA project. These classes and operators allow developers to manipulate astronomy images at a very high level - equivalent to languages such as Python or IDL, but unlike these interpreted languages, C++ is compiled and thus all operations are very fast. While vector and matrix operations resolve to C library calls in these interpreted languages, other operations such as indexing are very slow. Additionally, the advantages of object orientation disappear when dropping down from an object-oriented language to a C library call. In the case of C++ with these classes and operators, every operation including indexing, is inlined - hence fast ... and the benefits of object orientation are retained.

2. Major OPERA Image Classes

FITS (Flexible Image Transport System) is widely used in astronomy. Four main classes have been developed to handle the various types of FITS images:

- operaFITSImage - a container for simple FITS-format images
- operaMultiExtensionFITSImage - an extension of operaFITSImage that handles the FITS MEF format. MEF is commonly used to handle multiple-chip camera images.
- operaFITSCube - an extension of operaFITSImage that handles the cube format (such as a DSLR camera image). Cubes are commonly used to store sequences of images or sequences of filters.
- operaMultiExtensionFITSCube - an extension of operaFITSImage that handles the mixed FITS MEF cube format.

2.1 The operaFITSImage Class

An operaFITSImage is most easily created with a simple constructor:

```
operaFITSImage anImage();
```

Alternately the constructor can take a filename and a desired image type for computation on the image object:

```
operaFITSImage in(“foo.fits”, tfloat);
```

Here, an instance of the class operaFITSImage is created from a file. The headers and the pixels are read. The pixels are requested to be of type float, so no matter what the actual file contains, the class contains the image converted to floating point numbers.

You may also create in-memory operaFITSImages:

```
operaFITSImage out(“out.fits”,
    in.getnaxis1(),
    in.getnaxis2(),
    tfloat);
```

There are other useful methods such as:

```
out.operaFITSImageSave();
out.operaFITSImageSaveAs(“foocopy.fits”);
out.operaFITSImageClose();
```

The examples below show use of the C++ feature called operator overloading. You cannot create new operators in C++ but you can define a meaning of existing operators specific to a given class. OPERA defines operators for each of the major classes that have a specific meaning in the context of images. For example, the assignment operator “=” is overloaded:

```
out = 0.0;    // zero the whole image
```

In this case, the assignment operator of a float is defined to mean: set every pixel in the image to the floating point value zero.

```
out = in;     // copy the pixels from in to out
```

Here the pixels are copied from one image to another.

The operators +=, -=, *=, /= also have meaning:

```
flat -= bias;               // remove the bias from the flat
out /= flat;                // divide image by the flat
out *= badPixelMask;        // mask out bad pixels
out -= 300.0;               // remove a constant bias from out
```

The binary operators +, /, *, - also have meaning:

```
// out is the pixel-by-pixel difference between the flats
out = flat1 - flat2;
// mask out with the badpixel mask
out = out * badpixelImage;
```

The operators <, >, <=, >= also have meaning:

```
out *= out > 0.0;    // zero any negative values
```

Here `out` > 0.0 creates an in-memory pixel map of 1.0’s and 0.0’s of the dimensionality of “out”, where there is a 1.0 wherever “out” has a positive pixel values, else 0.0. This map is then multiplied by `out`, effectively masking all negative pixels from `out`.

Take care that this expression:

```
out = out > 0.0;     // save the mask in out
```

is very different, because here “out” takes the value of the mask itself, becoming all 1.0’s and 0.0’s.

Note that binary operators create in-memory temporary images that are reclaimed after use automatically by the operator definition.

It is also possible to get the value of a pixel by using the [] operator:

```
// print some pixel values from the image
// Note that y is the first dimension
for (unsigned y=0; y<2048; y++) {
     for (unsigned x=0; x<2048; x++) {
          cout << out[y][x] << ' ';     // print the pixel value
          out[y][x] = 0.0;              // zero the pixel value
     }
     cout << endl;
}
```

While most operators allow the user to operate on entire images, it is also possible to index to particular pixel positions using the [] operator both in assignment and referencing.

There is also an ImageIndexVector type defined that is created by the “where” function, which has similar semantics to the “where” function in IDL:

```
// store the null-terminated vector of indices into vector
// ImageIndexVectors are one-based image positions
ImageIndexVector vector = where(flatImage>0.0);
```

An ImageIndexVector is defined as a null-terminated, one-based vector of image indices, where the image is treated as being single dimensioned.

