The *mirror* operation

Objective -

Produce an output image which is the mirror image of the input image. For each row, the pixel in the output image at a distance *d* from the left edge of the output image should be a copy of the pixel in the input image that is at a distance *d* from the *right* edge of the input image

Input -

```
mirror
{
   in0 dive.ppm
   out divemir.ppm
}
```

The *gray* operation

Objective -

Produce an output image in which the (r, g, b) components of the output pixel -

- all have the same value – making the image appear gray, and
- the value is the NTSC luminance of the input pixel: $0.30 * r + 0.59 * g + 0.11 * b$.

Example

Input pixel:	30 220 15
Output pixel:	(137, 137, 137)

Do *not* produce an actual P5 (one byte per pixel image here).

Input -

```
gray
{
   in0 dive.ppm
   out divegray.ppm
}
```

The *fade* operation

Objective -

Produce an output image in which the (r, g, b) components of the output pixel -

- are a weighted average of the NTSC luminance of the input pixel and the input pixel.
- the value of the output pixel should be:
- *factor * (luminance, luminance, luminance) + (1-factor) (input.r, input.g, input. b);*

Example -

Input pixel:	30 220 15
Factor	0.4
Output pixel:	(67, 187, 64)

Input -

```
fade
{
   factor 0.4
   in0 dive.ppm
   out divegray.ppm
}
```

The *bright* operation

Objective -

Produce an output image in which the (r, g, b) components of the output pixel -

- are multiplied by an input brightness *factor* and
- each component clamped to the value 255 if it should exceed 255.

Example -

Input pixel:	30 220 15
Factor	2.0
Output pixel:	(60, 255, 30)

Input -

```
bright
{
   factor 2.0
   in0 trump.ppm
   out trumpbrt.ppm
}
```

The *monochrome* operation

Objective -

Given a color (color.r, color.g, color.b), produce an output image in which the (r, g, b) components of the output pixel -

- have the same NTSC luminance of the input pixel and whose
- (r, g, b) components are given by *factor * (color.r, color.g, color.b)*

This will produce an output image in which all pixels are varying degrees of brightness of the specified color.

Example:

Input pixel:	30 220 15
Color	10 40 30
Output pixel:	(46, 184, 138)

Input -

```
monchrome
{
   color 10 40 30
   in0 trump.ppm
   out trumpmono.ppm
}
```

The *resize* operation

Objective -

Produce an output image which is a resizing of the input image.

For each *(row, col)* of the output image, the pixel should be a copy of the pixel in the input image located at

*(row * in.height / out.height, col * in .width / out.width),*

in the input image. The dimensions of the output image *are given in (width, height) format*.

Input -

```
resize
{
   in0 mike.ppm
   dims 1024 1024
   out mikesize.ppm
}
```

The *smooth* operation

Objective -

Produce an output image in which the each pixel in the output image is the result of applying a 3 x 3 *convolution filter* to nearby pixels in the input image. The filter used should be the following:

```
static double smoother[3][3] =
{
  {0.10,  0.12,  0.10},
  {0.12,  0.12,  0.12},
  {0.10,  0.12,  0.10},
}
```

It the output pixel is on any edge of the image, the output pixel should be a copy of the input pixel.

If not, the filter should be logically overlaid on the input image in such a way that the center of the filter lies at location *(row, col)* in the input image and the (r, g, b) components of the output image should be the weighted averages of its 9 nearest neighbor pixels.

Example:
Suppose the the following diagram shows the value of the *red components* of the center pixel at location (row, col) whose value is 130.

100	120	130
110	130	140
120	140	150

Then the value of the *red component* of the output pixel should be 126.8 = 127.

```
smooth
{
   in0 mikesize.ppm
   out mikesmth.ppm
}
```

The *sobel* operation

Objective -

You will implement a variant of the sobel edge detection filter that will produce an output image in which the brightness of each output pixel is a function of the *difference in luminance* in the 3 x 3 neighborhood of the pixel. The larger the change in luminance, the brighter the pixel should be. Two Sobel filters measure the gradient (rate of change) in the *x* and *y* directions.

```
static double sobel_x[3][3] =
{
    {-1.0, 0.0,  1.0},
    {-2.0, 0.0,  2.0},
    {-1.0, 0.0,  1.0}
};

static double sobel_y[3][3] =
{
  { 1.0, 2.0,  1.0},
  { 0.0, 0.0,  0.0},
  {-1.0,-2.0, -1.0}
};
```

Unlike the smoothing filter in which the filter was applied to each of the *(r, g, b)* channels of the input image, the sobel filters should be applied *only to the luminance* of the neighboring pixels.

Suppose the the following diagram shows the value of the *luminance components* of the center pixel at location (row, col) whose luminance is 130.

100	120	130
110	130	140
120	140	150

Then the value *s_y* computed by applying the *soble_y* filter at (row, col) should be -80.
The *sobel_x* filter value *s_x* should be computed in an analogous way.

The (r, g, b) components of the output pixel should be:

*factor * (in.r, in.g, in.b)*

where *factor* used to adjust the brightness of the pixel should be *sqrt(s_x * s_x + s_y * s_y) / 256;*

Multiplying by *factor* can produce a value that exceeds 255.0 If any pixel component exceeds 255 it *must* be clamped to 255.

It may be desirable increase the brightness of an image processed by the *sobel* operation by reprocessing it with the *bright* operation.

