Week 13 - Lecture 25

István Albert

Huck Institutes for the Life Sciences
Image analysis

All lines are actually straight
Extremely simple text formats

Images specified as text files

• PBM – portable bit map (monochrome)

• PPM – portable pixel map (color)

**No operating** system can visualize them by default you’ll need to install software that can do it PaintShop, PhotoShop, Gimp, IvanView etc
PBM – portable bit map

Search for the formats and read about the various fields: magic number, info, width and height, bit values
PPM - portable pixel map

Search for the formats and read about the various fields:
magic number, info, width and height, number of colors, color triplets
The images contained in the previous slides

F E E P

The PBM example

The PPM example
When do we use simple formats

• Very simple visualizations of small rectangular images

• Usually magnified so that each pixel corresponds to a square and represents some information

• For anything more complicated use the Python Imaging Library
Sending Greg to exotic locations

Images from: Software Carpentry
http://software-carpentry.org/
by Greg Wilson
The python imaging library

```python
# will attempt to autodetect the format
im = Image.open('greg.png')

print 'Size', im.size
print 'Info', im.info
print 'Format', im.format
print 'Mode', im.mode

im.show()
```

Command Output

```
Size (640, 480)
Info {'dpi': (72, 72)}
Format PNG
Mode RGB
```
The **mode restricts** what operations may be performed on an image.

```python
import Image

# RGB - Red, Green, Blue triplets for every pixel
# L - Luminescence (intensity) values for every pixel

im = Image.open('greg.png')
print 'Mode', im.mode
print 'All known modes', Image.MODES
```

```
Mode RGB
All known modes ['1', 'CMYK', 'F', 'I', 'L', 'P', 'RGB', 'RGBA', 'RGBX', 'YCbCr']
```
Lossy and lossless formats

- Each format is optimized for some purpose, usually a tradeoff between file size and color accuracy

- Lossy compression is meant to be looked at by humans – allows for small imperfections – produces a very small file

- With lossy compression each time the file is saved the image gets worse! (JPG ➔ most of your photos are stored in JPG)
Common formats

• **TIFF – Tagged Image File Format** – very large number of modes – highest color depth – less efficient compression – too many variants

• **PNG – Portable Network Graphics** - multiple modes - efficient filesize – limited color depth

• **GIF – Graphics Interchange Format** - only 256 colors - very small files, very limited color depth

• **JPG – Joint Photographic Experts Group** – lossy compression, limited color depth for single channel
```python
import Image

im = Image.open('greg.png')
print 'Format=%s, Mode=%s' % (im.format, im.mode)

# convert to various formats
# new image type set by extension jpg, gif, tiff, bmp
for ext in ('gif', 'jpg', 'tiff', 'bmp'):
    newname = 'greg.%s' % ext
    im.save(newname)

# reopen the image
nim = Image.open(newname)
print 'Saved in format=%s, mode=%s' % (nim.format, nim.mode)
```

Output:
```
Format=PNG, Mode=RGB
Saved in format=GIF, mode=Pa8
Saved in format=JPEG, mode=RGB
Saved in format=TIFF, mode=RGB
Saved in format=BMP, mode=RGB
```
Crop parts of an image

```python
import Image

im = Image.open('greg.png')

# coordinates top x, y, bottom x, y
box = (180, 180, 300, 350)
cropped_im = im.crop(box)
cropped_im.show()
```
import Image

im = Image.open('greg.png')

# resize method -> ANTIALIAS, BICUBIC, BILINEAR, NEAREST
small = im.resize((100, 100), Image.ANTIALIAS)
small.show()

# expand -> include the whole image or not
# 180 -> upside down
rot = im.rotate(180, expand=False)
rot.show()
rot.save('upside.png')
Creating a new image

```python
import Image

im = Image.open('greg.png')

# create new image
collage = Image.new('RGB', (640, 480))
small = im.resize((100, 100))

for i in range(10):
    location = i*60, i*40  # new location
    print 'Pasting to', location
    collage.paste(small, location)

collage.show()
```

Pasting to (0, 0)
Pasting to (60, 40)
Pasting to (120, 80)
Pasting to (180, 120)
Pasting to (240, 160)
```python
import Image

im = Image.open('greg.png')

# create new image
collage = Image.new('RGB', (640, 480))
small = im.resize((100, 100))

for x in range(7):
    for y in range(5):
        location = x*100, y*100  # new location
        print 'Pasting to', location
        collage.paste(small, location)

collage.show()
```

```
Pasting to (0, 0)
Pasting to (0, 100)
Pasting to (0, 200)
Pasting to (0, 300)
```
The outputs of the last two programs
Image modes

**Mode**

The mode of an image defines the type and depth of a pixel in the image. The current release supports the following standard modes:

- **1** (1-bit pixels, black and white, stored with one pixel per byte)
- **L** (8-bit pixels, black and white)
- **P** (8-bit pixels, mapped to any other mode using a colour palette)
- **RGB** (3x8-bit pixels, true colour)
- **RGBA** (4x8-bit pixels, true colour with transparency mask)
- **CMYK** (4x8-bit pixels, colour separation)
- **YCbCr** (3x8-bit pixels, colour video format)
- **I** (32-bit signed integer pixels)
- **F** (32-bit floating point pixels)
RGB colors

- Additive color model → **Red**, **Green**, and **Blue** light are added together to reproduce a broad array of colors.

