

LuaIP 0.6 Reference Manual

CSC 4425/542 Digital Image Processing
John Weiss and Alex Iverson, Spring 2017

1. Introduction

LuaIP is a package for creating interactive menu-driven image processing programs in Lua.

LuaIP programs feature a main window with dropdown menus, and images displayed in tabbed window panes. Selecting File|Open brings up a file dialog box in which you may browse the file system to select images. Left-clicking and dragging a tab allows you to reposition the image in a split pane. Right clicking on a tab allows you to duplicate or reload the image. The Ctrl-+/- keys may be used to zoom in/out of an image.

LuaIP programs rely on the presence of ip.lua (the packed distributable) in the current directory (or on the Lua package path).^{*} LuaIP applications must start with the following require statements:

```
require "ip"
local viz = require "visual"
local il = require "il"
```

To execute a LuaIP program, call the start() function.

^{*} Also wx.dll (Windows) or wx.so (Linux) on the Lua package cpath. This is not an issue in ZeroBrane Studio, but may be if you wish to run LuaIP apps from the command line.

2. Menus

LuaIP applications are generally interactive, menu-driven programs. The default menu is a File menu, with Open, Save, and Exit menu items. To add new menus, use the following approach:

```
viz.imageMenu( "My Menu",
{
    {"Grayscale", il.grayscaleYIQ},
    {"Negate\tCtrl-N", il.negate, hotkey = "C-N"},
    {"Binary Threshold", il.threshold,
        {{name = "threshold", type = "number", displaytype = "slider",
            default = 128, min = 0, max = 255}}},
    }
)
```

This adds a new menu (My Menu) with three menu items (Grayscale, Negate, and Binary Threshold). The Grayscale entry invokes the LuaIP function il.grayscaleYIQ, Negate invokes il.negate, and Binary Threshold invokes il.threshold. Negate adds a hotkey (Ctrl-N). Binary Threshold pops up a dialog box for user input, displaying a slider bar. Input types include number, boolean, and string. Display types include sliders and spinners (“spin”) for integer input, and text boxes (“textbox”) for string and float input.

When a menu item is selected, the associated function is called. The current image (typically the image with the focus) is passed in as the first argument. Additional arguments, if any, are added via dialog boxes. The displayed image is automatically updated with the returned image from the

function call. You may attach your own functions to menu items; just be sure they expect an image as the first argument, and return an image. If multiple images are returned, these will be displayed in separate tabs. For example, “return img1,img2” will display img1 in the current tab, and open a new tab that displays img2.

The full specification for an imageMenu definition is as follows.

The first argument of the function is the menu name, as it will appear in the menubar. The second argument is an array-like table of menu items. The first table entry is the menu item name, as it will appear in the dropdown menu. The second entry is the function to call. The third entry is an (optional) additional argument specifier. The menu item may also define a hotkey as a string containing the control character (C/M/S for Ctrl/Alt/Shift), a hyphen, and the hotkey character.

The additional argument specifier is an array-like table containing argument descriptors. Each argument descriptor must have a name and a type, and may have a displaytype, default, min, max, and help. The name is used to label the UI fields and is not required to match any variable in the program. Currently supported types are number, boolean, string, point, and colour. Number displaytypes are spin, slider, and textbox (floating point input is only supported with textbox). Min and max are only supported for numbers.

3. Images

The image type is implemented as a C-style struct for performance. Because these are C structs, row/column indexing starts at 0 (not 1). The image library provides functions for creating images, and methods for manipulating them.

a. Creation

An image can be created in several ways, including:

`image.flat(width,height,fill)` – create an RGB image with dimensions width X height. Flat supports multiple types of (optional) fill specifiers, including a single intensity value (to create a monochrome RGB image), or three separate RGB values (to create an RGB image of that color).

`image.open(fname)` - read an image from the image file fname. Most common image file formats (JPG,PNG,GIF,BMP,etc.) are supported.

b. Manipulation

Image objects have methods that allow accessing individual pixels, iteration over the image, and mapping input pixel values to output pixel values.