The “where” function is useful for creating bins of image pixel values:

```
// mask pixels in a range of ADU values
out[where(in >= 400.0 && in <= 800.0)] = -1.0;
```

This example introduces a number of concepts.

Firstly the II (or) and && (and) operators are defined for operaFITSImages. The meaning assigned to && is: for every pixel if the either the corresponding lhs is zero or the rhs is zero, then 0.0, else 1.0. The operator II means: for every pixel if the either the corresponding lhs is non-zero or the rhs is non-zero, then 1.0, else 0.0.

Secondly, the “where” function takes an operaFitsImage as an argument and returns a null-terminated, one-based vector of operaFITSImage *indices* where the operaFITSImage is non-zero.

Thirdly, a operaFITSImage can be multiply indexed from another image.

The unary ! operator is also defined. It has the same meaning as ! in C++, except that it creates a map of every pixel.

```
// mask anImage with the inverse of badpixel mask
out *= !badPixelMask;
```

Here is an interesting snippet of executable C++ code that creates an operaFITSImage, removes the bias image, divides by a flat image, masks bad pixels, finds the maximum pixel value and normalizes the entire image!

```
operaFITSImage in(“in.fits”);
operaFITSImage out(“out.fits”, tfloat);
out = in;                           // copy the input image pixels
out -= bias;                        // remove the bias from the image
out /= flat;                        // divide the image by the flat
out *= badPixelMask;                // mask with the badpixel mask
float maxvalue = anImage.max();     // find the maximum pixel value
out /= maxvalue;                    // normalize the image
```

The result is fast inlined executable code, compiled directly from C++. How fast is it? There is a small test program in opera-1.0/test called operaAsmTest.cpp.

```
void foo(operaFITSImage *a, operaFITSImage *b) {
        *a -= *b;
}
int main()
{
        operaFITSImage a("a.fits");
        operaFITSImage b("b.fits");
        return 0;
}
```

Image b (say a bias) is deleted pixel by pixel from image a. The operation is isolated in function foo for convenience. The machine instructions produced by gcc are:

```
void foo(operaFITSImage *a, operaFITSImage *b) {
          0:     55                         push   %ebp
          1:     89 e5                      mov    %esp,%ebp
          3:     56                         push   %esi
          4:     53                         push   %ebx
          5:     83 ec 10                   sub    $0x10,%esp
          8:     8b 45 08                   mov    0x8(%ebp),%eax
b:        8b 75 0c                                  mov    0xc(%ebp),%esi

          operaFITSImage& operator-=(operaFITSImage& b) {
                 float *p = (float *)pixptr;
                 float *bp = (float *)b.pixptr;
                 unsigned n = npixels;
          e:     8b 50 2c                   mov    0x2c(%eax),%edx
                 operaFITSImage& operator-=(operaFITSImage& b) {
                         float *p = (float *)pixptr;
                         11:     8b 48 3c                   mov    0x3c(%eax),%ecx
                         float *bp = (float *)b.pixptr;
                         14:     8b 5e 3c                   mov    0x3c(%esi),%ebx
                         unsigned n = npixels;
                         while (n--) *p++ -= *bp++;
                         17:     85 d2                      test   %edx,%edx

                         19:     74 16                      je     31 <_Z3fooP14operaFITSImageS0_+0x31>
                         1b:     31 c0                      xor    %eax,%eax
                         1d:     8d 76 00                   lea    0x0(%esi),%esi
                         20:     d9 04 01                   flds   (%ecx,%eax,1)      <-- top of assign loop
                         23:     d8 24 03                   fsubs  (%ebx,%eax,1)      | 2
                         26:     d9 1c 01                   fstps  (%ecx,%eax,1)      | 3
                         29:     83 c0 04                   add    $0x4,%eax          | 4
                         2c:     83 ea 01                   sub    $0x1,%edx          | 5
                         2f:     75 ef                      jne    20                 <-- bottom of assign loop
                         if (b.istemp) delete &b;
                         31:     80 7e 38 00                cmpb   $0x0,0x38(%esi)
                         35:     74 19                      je     50 <_Z3fooP14operaFITSImageS0_+0x50>
                         37:     89 34 24                   mov    %esi,(%esp)
                         3a:     e8 fc ff ff ff             call   3b <_Z3fooP14operaFITSImageS0_+0x3b>
                         3f:     89 75 08                   mov    %esi,0x8(%ebp)
                         *a -= *b;
                 }
                 42:     83 c4 10                   add    $0x10,%esp
                 45:     5b                         pop    %ebx
                 46:     5e                         pop    %esi
                 47:     5d                         pop    %ebp
                 48:     e9 fc ff ff ff             jmp    49 <_Z3fooP14operaFITSImageS0_+0x49>
                 4d:     8d 76 00                   lea    0x0(%esi),%esi
```

```
50:      83 c4 10                  add    $0x10,%esp
53:      5b                        pop    %ebx
54:      5e                        pop    %esi
55:      5d                        pop    %ebp
56:      c3                        ret
57:      89 f6                     mov    %esi,%esi
59:      8d bc 27 00 00 00 00      lea    0x0(%edi,%eiz,1),%edi
```