- Often represented as triplets ranging from (0.0, 1.0) or (0, 255)

  
  
  
  (0, 0, 0) → black

  (255, 0, 0) → red

  (0, 0, 255) → blue

  (255, 255, 0) → yellow

  (255, 255, 255) → white

  (100, 100, 100), (200, 200, 200) → grey of different hues
Let’s get the data for the image

```python
import Image

im = Image.open('greg.png')

# get the data for the image
data = im.getdata()

# turn it into a list (usually numpy array)
data = list(data)

# a list of tuples of (red green blue) colors
print data[:10]
```

```
[(15, 21, 243), (15, 21, 243), (15, 21, 243), (15, 21, 243), (15, 21, 243), (15, 21, 243), (15, 21, 243), (15, 21, 243), (15, 21, 243), (15, 21, 243)]
```
Changing the blue channel

```python
import Image

im = Image.open('greg.png')
data = im.getdata()
data = list(data)

def transform(colors):
    r, g, b = colors
    if b >= 243:
        return r, g, 0
    else:
        return r, g, b

data = map(transform, data)

# put the colors back into the image
im.putdata(data)
im.show()
im.save('greg-changed.png')
```
Change the color when the blue dominates

```python
import Image

im = Image.open('greg.png')
data = im.getdata()
data = list(data)

def transform(colors):
    r, g, b = colors
    if b > r and b > g:
        return r, g, 0
    else:
        return r, g, b

data = map(transform, data)

# put the colors back into the image
im.putdata(data)
im.show()
im.save('greg-changed.png')
```
Swap the color with another image

```python
import Image

im = Image.open('greg.png')
bg = Image.open('elmore-sunset.png')

fore = list( im.getdata() )
back = list( bg.getdata() )

def transform(fg, bg):
    r, g, b = fg
    if b > r and b > g:
        return bg
    else:
        return fg

data = map(transform, fore, back)

# put the colors back into the image
im.putdata(data)
im.show()
im.save('greg-changed.png')
```
Oops.

Homework: do a little better than this.
Tip: investigate the colors around his shirt
Channel Operations

Original Sensor Array

Red

Green

Blue

Incomplete Red, Green, and Blue Channels

Raw Converter

Final RGB File
Project reminder

Project structure:

• Obtain the data of interest
• Describe the properties of your data
• Write program(s) that operate on your data
• Generate a report and/or plots for your results

Deadline: Thursday - December 10
Image modes

**Mode**

The mode of an image defines the type and depth of a pixel in the image. The current release supports the following standard modes:

- **1** (1-bit pixels, black and white, stored with one pixel per byte)
- **L** (8-bit pixels, black and white)
- **P** (8-bit pixels, mapped to any other mode using a colour palette)
- **RGB** (3x8-bit pixels, true colour)
- **RGBA** (4x8-bit pixels, true colour with transparency mask)
- **CMYK** (4x8-bit pixels, colour separation)
- **YCbCr** (3x8-bit pixels, colour video format)
- **I** (32-bit signed integer pixels)
- **F** (32-bit floating point pixels)
Sample images – green/red dyes

green

red
Check image mode and data

```python
import Image

im = Image.open('samples/red.png')
data = list(im.getdata())

print im.mode
print data[:25]

# im.show()
```
Correction: noticed only after printing the slides

- ‘P’ mode is palette – it is not a color – is the index of the color in the palette!

- We need to convert the image to a luminosity mode, in each example add:

```
im = im.convert('L')
```

It still works with P, and thus went unnoticed because the palette contains the luminosities in the correct increasing order.
Histogram of values

```python
import Image
import pylab

im = Image.open('samples/red.png')
data = list(im.getdata())

pylab.hist(data, bins=50)
pylab.show()
```
HW: Improve the histogram that is dominated by low values (black)

Homework 1: filter the low color values and plot the new histogram. How many data points do you have before and after filtering?
Inverting the image

```
import Image, ImageChops

red = Image.open('samples/green.png')
inv = ImageChops.invert(red)
inv.save('inverted.png')

inv.show()
```