`img:at(r,c)` - get the pixel at the given row and column, and return a pixel struct (described in the next section)

`img:pixels(border)` - iterate over all the row, column indices of an image, except for an optional border (default 0). The following loops are equivalent:

```
for r,c in img:pixels(w) do . . . end
for r=w,img.height-w-1 do for c = w,img.width-w-1 do . . . end end
```

`img:mapPixels(func)` - iterate over all the pixels of an image calling the provided function with the channels to transform the image. `mapPixels` transforms the image in place, and returns the image transformed image.

For example, to negate an image using `mapPixels`:

```
local function negate( img )
    return img:mapPixels(function( r, g, b )
        return 255 - r, 255 - g, 255 - b
    end
)
end
```

`img:clone()` – return a copy of an image

`img:write(fname)` - write an image to the file `fname`

c. Structs and Fields

The image type has three fields: width, height, and data. Data is a flat C-style array containing all the pixel data. Array access is unchecked, and can cause a segfault. It is recommended to use the methods provided (`img:at(r,c)`) rather than accessing the data field directly.

The pixel type has fields `r`, `g`, `b`, `y`, `in`, `q`, `u`, `v`, `i`, `h`, `s`, `rgb`, `yuv`, `yiq`, and `ihs`. The single-channel fields are unsigned 8-bit chars (bytes), and the three-channel fields are 3-element byte arrays. Because these are C structs, the arrays are indexed from 0 (not 1). For space efficiency, these fields overlap, and a pixel can only have valid data in one color space at a time. In other words, the `rgb`, `yuv`, `yiq`, and `ihs` arrays all overlap, and also overlap with the individual channels (`r,g,b,y,in,q,u,v,i,h,s`).

For example, to negate RGB image intensities, you can access the RGB fields in several ways:

```
for r,c in img:pixels() do
    img:at(r,c).rgb[0] = 255 - img:at(r,c).rgb[0] -- red
    img:at(r,c).yiq[1] = 255 - img:at(r,c).yiq[1] -- green
    img:at(r,c).b = 255 - img:at(r,c).b -- blue
end
```

4. LuaIP Function Reference

To call a LuaIP function, prepend “`il.`” to the function name. The first function argument is the input image (which may or may not be modified). Additional arguments may be used to supply other function inputs. LuaIP functions return processed images. For example,

```
img2 = il.threshold( img1, 100 )
```

performs binary thresholding on `img1`, using an intensity threshold of 100, and returns a binary thresholded image that is referenced by `img2`.

The following functions, grouped by category, are available in LuaIP.

Color Models

These routines are useful for color image processing. In general, you will convert a color image from RGB to YIQ (or YUV or IHS), process the intensity component, and convert the result back to RGB.

RGB2YIQ(img), YIQ2RGB (img) – convert between RGB and YIQ

RGB2YUV(img), YUV2RGB (img) – convert between RGB and YUV

RGB2IHS(img), IHS2RGB (img) – convert between RGB and IHS (aka HSI)

GetR(img), GetG(img), GetB(img) - return a grayscale image consisting of the R (red), G (green), or B (blue) component of RGB

GetY(img), GetInphase(img), GetQuadrature(img) - return a grayscale image consisting of the brightness (Y), inphase (I), or quadrature (Q) component of YIQ

GetY(img), GetU(img), GetV(img) - return a grayscale image consisting of the brightness (Y), U, or V component of YUV

GetI(img), GetH(img), GetS(img) - return a grayscale image consisting of the intensity(I), hue (H), or saturation (S) component of HIS

GetIntensity(img,model) – returns intensity component based on model, which may be ‘yiq’, ‘yuv’, or ‘ihs’ (default: ‘yiq’)

RGB2XYZ(img) - return a color image with the RGB components swapped

RGB2XYZ may be: RGB2BGR, RGB2BRG, RGB2GBR, RGB2GRB, RGB2RBG

falseColor(img,img1,img2,img3) – create an RGB image from three monochrome images (the input image is ignored)

Point Processes

Point processes map input intensities to output intensities. The output depends only on the pixel value at a single point. Point processes include changes to image brightness, contrast, and color. In some of these operations, you may specify the color model (rbg, yiq, yuv, ihs).

