Point Processing

Point processing is a simple form of image processing where operations involve only corresponding pixels (those with the same x,y).

Point processing is a bit limited, but it's simple and fast.

Single Image Point Processing: $b(x,y) = f(a(x,y))$

output image **b** is a function of input image **a**

we call *f* the processing function

Multi-Image Point Processing: $b(x,y) = f(a_1(x,y),a_2(x,y),\ldots,a_n(x,y))$

output image **b** is a function of *n* input images $\mathbf{a}_1$, $\mathbf{a}_2$, ..., $\mathbf{a}_n$

How to Do Single Image Point Processing

$$b(x,y) = f(a(x,y))$$

Can often be done simply by **changing the colormap**!

The intensity that we see is $I(x,y) = m(v(x,y))$, where v is pixel value, m is the colormap function, and I is intensity, so:

- suppose input image is already in memory (i.e. **v=a**)
- put processing function f into colormap ($m=f$)

Colormap changes are fast (on an 8 bit frame buffer, only 256 memory writes to do), nondestructive (we don't change pixel values), but they apply to the whole screen, even other windows, and you might not want that.

The other way to do single image point processing is by modifying the pixel values (also known as **indirecting through a lookup table**):

- loop over all pixels, assign $v(x,y) \leftarrow f(v(x,y))$
- leave colormap as-is

This method is slower, it's destructive, but it's more "polite" to the window system.

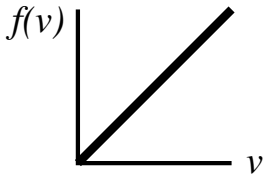
Uses of Single Image Point Processing

Suppose our picture is grayscale (a.k.a. monochrome).

Let v denote pixel value, suppose it's in the range [0,1].

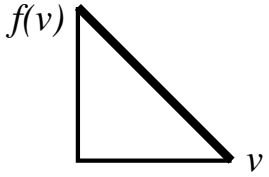
In practice, you typically work with pixel values in the range 0-255.

$f(v) = v$ **identity**; no change

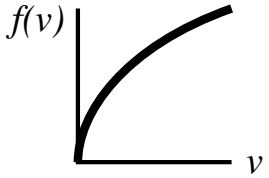


$f(v) = 1-v$ **negate** an image

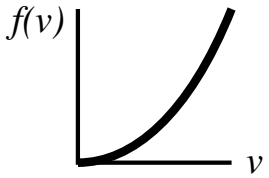
(black→white, white→black)



$f(v) = v^p, p<1$ **brighten**

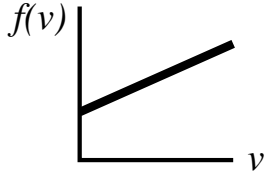


$f(v) = v^p, p>1$ **darken**

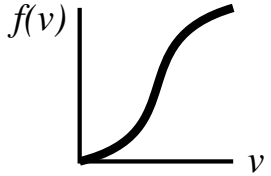


Contrast Enhancement

decreased contrast

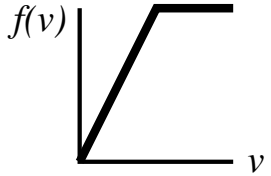


increased contrast



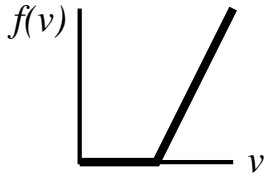
spread out low values

good on dark pictures



spread out high values

good on bright pictures



But it's tedious to manually find the colormap that increases the contrast best.

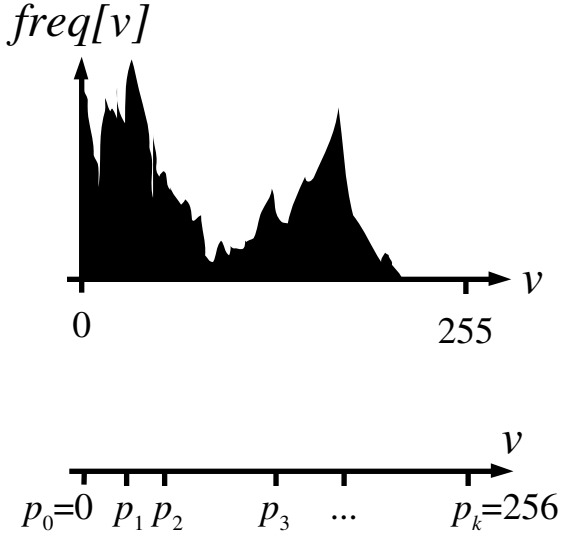
How to do it automatically?