You can see that the inner loop where the pixels are processed consists of just 6 instructions! Look for the annotation top of assign loop and bottom of assign loop. These instructions are executed NAXIS1 * NAXIS2 times. The function entry and exit instructions are executed only once. It would be difficult to express this high level image subtraction operation more efficiently, even if coding directly in assembly language.

2.2 The operaMultiExtensionFITSImage Class

The operaMultiExtensionFITSImage class encapsulates the FITS notion of a FITS Image that contains multiple images. Ordinarily this would be used to store images with more than one detector chip:

```
operaMultiExtensionFITSImage in("foo.fits", tfloat);
```

Here a "MEF" image object is created.

All of the operators defined for operaFITSImages also work for operaMultiExtensionFITSImage, but the indexing operator [] has a special meaning:

```
operaFITSImage out(in[4]);
```

In this example a regular FITSImage object is created from extension 4 of the MEF FITS image. The definition of [] for the operaMultiExtensionFITSImage class carries the meaning of indexing into an extension, so to access pixels in an image:

```
unsigned lastextension = in.getNExtensions();
for (unsigned extension=1; extension<=lastextension; extension++) {
     cout << in[extension][1024][1024] << endl;
}
```

The constructor for the operaMultiExtensionFITSImage class adds an additional argument: *isLazy*. The operaMultiExtensionFITSImage supports lazy reads, meaning that an extension is not read from disk until it is referenced and it is not written to disk until it takes on a new value. Only one extension is in memory at any one time. Very large images cannot be stored in entirety in memory on 32 bit architecture hardware. The lazy read permits operations on large images where I/O is largely transparent to the user. For example:

```
operaMultiExtensionFITSImage in(“foo.fits”, tfloat);
operaMultiExtensionFITSImage out("out.fits",
     in.getXDimension(),
     in.getYDimension(),
     inImage.getNExtensions(),
     tfloat);

unsigned lastextension = in.getNExtensions();
for (unsigned extension=1; extension<=lastextension; extension++) {
     out[extension] = in[extension];
}
```

This example is interesting because the operaMultiExtensionFITSImage objects are by default lazy. That means the extension is not read until requested and the extension is not written until assigned. So, in the loop above, first extension 1 of in is read from disk, then the pixels are copied into the out object in memory, and then stored to disk, since out[] was assigned. Then the next iterations of the loop to reads/writes for each extension happen automatically. The user need not be concerned about file I/O, but can rather concentrate on the algorithm.

2.3 The operaMultiExtensionFITSCube Class

The operaMultiExtensionFITSCube extends the operaMultiExtensionFITSImage class to handle the FITS image notion of a FITS Cube. The meaning of the [] operator is defined for this class to index into a slice:

```
unsigned lastextension = in.getNExtensions();
unsigned lastslice = in.getNSlices();
for (unsigned extension=1; extension<=lastextension; extension++) {
     out[extension] = in[extension];
     for (unsigned slice=1; slice<=lastslice; slice++) {
            cout << out[extension][slice][1024][1024] << endl;
     }
}
```

The operaMultiExtensionFITSCube inherits the laziness of the The opera-MultiExtensionFITSImage class. The [] operator does add another layer of granularity of image access. In this example, slices rather than extensions are read/written:

```
operaMultiExtensionFITSCube in(“foo.fits”, tfloat);
operaMultiExtensionFITSCube out("out.fits",
     in.getXDimension(),
     in.getYDimension(),
     in.getZDimension(),
     inImage.getNExtensions(),
     tfloat);

unsigned lastext = in.getNExtensions();
unsigned lastslice = in.getNSlices();
for (unsigned extension=1; extension<=lastext; extension++) {
     for (unsigned slice=1; slice<= lastslice; slice++) {
           out[extension][slice] = in[extension][slice];
     }
}
```

If available memory is very tight, single slices of single extensions may be read/written.