brighten(img,gamma,model) – change image brightness

contrastStretch(img,min,max) – rescale image intensities between [min,max] to [0,255]
(intensities below min get set to 0, above max get set to 255)

gamma(img,gamma,model) – change image gamma

grayscale(img,model) – convert image to grayscale

grayscaleIHS(img) – convert to grayscale using IHS

grayscaleYIQ(img) – convert to grayscale using YIQ

logscale(img, model) – perform log scaling on image

negate(img, model) – negate image

posterize(img,n,model) – posterize image by requantizing intensities to n levels
posterizeIHS(img,n) – posterize to n levels using IHS intensity
posterizeYIQ(img,n) – posterize to n levels using YIQ intensity
posterizeRGB(img,n) – posterize to n levels using individual RGB channels

pseudocolor1(img) – 8-level pseudocolor
pseudocolor2(img) – continuous pseudocolor
pseudocolor3(img) – “walk around color cube” pseudocolor
pseudocolor4(img) – random pseudocolor

sawtooth(img,n) – n-level grayscale sawtooth scaling
sawtoothBGR(img) – 8-level color sawtooth scaling
sawtoothRGB(img) – 8-level color sawtooth scaling

scaleIntensities(img,min,max) – rescale image intensities from [0,255] to [min,max], reducing the dynamic range and contrast of the image

slice(img,plane) – bit-plane slicing (plane is 0 to 7)

solarize(img) – image solarization (inverts dark intensities)

Histograms

An image histogram is a frequency distribution of pixel intensities. Histogram manipulation alters the statistical distribution of pixel intensities in the image, and supports automatic linear contrast stretch and equalization. In some of these operations, you may specify the color model (rbg, yiq, yuv, ihs).

stretch(img,model) – histogram-based contrast stretch; applies a linear ramp that maps from [imin,imax] to [0,255] (imin, imax are min,max image intensities)
stretchSpecify(img,dark,light,model) – histogram-based contrast stretch, ignoring specified percentages of dark and light pixels

equalize(img,model) – histogram equalization (based on image intensities)
equalizeClip(img,percent,model) – histogram equalization (intensities), with clipping of histogram values that exceed the specified percentage of image pixels
equalize(img,model) – histogram equalization
equalizeRGB(img) – histogram equalization of individual RGB channels
equalizeYIQ(img) – histogram equalization of intensity channel (YIQ version)
equalizeYUV(img) – histogram equalization of intensity channel (YUV version)
equalizeIHS(img) – histogram equalization of intensity channel (IHS version)

adaptiveEqualize(img,w) – adaptive histogram equalization using wxw neighborhood
adaptiveContrastStretch(img,w) – adaptive histogram-based contrast stretch using wxw neighborhood

histogram(img,model) – returns intensity (or RGB) histogram (table, NOT an image)
histogramRGB(img) – returns RGB histogram (table, NOT an image)

showHistogram(img,model) – displays intensity (or RGB) histogram of img in new tab
renderMonoHistogram(hist) – displays given histogram in image tab
showHistogramRGB(img) – displays color (RGB) histogram of img in new tab
renderHistogramRGB(hist) – displays color histogram in image tab

Convolution Filtering

Convolution-based filtering produces an output intensity from a weighted sum of pixel intensities in a neighborhood. Smoothing, sharpening, and edge detection are common convolution-based filtering operations.

sharpen(img) – 3x3 sharpening
smooth(img) – 3x3 center-weighted smoothing

mean(img,w) - wxw neighborhood mean (unweighted)
meanW1(img,w) - wxw neighborhood mean (center weighted)
meanW2(img,w) - wxw neighborhood mean (more center weighted)
meanW3(img,w) - wxw neighborhood mean (Gaussian weighted)

emboss(img) – image embossing

Rank Order Filtering

Rank order filtering produces an output intensity from a sorted list of neighborhood pixel intensities. Median, minimum, and maximum are common rank order filtering operations.

median(img,w) - wxw neighborhood median
medianPlus(img) – 3x3 plus-shaped median filter

maximum(img,w) - wxw neighborhood maximum
minimum(img,w) - wxw neighborhood minimum
range(img,w) - wxw neighborhood range

Statistical Filtering

Statistical filtering includes neighborhood operations such as variance and standard deviation.

stdDev(img,w) - wxw neighborhood standard deviation
variance(img,w) – wxw neighborhood variance

statDiff(img,w,k) – wxw statistical differencing (k is scale factor)