Histogram Equalization 1

Look at the histogram of the image:

to find histogram of 8 bit image:

```
int freq[256];
for (v=0; v<256; v++)
    freq[v] = 0;
for (y=0; y<ny; y++)
    for (x=0; x<nx; x++)
        freq[pixel[y][x]]++;
```



Partition the histogram into k equal bins (typically, k=256):

place ith partition p_i so that: $$\sum_{v=0}^{p_i-1} freq[v] \approx i\frac{nx \cdot ny}{k}$$

do this quickly by pre-computing partial sums: $$sum[v] = \sum_{u=0}^{v-1} freq[u]$$

Histogram Equalization 2

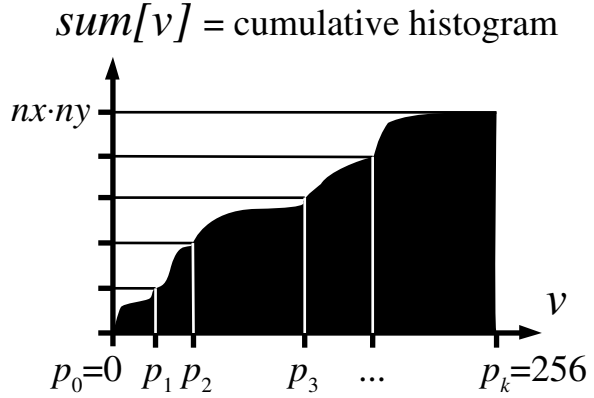
After computing *sum* array,
find partition points p_i such that

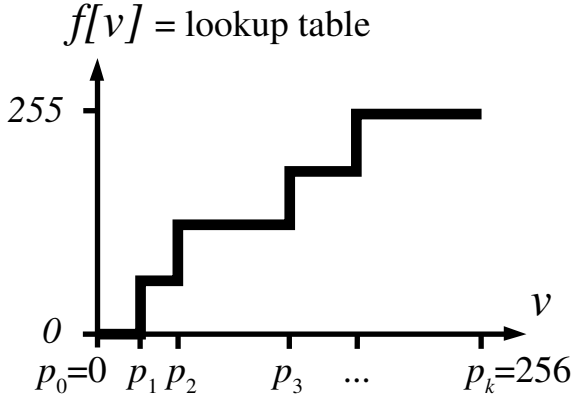
$$sum[p_i] \approx i\frac{nx \cdot ny}{k}$$

and make a lookup table where

$$f[v] = \frac{255 \cdot i}{k-1} \quad \text{if} \quad p_i \le v < p_{i+1}$$

and indirect pixel values through *f*
or load *f* into colormap.





After doing this, the histogram of the picture will be approximately flat (hence the name).
Histogram equalization is adaptive to the image -- a nice feature.

original, 128x128 pixels

original, 64x64 pixels

negated

histogram equalized

darkened

darkened,histogram equalized

Multi-Image Point Processing

$$b(x,y) = f(a_1(x,y), a_2(x,y), \ldots, a_n(x,y))$$

$$\text{or } \mathbf{b} = f(\mathbf{a}_1, \mathbf{a}_2, \ldots, \mathbf{a}_n)$$

Think of $\mathbf{a}_1, \mathbf{a}_2, \ldots, \mathbf{a}_n$ as matrices (as in linear algebra).
Do algebra with images just as with matrices.

Subtraction: $\mathbf{b} = \mathbf{a}_1 - \mathbf{a}_2$ i.e. $b(x,y) = \boldsymbol{a}_1(x,y) - \boldsymbol{a}_2(x,y)$

to compare images or $\mathbf{b} = |\mathbf{a}_1 - \mathbf{a}_2|$, or $\mathbf{b} = \mathbf{a}_1 - \mathbf{a}_2 + .5$

Averaging: $\mathbf{b} = \frac{1}{n} \sum_{i=1}^{n} \mathbf{a}_i$

to reduce noise

Compositing

Compositing: laying one image $\mathbf{a}_2$ over another, $\mathbf{a}_1$, according to a **matte image**, **m**.

Compositing with a hard (binary) matte, where $m(x,y)$=0 or 1:

if $m(x,y)$ then $b(x,y) = a_2(x,y)$ else $b(x,y) = a_1(x,y)$

Compositing with a soft matte, where m(x,y) can lie between 0 and 1:

$$b(x,y) = (1-m(x,y))\cdot a_1(x,y) + m(x,y)\cdot a_2(x,y)$$
$$= a_1(x,y) + m(x,y)\cdot(a_2(x,y)-a_1(x,y))$$

Simple case: $m(x,y) = m$ = constant

Called **blending** two images.

If you do it over time, and let m gradually increase from 0 to 1,

you get a **cross-dissolve**.