2.4 Casting and Converting Classes

The opera FITS image classes are object-oriented and hierarchical and so may be inter-converted. This is an example of the creation of a Cube from 3 WIRCam images taken with different filters of the same object. The output Cube is similar to the color cube taken by a DSLR camera. This example is abstracted from the “wircolorcomposite” tool that is distributed with OPERA, used to create false color WIRCam images. First, we create the three rgb input image objects:

```
operaWIRCamImage red(r, tfloat, READONLY);
operaWIRCamImage green(g, tfloat, READONLY);
operaWIRCamImage blue(b, tfloat, READONLY);
```

Next, we create the output 3 - Cube and populate the headers from one of the inputs:

```
operaFITSCube out(output,
     red.getXDimension(),
     red.getYDimension(),
     3,
     tfloat);
out.operaFITSImageCopyHeader(&red);
```

Next, we create the output Cube and populate the headers from one of the inputs:

```
// median collapse to get rid of cosmic rays and reduce noise
red.medianCollapse();
green.medianCollapse();
blue.medianCollapse();
```

In this step we create regular fits images of the 4th extension, where the desired image resides:

```
// create FITS images of the 4th extension
operaFITSImage redfits(red[4]);
operaFITSImage greenfits(green[4]);
operaFITSImage bluefits(blue[4]);
```

Now remove the chip bias offset, and mask the guide windows and bad pixels to the image median value (do the same for green and blue):

```
/*
 * 1. remove the chip bias
 * 2. get the image median
 * 3. set the inverse of the badpix to image median
 * 4. mask the guidewindow to image median
 */
redfits -= red.getChipBias();
float imageMedian = operaArrayMedian(redfits.getnpixels(),
     (float *)redfits.getpixels());
*redfits[where(!badpixfits)] = imageMedian;
*redfits[new operaImageVector(red.getGuideWindow(extension),1)] = imageMedian;
...
```

Finally copy the red, the green, and the blue images to the three axes of the output cube and save it:

```
// copy each color into the cube
out[1] = redfits;
out[2] = greenfits;
out[3] = bluefits;

out.operaFITSImageSave();
```

We are done! We manipulated various opera FITS Image classes to create the final Cube product.

2.5 The Matrix

No, not that matrix... The indexing operator [] defined for the operaFITSImage class is useful only to C++ programs. However many of the OPERA image and statistics libraries are written in C. There is a type *Matrix*, type-defined as a float **, that allows C functions to access the FITS images using the [y][x] notation. The matrix base is created every time a class image is created. The first dimension is a vector of length naxis2, which is a vector of float* to the address of each row in the image. It is used thusly:

```
// get a Matrix to pass to a C function
        Matrix matrix = anImage.getmatrix;
             fooInC(matrix);

        void fooInC(float **matrix) {
             for (unsigned y=0; y<maxy; y++) {
                  for (unsigned x=0; x<maxx; x++) {
                       cout << matrix[y][x] << ' ';     // output the pixel value
                  }
                  cout << endl;
             }
        }
```

Here is an example of iteration by column by reversing the for loops in the image:

```
for (unsigned x=1024; x<1032; x++) {
        for (unsigned y=2048; y<2056; y++) {
             matrix[y][x] = matrix[x][y];
        }
}
```

3. OPERA Instrument Image Classes

3.1 The operaEspadonsImage Class

An operaEspadonsImage is an extension of operaFITSImage. It is most easily created by definition, from an existing FITS file:

```
operaEspadonsImage *in = new operaEspadonsImage("foo.fits", tfloat);
```

While an operaEspadonsImage contains extended information unique to espadons images, the most important attribute is that the image pixels are from the datasec, and the overscan pixels are not included in the image. Here are some examples of the extended information:

```
cerr << "IMTYPE= " << in->getimtype() << ' ' << in->getimtypestring() << '\n';
cerr << "DETECTOR= " << in->getdetector() << ' ' << in->getdetectorstring() << '\n';
cerr << "AMPLIFIER= " << in->getamplifier() << ' ' << in->getamplifierstring() << '\n';
cerr << "MODE= " << in->getmode() << ' ' << in->getmodestring() << '\n';
cerr << "SPEED= " << v->getspeed() << ' ' << v->getspeedstring() << '\n';
cerr << "STOKES= " << in->getstokes() << ' ' << in->getstokesstring() << '\n';
cerr << "POLAR QUAD= " << in->getpolarquad() << ' ' << in->getpolarquadstring() << '\n';
```

The mode, speed, detector, enums are defined in the header file and will not be further detailed here.