Channel operations: ImageChops
The inverted images that we wish to compare

green.png  red.png
HW: Improve the inverted histogram

Homework 2: do the same task as in Homework 1 but now on the inverted image
Absolute image differences

```python
import Image, ImageChops

red = Image.open('samples/red.png')
grn = Image.open('samples/green.png')

# absolute difference pixel by pixel
diff = ImageChops.difference(red, grn)

# invert the image to save on ink
inv = ImageChops.invert(diff)
inv.save('diff.png')
inv.show()

newpixel = abs(pixel1 – pixel2)
```
Absolute difference
Subtract – relative differences

```python
import Image, ImageChops

red = Image.open('samples/red.png')
grn = Image.open('samples/green.png')

# relative differences pixel by pixel
res = ImageChops.subtract(red, grn)

# invert the image to save on ink
res = ImageChops.invert(res)
res.show()
res.save('diff.png')
```

\[
\text{out} = \frac{\text{image1} - \text{image2}}{\text{scale}} + \text{offset}
\]
Subtracting images

red - green

green - red
Homework 3: do the same task as in Homework 1 but now on the subtracted image
import Image, ImageChops, ImageFilter

grn = Image.open('samples/green.png')
inv = ImageChops.invert(grn)

size = 15

# For each pixel in the input image,
# this filter copies the smallest pixel
# value from a (size, size) environment
# to the output image.
out = inv.filter( ImageFilter.MinFilter(size) )

#out = inv.filter( ImageFilter.MaxFilter(size) )
out.save('filtered.png')
out.show()
Minfilter - Maxfilter

MinFilter strengthens low values, MaxFilter strengthens high values
```python
import Image, ImageStat

greg = Image.open('greg.png')

# generate statistics
stat = ImageStat.Stat(greg)

# reported by channels
print 'Count', stat.count
print 'Sum', stat.sum
print 'Mean', stat.mean
print 'Dev', stat.stddev
```
Part II. Module Reference

The Image Module
The ImageChops Module
The ImageColor Module
The ImageDraw Module
The ImageEnhance Module
The ImageFile Module
The ImageFileIO Module
The ImageFilter Module
The ImageFont Module
The ImageGrab Module
The ImageMath Module (PIL 1.1.6 only)
The ImageOps Module
The ImagePalette Module
The ImagePath Module
The ImageQt Module (PIL 1.1.6 only)
The ImageSequence Module
The ImageStat Module
The ImageTk Module
The ImageWin Module
The PSDraw Module

The ImageCrackCode Module (PIL Plus only)
The ImageGL Module (PIL Plus only)
Other filters

**RankFilter**

`RankFilter(size, rank)`

(New in 1.1.5) Create a rank filter of the given size. For each pixel in the input image, the rank filter sorts all pixels in a (size, size) environment by pixel value, and copies the rank'th value to the output image.

**MinFilter**

`MinFilter(size=3)`

(New in 1.1.5) Create a min filter of the given size. For each pixel in the input image, this filter copies the smallest pixel value from a (size, size) environment to the output image.

**MedianFilter**

`MedianFilter(size=3)`

(New in 1.1.5) Create a median filter of the given size. For each pixel in the input image, this filter copies the median pixel value from a (size, size) environment to the output image.

**MaxFilter**

`MaxFilter(size=3)`

(New in 1.1.5) Create a max filter of the given size. For each pixel in the input image, this filter copies the largest pixel value from a (size, size) environment to the output image.

**ModeFilter**

`ModeFilter(size=3)`

(New in 1.1.5) Create a mode filter of the given size. For each pixel in the input image, this filter copies the most common pixel value in a (size, size) environment to the output image. If no pixel value occurs more than once, the original pixel value is used.
**difference**

ImageChops.difference(image1, image2) => image

Returns the absolute value of the difference between the two images.

\[ cut = \text{abs}(image1 - image2) \]

**multiply**

ImageChops.multiply(image1, image2) => image

Superimposes two images on top of each other. If you multiply an image with a solid black image, the result is black. If you multiply with a solid white image, the image is unaffected.

\[ cut = image1 \times image2 / \text{MAX} \]

**screen**

ImageChops.screen(image1, image2) => image

Superimposes two inverted images on top of each other.

\[ cut = \text{MAX} - ((\text{MAX} - image1) \times (\text{MAX} - image2) / \text{MAX}) \]

**add**

ImageChops.add(image1, image2, scale, offset) => image

Adds two images, dividing the result by scale and adding the offset. If omitted, scale defaults to 1.0, and offset to 0.0.

\[ cut = (image1 + image2) / \text{scale} + \text{offset} \]

**subtract**

ImageChops.subtract(image1, image2, scale, offset) => image

Subtracts two images, dividing the result by scale and adding the offset. If omitted, scale defaults to 1.0, and offset to 0.0.

\[ cut = (image1 - image2) / \text{scale} + \text{offset} \]
**constant**

`ImageChops.constant(image, value) => image`

Return a layer with the same size as the given image, but filled with the given pixel value.

**duplicate**

`ImageChops.duplicate(image) => image`

Return a copy of the given image.

**invert**

`ImageChops.invert(image) => image`

Inverts an image.

\[
\text{out} = \text{MAX} - \text{image}
\]

**lighter**

`ImageChops.lighter(image1, image2) => image`

Compares the two images, pixel by pixel, and returns a new image containing the lighter values.

\[
\text{out} = \text{max(image1, image2)}
\]

**darker**

`ImageChops.darker(image1, image2) => image`

Compares the two images, pixel by pixel, and returns a new image containing the darker values.

\[
\text{out} = \text{min(image1, image2)}
\]
HW4: optional homework for extra credit

generate an image by applying a number of channel operations