UNIX-based Image Analyst (UIA)

Written and developed by: Dmitriy Bernsteyn, Suzanne Curry, Craig Holmquist, Robert Pollock, Mark Schmalz, and Matthew Weingarten

Purpose

The purpose of UIA is to be able to analyze and manipulate images from the command line on UNIX systems. No longer are GUI frameworks, which are difficult to work with and/or expensive for the developer, necessary as UIA can do everything they can do in a much more simplistic fashion.

How to Use

UIA relies on shell commands and built-in tools such as ImageMagick for its successful use. If the location of these libraries are different on your machine than they are on the ones on which this was developed, then those lines of code will need to be updated before running the program or else those routines will not run correctly (if they run at all). Make sure that you have checked this and verified that they are right before running for the first time.

Running UIA is done as follows:

```
./img.exe <operation> <arguments>
```

What are the commands, what arguments do they take in, and what do they produce? That is answered below in the following section.

Commands

Setting Up

No matter what command is entered, the initial procedure is the same. First comes argument checking, which deals with making sure that the correct number of arguments are entered for the corresponding command and making sure that they are in a correct format. After that comes image conversion. All images in UIA are converted to Portable Graymap Format (PGM). You can find out more about PGM here.

After converting the image(s) to PGM, the images are read into their double arrays. If two images are used for the operation, it’s necessary to take the common area of those two images so that the operation can overlay them. The resize_img method takes the image where the dimension of the image (row or column) is bigger and shrinks it to the size of the smaller dimension image. Resize_img is currently bugged. I highly suggest using ImageMagick’s convert routine to resize images if necessary for the time being.

When all of that is complete, the individual operations are run.

Operations

imgadd

Takes in: two images
Produces: one output image

imgadd uses the “plus” option in ImageMagick’s *composite* method to add the two images pixel-by-pixel.

imgsub

Takes in: two images
Produces: one output image

imgsub uses the “subtract” option in ImageMagick’s *composite* method to subtract the two images pixel-by-pixel.

imgmul

Takes in: two images
Produces: one output image

imgmul multiplies the two image arrays pixel-by-pixel to produce its output.

imgdiv

Takes in: two images

Produces: one output image

imgdiv divides the two image arrays pixel-by-pixel to produce its output. If a pixel in the second image array is 0, it sets the corresponding pixel in the output image to 0 to avoid a divide-by-zero error.

imgsin

Takes in: one image

Produces: one output image

imgsin runs a sine transformation on the image array. Values are normalized to pi/2 whenever necessary and then further normalized to the range [0,255], which represents the standard grayscale range.

imglog

Takes in: one image

Produces: one output image

imglog runs a log base 10 transformation on the image array. Values are normalized to [0,255] whenever necessary.

gray

Takes in: one image

Produces: one output image

gray turns an image into a grayscale image. You can find out more about grayscale here.

thold

Takes in: one image and a threshold value

Produces: one output image

thold converts a regular image to a monochrome image based on the threshold value. If the image value is greater than the threshold, then the pixel value is set to white and is set to black otherwise. An all-zero array is not allowed.

comp

Takes in: one image
Produces: one output image

comp changes the background of an image to a new color that is determined from other methods.

ftruth

Takes in: two images
Produces: tabular output

ftruth determines the number of correct hits, false alarms, correct misses, and miss detections for two monochrome images. This is represented in a truth table format.

ctruth

Takes in: two images
Produces: text output

ctruth is very similar to *ftruth*, but doesn’t output the truth table that ftruth does. Instead, that output is represented textually.

wht

Takes in: one image
Produces: an image shown using ImageMagick

wht runs the Walsh-Hadamard transformation on an image. The Walsh-Hadamard transform (WHT) uses a divide-and-conquer approach to manipulate an image. It runs in O(nlogn) time and is a generalized form of a discrete fourier transform (DFT).

The algorithm begins by padding the columns to the next highest power of two. This is required by a standard WHT implementation. The extra pixels generated as a result of this padding are set to zero.

The algorithm then calculates the Hadamard matrix for the image. A Hadamard matrix is used for scaling purposes. Note that the Hadamard matrix computed here isn't necessarily the one you'll find in common literature, but rather matches MATLAB's implementation. After that, the image array and the Hadamard matrix are multipled so that the image is scaled. Finally, the image array is normalized to the standard grayscale range and then ImageMagick's *display* routine is used to display the image.

whtrmse

Takes in: two images and a number of blocks
Produces: an image shown using ImageMagick and text output

whtrmse runs *wht* on the first image argument, so the image values are normalized and then ImageMagick's *display* routine is used to display the image. The second argument is MATLAB's output of *fwht* run on the same image.

Before we do any error analysis, we need to convert our double array for image values to a nearest integer array. This is because PGM stores the array values as unsigned chars and when those unsigned chars are converted back to doubles, the fractional part is lost.

An error image that represents this nearest integer array and the second image is calculated. From there, the mean and root-mean-square error (RMSE) are calculated for the error image. This is used to verify that the wht routine is running properly. If it's running properly, the RMSE will be at most 1/2^(bits per pixel), which represents the natural error inherent in an image.

We actually run this error analysis on the whole image and segments of the image. Segments are calculated based off of the number of blocks that the user provides in a row-wise fashion. We do this in order to better identify where the error within an image may lie.

gcon, amax, amin, mmax, and mmin

Takes in: one image, a 3x3 or 5x5 list of integers, and the size of the file to be produced.

Produces: one output image

These routines run geometric and linear convolutions on an image array.

rmse

Takes in: two images

Produces: text output

rmse computes the RMSE for all four quadrants of the difference of the two images as well as the whole difference image.

rmsegrid

Takes in: two images and a number of rows and columns

Produces: tabular output

rmsegrid runs *rmse* and produces its output as a table on the screen, breaking the output up into multiple tables if there are more than 10 columns.

rmsegrid01

Takes in: two images and a number of rows and columns

Produces: tabular output

rmsegrid01 normalizes the RMSE array based on the min/max from both of the input images. It then displays the result in a table format similar to *rmsegrid*.

rmseann

Takes in: two images and a number of rings

Produces: text output

rmseann produces an annular RMSE table based on a number of rings.

fft

Takes in: an image and the root of the output file

Produces: an output file

fft runs the fast Fourier transformation (FFT) on an image and shows its results using ImageMagick’s *display* routine after normalizing the image to a standard grayscale range.

fftspect

Takes in: an image and an output file
Produces: an output file

fftspect runs graph programs on the result of fft to look at its spectrometry.

help

Takes in: nothing
Produces: nothing

help displays the format of every command currently available in UIA.

Cleaning Up

After running the command, UIA removes all temp files and frees unnecessary image arrays. The code picks up after itself for subsequent uses.

For More Information

The code is well-commented in most areas. For more technical information, look at the comments corresponding to the command you are trying to run. And when that isn’t enough, try researching the transformation and how it is typically implemented in code. What we do here is nothing out of the ordinary, so background research should have you well-acquainted to work with the code.

If all else fails, email Dr. Mark Schmalz at mssz@cise.ufl.edu for more clarification.