Note that these examples use an operaEspadonsImage pointer, so access to the operators is a little different:

```
// Note the fairly tricky *Out notation here, as Out is a pointer
*Out -= 100.0;                  // remove bias
*Out *= *Out < 35000.0;         // remove saturated pixels
// print some pixel values from the center of the image
for (unsigned y=2048; y<2056; y++) {
     for (unsigned x=1024; x<1032; x++) {
          cout << (*Out)[y][x] << ' ';   // output the pixel value
          (*Out)[y][x] = 0.0;            // zero the pixel value
     }
}
cout << endl;
```

The datasec portion of the image is stored in the private member datasecSubImage which has a getter:

```
/*! * operaFITSSubImage *getDatasecSubImage()
 * \brief get the datasec subImage
 * \return subImage pointers */
```

```
operaFITSSubImage *getDatasecSubImage();
```

This is the preferred access method to the espadons image data, for example:

```
operaFITSSubImage *subImage = In.getDatasecSubImage();
*subImage += 100.0;                     // add bias
*subImage *= *subImage < 35000.0;       // remove pixels above 35000 ADU
for (unsigned y=2048; y<2056; y++) {
     for (unsigned x=1024; x<1032; x++) {
          cout << (float)(*subImage)[y][x] << '';
     }
     cout << endl;
}
cout << endl;
```

Note that the subImage is a copy of the espadons image, and must be set back into the full espadons image if the subImage pixels are changed:

```
// now, if we ever change the subImage, which is a copy
// we need to set it back in to the espadons image
Out.operaFITSImageSetData(*subImage);
```

3.2 The operaWIRCamImage Class

The operaWIRCamImage class describes an object that has specific WIR-Cam pieces. It is derived from the operaMultiExtensionFITSCube, but adds:

exposure time

chip bias

sky levels

guide cube boxes

There are also a number of methods available for WIRCam images:

```
/*
* void detrend(operaWIRCamImage &image, operaWIRCamImage &dark, operaWIRCamImage &badpixelmask, operaWIRCamImage *flat)
* \brief detrend a WIRCam image. all extensions in parallel
*/
void operaWIRCamImage::detrend(operaWIRCamImage &image, operaWIRCamImage &dark, operaWIRCamImage &badpixelmask, operaWIRCamImage *flat);
/*
* void weightmap(operaWIRCamImage &image, operaWIRCamImage &badpixelmask)
* \brief create a weightmap of bad and saturated pixxels of a WIRCam image. all extensions in parallel
*/
void operaWIRCamImage::weightmap(operaWIRCamImage &image, operaWIRCamImage &badpixelmask);
 /*
 * void skySubtraction(operaWIRCamImage &image, operaWIRCamImage &flat, operaWIRCamImage &dark)
 * \brief sky subtract a WIRCam image
 */
void operaWIRCamImage::skySubtraction(operaWIRCamImage &image, operaWIRCamImage &sky);

 /*
 * void maskGuideWindows(operaWIRCamImage &image)
 * \brief mask the entire rows and columns at the guide window positions in a WIRCam image
 */
```

```
        void operaWIRCamImage::maskGuideWindows(operaWIRCamImage &image);
        /*
        * void masterFlat(operaWIRCamImage &image[], operaFITSImage *weight, count)
        * \brief create a master flat from a medianCombined cube
        * \brief optionally xreating a weight map
        */
        void operaWIRCamImage::masterFlat(operaWIRCamImage *images[], operaWIRCamImage *weight,
unsigned count);
        /*
        * void masterDark(operaWIRCamImage &image)
        * \brief create a master dark by median combining the stack
        */
        void operaWIRCamImage::masterDark(operaWIRCamImage *images[], unsigned count);
        /*
        * void subtractReferencePixel(operaWIRCamImage &image)
        * \brief subtract reference pixels from WIRCam image
        */
        void operaWIRCamImage::subtractReferencePixels(operaWIRCamImage &image);
        /*
        * void calculateSkyBackground(operaWIRCamImage &image)
        * \brief Calculate the Sky Background - the median / exposure time
        * Could also be gotten from the headers SKYLVL, SKYDEV -- if they exist
        */
        void operaWIRCamImage::calculateSkyBackground(operaWIRCamImage &image);

        /*
        * void calculateQERatio(operaWIRCamImage *images[], unsigned count)
        * \brief Calculate the QE Ratio - the median sky rate over the images
        * Could also be gotten from the headers SKYLVL, SKYDEV -- if they exist
        */
        void operaWIRCamImage::calculateQERatio(operaWIRCamImage *images[], unsigned count);
        /*
        * void createSky(operaWIRCamImage *images[], unsigned count)
        * \brief Create a master sky image
        */
        void operaWIRCamImage::createSky(operaWIRCamImage *images[], unsigned count);
        /*
        * void correctSkyLevel(operaWIRCamImage *images[], unsigned count)
        * \brief Correct sky level
        */
        void operaWIRCamImage::correctSkyLevel(operaWIRCamImage *images[], unsigned count);
```