Input -

```
sobel
{
   in0 trump.ppm
   out trumpedge.ppm
}
```

The *caption* operation

Objective - The primary input image, whose name is given in *in0* ,is a picture on which a caption is to be superimposed. The input and output images will have the *same size.* The caption image may have a different size.

The caption image whose name is given in *in1* should be the image of some *black text* written on a *white background*. The output image should be the same size as the input image, and the caption image should be no larger than the input image. The *loc* parameter should specify the (*x, y)* offset where the upper left corner of the caption image should be placed on the image being captionend.

The caption should be composited with the input image as follows.

For each pixel in the output image

- If the pixel is not within the area overlaid by the caption, then the pixel from the corresponding location in the input image should be copied to the output image.
- If the pixel does lie in the caption area, it is necessary to determine if the corresponding pixel in the caption is a white background pixel or a black data pixel.
- If the caption pixel is a white background pixel, then the output pixel should be copied from the corresponding location in the input image.
- If the caption pixel is a black data pixel, then the output pixel should be set to the color specified in the color parameter.

Input -

```
caption
{
   in0 trump.ppm
   in1 t100c.ppm
   loc 300  100
   color 255 215 0
   out trumpcap.ppm
}
```

The *composite* operation

Objective - The primary input image, whose name is given in *in0* ,is a picture onto which a *blue-screen* image whose name is given by *in1* is to be superimposed. The blue-screen image is so-called because it is photograph taken with the subject standing in front of a blue screen. The output image should be the same size as the input image, and the blue-screen image should be no larger than the input image. The *loc* parameter should specify the (*x, y)* offset of the upper left corner of the blue-screen image within the output image.

The blue-screen image should be composited and blended with the input image as follows.

For each pixel in the output image

- If the pixel is not within the area overlaid by the blue-screen image, then the pixel from the corresponding location in the input image should be copied to the output image.
- If the pixel does lie in the blue-screen area, it is necessary to determine if the corresponding pixel in the blue-screen image is a blue background pixel or a foreground pixel.
- To decide this convert the blue-screen image pixel from (r, g, b) to (Y, U, V) representation. If (U-V) exceeds some appropriate threshold, then the pixel is a background pixel from the blue screen.
- If the pixel from *in1* is a background pixel from the blue screen, then the output pixel should be copied from the corresponding location in the input image.
- If the pixel from *in1* is an image data pixel, then the output pixel should be set to

  *(1- factor) * (bscr.r, bscrn.g. bscrn.b) + factor * (input.r, input.g, input. b);*

Input -

```
composite
{
   in0 sky.ppm
   in1 darth.ppm
   loc 300 100
   out skydarth.ppm
}
```

The *recolor* operation

***Objective* -** Transfer the color characteristics but *not* the luminance from the image specified in *in1* to the image specified in *in0*. This operation requires the use of the YCbCr representation. Begin by adding a new structure defining a YCbCr pixel to *imp.h*. The picture that is stored in the *out* image should be the *in0* image but with the *in1* color scheme.

You will need to create a structure of this type:

```
typedef struct ypix_type
{
   double Y;     // luminance
   double U;     // blue channel chrominance
   double V;     // red channel chrominance
}  ypix_t;
```

Add two new functions to *image.c*

```
void YUVtoRGB(ypix_t *pix ,pixel_t  *out);
void RGBtoYUV(pixel_t  *pix, ypix_t *out);
```

these function should transform between RGB and YUV respresentations as described in the notes.

Add a new structure to the recolor.c module. This structure will contain the information used to compute the mean and standard deviation of the U and V channels of the in1 (color source) image.

```
typedef struct stx_type
{
   double sumU;
   double sumsqrU;
   double meanU;
   double stdU;
   double sumV;
   double sumsqrV;
   double meanV;
   double stdV;
}  stx_t
```

Create two instances of this structure one for the picture source image in0 and the other for the color source image in1.

```
stx_t in0_stats;
stx_t in1_stats;
```

Complete the recolor_analyze() function which will compute statistics. It must be invoked twice, one for the in0 and one for in1.

```
void recolor_analyze(
image_t    *image,     // Image to analyze
stx_t  *stx)           // where to store results
{

}
```

For *each pixel* in the image passed in, it should convert the pixel to YUV format and accumulate the sums and sum of the squares of the U and V channels. For example:

For EVERY pixel in the image:

```
   RGBtoYUV(&in1->pixels[ndx], &ypix);
   stx->sumU  += ypix.U;
   stx->sumsqrU += ypix.U * ypix.U;
   ditto for V channel
```

To compute the U and V means just divide the U and V sums by the number of pixels in the in1 image. To compute the U standard deviation use the following formula (where count is the number of pixels in the in1 image.

```
  stx->stdU  = sqrt(stx->sumsqrU / count -
                    stx->meanU *  stx->meanU);
```

All of this should be done before recolor_ppm_pixel() is called.

The *recolor_ppm_pixel()* function

For every pixel in the output image (which has the same dimensions as the in0 picture source image),

- Convert the corresponding pixel in the in0 image to YUV format

- Renormalize the U and V components of the YUV to have the mean and standard deviations found in the output image (Use the same approach used in lab). (Note: DO NOT change the Y component!)

- Convert the modified YUV pixel back to RGB. Because of the renormalization it is quite possible that the double precision values produced by the YUVtoRGB transform might be negative or greater than 255. (Therefore your YUV_to_RGB() function must be sure to clamp values to the range [0, 255] *before storing them in the pixel_t.)*

- Store the rgb pixel_t in the output pixmap.