Edge Detection

Edge detection may be accomplished by first derivative operators (Sobel), second derivative operators (Marr-Hildreth), template matching (Kirsch), etc. Gradient-based edge operators (directional 2-D first derivative) are vectors with magnitude and direction.

sobelMag(img) – Sobel edge magnitude
sobel(img) – Sobel edge magnitude and direction (returns orig, mag, dir images)

kirsch(img) – Kirsch edge magnitude and direction (returns orig, mag, dir images)

laplacian(img) – Laplacian edge magnitude (nondirectional)
marrHildreth(sigma) – Marr-Hildreth edge operator (Laplacian of Gaussian, with zero crossings),
with sigma for Gaussian smoothing

Morphological Operations

Mathematical morphology is based on set theory. Morphological (“shape”) filters are based upon the operations of erosion and dilation (also hit-miss), and can be used as alternatives to many convolution filters.

dilate(img,w) – wxw image dilation
erode(img,w) - wxw image erosion

close(img,w) - wxw image closing (dilation followed by erosion)
open(img,w) - wxw image opening (erosion followed by dilation)

smoothCO(img,w) - wxw image smoothing (closing followed by opening)
smoothOC(img,w) - wxw image smoothing (opening followed by closing)

morphGradient(img) – morphological gradient (edge detection)

sharpenMorph(img) - morphological sharpening

Frequency Domain

Convolution may also be performed in the frequency domain. Frequency filters include low pass filters (for smoothing), high pass filters (for edge detection), and high frequency enhancement (for sharpening).

dftMagnitude(img) – discrete Fourier transform magnitude (centered and log scaled)
dftPhase(img) – discrete Fourier transform phase
dft(img) – display DFT magnitude and phase in separate tabs

frequencyFilter(img,type,cutoff,boost,low,high) – frequency domain filtering; type is ideal, gaussLPF, gaussHPF; low/high are scale factors for frequencies below/above cutoff percentage; boost is for high-frequency emphasis

fft1D(dir,real,imag) – 1-D FFT of real and imag 1-D arrays (zero index); “dir” is negative for inverse transform
fft2D(dir,real,imag) – 2-D FFT of real and imag 2-D arrays (zero index); “dir” is negative for inverse transform

Image Arithmetic

Image arithmetic allows you to add, subtract, multiply and divide two images (or one image and a constant).

add(img1,img2) – sum of two images (img1 + img2)
sub(img1,img2) – difference between two images (img1 – img2)

Image Geometry

Affine transforms include translation/scaling/rotation, along with more general image warping. Pixel color interpolation may be nearest neighbor (nn) or bilinear (bi).

`rescale(img,nrows,ncols,interp)` – resize image to `nrows x ncols`

`rotate(img,angle,interp)` – rotate image by `angle` degrees

Segmentation

Image segmentation partitions an image into meaningful regions.

`contours(img,interval)` – isointensity contours at given intensity interval

`addContours(img,interval)` – add isointensity contours to image

`connComp(img,epsilon)` – image segmentation via connected components; epsilon is “fudge factor” for deciding whether pixel belongs to a component

`sizeFilter(img,epsilon,thresh)` – connected components followed by filtering out any component below `thresh` pixels in size

`chamfer34(img)` – chamfer 3-4 distance transform (distance to nearest black pixel)

`threshold(img,thresh)` – binary threshold at `thresh`

`iterativeBinaryThreshold(img)` – automated iterative binary threshold (prints threshold)

`adaptiveThreshold(img,w)` – adaptive iterative binary threshold using `wxw` neighborhood

Misc

Catch-all category, including additive noise and utility routines.

`impulse(img,p)` – add impulse noise to image with probability $1/p$ (1 pixel in every `p` pixels gets an impulse); impulse is black (0) for light pixels, white (255) for dark pixels

`gaussianNoise(img,sd)` – add random noise sampled from Gaussian (normal) distribution with mean 0 and given `sd` (default 1) to each pixel intensity in the `img`

`curry(func,...)` – utility routine to generate a function; e.g., `il.curry(il.median,3)` generates a median filter of size 3

Message Boxes

`viz.imageMessage(title,msg)` – display a message box with specified title and message (strings)
(Note: `viz.imageMessage`, not `il.imageMessage`)