Details may be found in the doxygen documentation.

4. Other OPERA Image Classes of Note

4.1 The operaFluxVector Class

The operaFluxVector class encapsulates vectors of flux, variance pairs. It is intended for use in polarimetry. Operators are defined that operate on whole vectors, element-by-element, most importantly also calculating the resultant variances. so that variances may be propagated through the various vector operations. A test case shows a snippet from polarimetry steps below:

```
const unsigned length = 5;

double i1EFluxes[] = {2000.0, 1450.0, 123.0, 4324.9, 1235.0};
double i1EVariances[] = {0.25, 11.2, 0.2, 9.8, 12.2};
double i1AFluxes[] = {2500.0, 1050.0, 1230.0, 434.9, 235.0};
double i1AVariances[] = {0.25, 1.2, 0.2, 1.8, 2.2};

double i2EFluxes[] = {2400.0, 1650.0, 103.0, 4424.9, 1035.0};
double i2EVariances[] = {0.25, 11.2, 0.2, 9.8, 12.2};
double i2AFluxes[] = {2200.0, 1250.0, 1030.0, 494.9, 225.0};
double i2AVariances[] = {0.25, 1.2, 0.2, 1.8, 2.2};

double i3EFluxes[] = {2200.0, 1430.0, 163.0, 4524.9, 1295.0};
double i3EVariances[] = {0.25, 11.2, 0.2, 9.8, 12.2};
double i3AFluxes[] = {2560.0, 1050.0, 1230.0, 434.9, 235.0};
double i3AVariances[] = {0.25, 1.2, 0.2, 1.8, 2.2};

double i4EFluxes[] = {2300.0, 1420.0, 173.0, 4624.9, 1265.0};
double i4EVariances[] = {0.25, 11.2, 0.2, 9.8, 12.2};
double i4AFluxes[] = {2520.0, 1550.0, 1238.0, 404.9, 205.0};
double i4AVariances[] = {0.25, 1.2, 0.2, 1.8, 2.2};

/*
 * Populate the E/A vectors
 */
operaFluxVector i1E(i1EFluxes, i1EVariances, length);
operaFluxVector i1A(i1AFluxes, i1AVariances, length);
operaFluxVector i2E(i2EFluxes, i2EVariances, length);
operaFluxVector i2A(i2AFluxes, i2AVariances, length);
operaFluxVector i3E(i3EFluxes, i3EVariances, length);
operaFluxVector i3A(i3AFluxes, i3AVariances, length);
operaFluxVector i4E(i4EFluxes, i4EVariances, length);
operaFluxVector i4A(i4AFluxes, i4AVariances, length);
/*
 * Populate the rn vectors with the E data
 */
operaFluxVector r1(length);
operaFluxVector r2(length);
operaFluxVector r3(length);
operaFluxVector r4(length);
/*
 * STEP 1 - calculate ratio of beams for each exposure
 * r1 = i1E / i1A
 * r2 = i2E / i2A
 * r3 = i3E / i3A
 * r4 = i4E / i4A
 */
r1 = i1E / i1A;
r2 = i2E / i2A;
r3 = i3E / i3A;
r4 = i4E / i4A;
/*
 * STEP 2 - calculate the quantity R (Eq #2 on page 663 of paper)
 *   R^4 = (r1 * r4) / (r2 * r3)
 *   R = the 4th root of R^4
 */
operaFluxVector R(length);
R = Sqrt(Sqrt((r1 * r4) / (r2 * r3)));
/*
 * Now print out the flux (first) and variance (second)
 */
printf("r1=");
for (unsigned i=0; i<length; i++) {
     printf("%.2f %.2f, ", r1[i]->first, r1[i]->second);
}
printf("\n");
/*
 * Finally print out the errors
 */
printf("R errors=");
for (unsigned i=0; i<length; i++) {
     printf("%.2f, ", R.geterror(i));
}
printf("\n");
```

Note that as the calculations are done, variances are propagated to the final product. The fluxvector class stores both flux and variance for each point in the vector. The operators are defined in such a way as to propagate the variances. The error can be obtained at any point using the "geterror" function call.

The user of the fluxvector class need not know the details of the fluxvector class, but just how to use it. For tutorial purposes, to help understand the concept, an example of one definition in the class is given.

As an example, the operator "=" is defined as:

```
operaFluxVector& operator=(operaFluxVector& b) {
        double *afluxes = (double *)fluxes;
        double *bfluxes = (double *)b.fluxes;
        double *avariances = (double *)variances;
        double *bvariances = (double *)b.variances;
        unsigned n = length;
        while (n--) { *afluxes++ = *bfluxes++; *avariances++ = *bvariances++; }
        if (b.istemp) delete &b;
              return *this;
};
```

Note that both the flux and variance are retained. Variance is propagated by the mathematical rules of variance propagation. Each operator uses the statistical definition of variance propagation relevant to the particular operator.

The propagation of variances is defined for the operaFluxVector class according to the statistical definition. We will not go in to the entire set of definitions for all operators, the interested reader may consult the doxygen documentation for that level of detail. As an example, this example shows the definition for the operator “*” - multiply two flux vectors

```
/*!
 * \brief operator *
 * \brief multiply operator.
 * \param b - operaFluxVector*
 * \note usage: operaFluxVector a = operaFluxVector b * operaFluxVector c; multiplies the fluxes
and error values  b * c and assigns to a
 * \return operaFluxVector&
 */
operaFluxVector& operator*(operaFluxVector* b) {
        double *tfluxes, *tvariances;
        operaFluxVector *t = NULL;
        if (this->istemp) {
              t = this;
              tfluxes = (double *)this->fluxes;
              tvariances = (double *)this->variances;              } else {
              t = new operaFluxVector(*this, true);
              tfluxes = (double *)t->fluxes;
              tvariances = (double *)t->variances;
        }
        double *afluxes = (double *)this->fluxes;
        double *bfluxes = (double *)b->fluxes;
        double *avariances = (double *)this->variances;
        double *bvariances = (double *)b->variances;
        unsigned n = length;
        while (n--) { *tfluxes++ = *afluxes * *bfluxes; *tvariances++ = sqrt( pow(*afluxes *
*bvariances++,2) + pow(*bfluxes * *avariances++,2) ); afluxes++; bfluxes++; }
        if (b->istemp) delete b;
        return *t;
};
```

The relevant line is the “while” statement, where you can see the formula used to propagate variance through multiplication. Each fluxvector operator is a binary operator operating on a left hand and a right hand side. Complex expressions are built up pairwise according to C++ operator precedence rules. note also that we relieve the user of the necessity of memory management. Although complex expressions may generate temporaries, the user of the operator need not be aware of this complication.

The function “geterror” is defined as:

```
double geterror(unsigned index) {
        return fluxes[index]/sqrt(variances[index]);
};
```

An additional feature is the handling of infinities. Algebraically, these two cases produce different results when the numerator and denominator have the value infinity:

```
PoverI = (R - 1.0) / (R + 1.0);
PoverI = (R*R - 1.0) / (R*R + 1.0);
```

Case one produces the value 1.0 and case two produces the value 0.0, in the case of two infinities at a point in the vector. The fluxvector class does not do algebra, but it does allow the user to control the effect of division by infinities. The vector can be declared with a TendTowards_t parameter:

```
operaFluxVector R(length,ToOne);
operaFluxVector RR(length,ToZero);
```

TendsTowards_t is defined in the operafluxvector header:

```
/*!
 * \brief Definition of the value of each "tends towards" optional field.
 * \details The possible values of the enum are : ToDefault, ToINF, ToNAN, ToZero, ToOne
 */
typedef enum {ToDefault, ToINF, ToNAN, ToZero, ToOne} TendsTowards_t;
```

Correspondingly, the operafluxvector division operator implements the appropriate cases:

```
        /*!
         * \brief Division operator.
        * \details The operator divides the elements on the left side of the operator
by the corresponding elements on the right side.
         * \param b An operaFluxVector pointer
         * \note Usage: operaFluxVector t = operaFluxVector a / operaFluxVector b;
         * \return An operaFluxVector address
         */
        operaFluxVector& operator/(operaFluxVector* b) {
             double *tfluxes, *tvariances;
             operaFluxVector *t = NULL;
             if (this->istemp) {
                   t = this;
                   tfluxes = (double *)this->fluxes;
                   tvariances = (double *)this->variances;
             } else {
                   t = new operaFluxVector(*this, towards, true);
                   tfluxes = (double *)t->fluxes;
                   tvariances = (double *)t->variances;
             }
             t->towards = this->towards;
             double *afluxes = (double *)this->fluxes;
             double *bfluxes = (double *)b->fluxes;
             double *avariances = (double *)this->variances;
             double *bvariances = (double *)b->variances;
             unsigned n = length;
             switch (this->towards) {
             case ToDefault:
             while (n--) { *tfluxes++ = *afluxes / *bfluxes; *tvariances++ = (
pow(*bfluxes,-2) * *avariances++ ) + ( pow(*afluxes / pow(*bfluxes,2),2) * *bvari-
ances++ ); afluxes++; bfluxes++;  }
             break;
             case ToINF:
             while (n--) { if (isinf(*afluxes)&&isinf(*bfluxes)) {*tfluxes++ = FP_IN-
FINITE; *tvariances++ = 0.0; afluxes++; bfluxes++; avariances++; bvariances++;} else
{*tfluxes++ = *afluxes / *bfluxes; *tvariances++ = ( pow(*bfluxes,-2) * *avariances++
) + ( pow(*afluxes / pow(*bfluxes,2),2) * *bvariances++ ); afluxes++; bfluxes++; } }
             break;
             case ToNAN:
             while (n--) { if (isinf(*afluxes)&&isinf(*bfluxes)) {*tfluxes++ = FP_NAN;
*tvariances++ = FP_NAN; afluxes++; bfluxes++; avariances++; bvariances++;} else
{*tfluxes++ = *afluxes / *bfluxes; *tvariances++ = ( pow(*bfluxes,-2) * *avariances++
) + ( pow(*afluxes / pow(*bfluxes,2),2) * *bvariances++ ); afluxes++; bfluxes++; } }
             break;
             case ToZero:
             while (n--) { if (isinf(*afluxes)&&isinf(*bfluxes)) {*tfluxes++ = 0.0;
*tvariances++ = 0.0; afluxes++; bfluxes++; avariances++; bvariances++;} else
{*tfluxes++ = *afluxes / *bfluxes; *tvariances++ = ( pow(*bfluxes,-2) * *avariances++
) + ( pow(*afluxes / pow(*bfluxes,2),2) * *bvariances++ ); afluxes++; bfluxes++; } }
             break;
             case ToOne:
             while (n--) { if (isinf(*afluxes)&&isinf(*bfluxes)) {*tfluxes++ = 1.0;
*tvariances++ = 0.0; afluxes++; bfluxes++; avariances++; bvariances++;} else
{*tfluxes++ = *afluxes / *bfluxes; *tvariances++ = ( pow(*bfluxes,-2) * *avariances++
) + ( pow(*afluxes / pow(*bfluxes,2),2) * *bvariances++ ); afluxes++; bfluxes++; } }
             break;
             default:
             break;
             }
             if (b->istemp) delete b;
             return *t;
        };
```

5. Conclusion

This tutorial showed by example how to use the major classes and operators developed for the OPERA image reduction pipeline project. It showed how, using features of the C++ language, it is possible to create fast, efficient programs to manipulate FITS images, doing so at a very high level of abstraction - whole image operations, with the readability of an interpreted language such as Python or IDL, while retaining the speed of a compiled language and the level of abstraction of an object-oriented language.