Embedded Software Design Techniques

C programming 3: programming techniques in image processing

Tsuyoshi Isshiki
Dept. Communications and Integrated Systems
Tokyo Institute of Technology
TAIST ICTES

Conduct by Asst. Prof. Dr. Kanjanapan Sukvichai

Lecture Outline

- Embedded software overview
  - What are “embedded systems” and “embedded software”?
- C programming 1: C language overview
  - Function, declaration, statement, expression
  - Data types, data structure, pointers and pointer dereferences
- C programming 2: algorithm complexity, program execution model
  - Bubble sort vs quick sort
  - Stack memory and program execution
- C programming 3: programming techniques in image processing
  - Dynamic memory allocation, image array implementation
  - Greyscaling, filtering, binarization, color quantization, dithering
- C programming 4: programming complex applications
  - Program development steps (ex. Huffman coding)
  - Binary tree construction, tree traversal
  - Bitstream handling
- Real time operating systems and application development
  - RTOS services, kernels
  - Context switching, task scheduling
  - Multi-task programming model

Digital Image Processing Applications

- Image compression: JPEG, JPEG200, GIF, PNG
- Image enhancement: contrast enhancement, color balancing, noise removal, deblurring (removing blurs caused by misfocus and camera movement)
- Image segmentation, object tracking, image recognition, feature extraction
- Here, we will learn the basic tools for programming image processing applicationsx

Representation of Image Data

- Image data is a 2-dimensional array of pixels

→ Yes, we can use 2-dimensional arrays in C program

- Problem with using 2D arrays for image data: image size cannot be known at compile time

→ All array declarations must have fixed values for array sizes

  - `int a[n][m];` → both **n** and **m** must be an integer CONSTANT!
  - One solution to this problem is to assume some maximum image size (in both dimensions) and declare the image array on these maximum sizes → BUT how big should the maximum image size be??
  - Another way is to use "malloc" → dynamic memory allocation

malloc and free

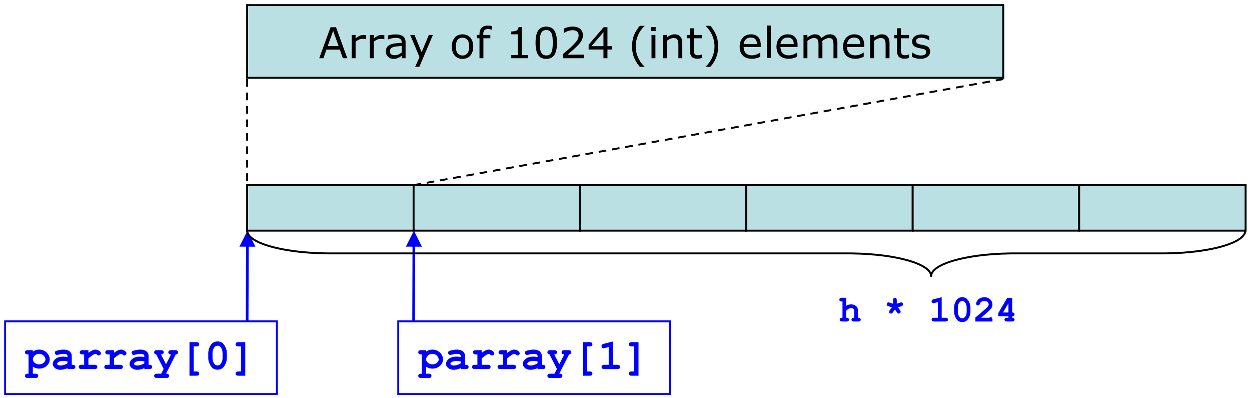
- `void * malloc(unsigned long n);`
  - Allocates a memory block of n bytes and returns the address to that memory block
  - This allocated memory block is located in the “heap memory” space, which is separate from stack memory and global memory spaces
  - Allocated memory block will exist until it is deallocated or the program terminates (will continue to exist after the function which called malloc() returns)
- `void free(void * memblock);`
  - Frees the dynamically allocated memory block by malloc() so that it can be reused in subsequent malloc() calls
  - Need to pass the same address value (memblock) as the one that malloc() returned during allocation
  - The “heap memory” manager program remembers the memory block size that it allocated

malloc and free

- "Memory leaks" occur when the allocated heap block is not "freed" in the program → can cause the heap to run out of allocation space (many commercial software products are said to have this memory leak problem)
  - Best practice is to "free" EVERY heap block in the program
- Freeing the heap block more than once can cause unpredictable behavior (usually a program crash)
- One common bug that is extremely difficult to track is when accessing a heap block which has already been "freed" → this will also result in a program crash in most cases, but this kind of bug is very hard to find

Techniques to Mimic a Multidimensional Array on Heap Data

- Pointer to an array
  - `int (* parray)[1024] = malloc(sizeof(int) * h * 1024);`
    - `malloc` allocates `h * 1024` elements of `(int)`
    - `parray` is a pointer to an array of `1024 (int)` elements which can be used as 2D array of `1024` columns and `h` rows
    - `parray[i][j]` : element at row `i`, column `j`
    - → equivalent to `*(parray[0] + i * 1024 + j)`



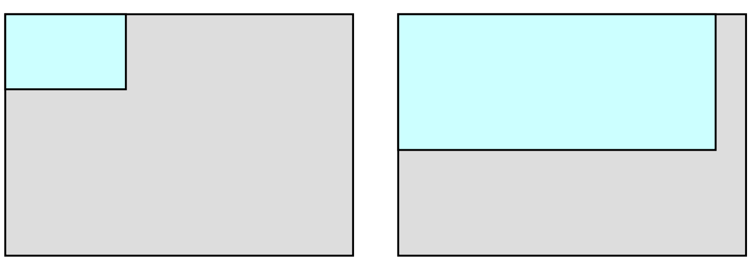
Techniques to Mimic a Multidimensional Array on Heap Data

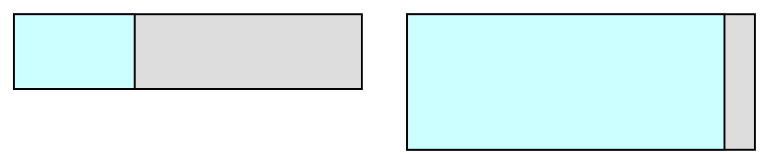
- Pointer to an array
  - `int (* parray)[1024] = malloc(sizeof(int) * h * 1024);`
    - `malloc` allocates `h * 1024` elements of `(int)`
    - `parray` is a pointer to an array of `1024` elements which can be used as 2D array of `1024` columns and `h` rows
    - `parray[i][j]` : element at row `i`, column `j`

    → equivalent to `*(parray[0] + i * 1024 + j)`

→ But this requires that the column count of the array needs to be determined at compile time as a constant

→ You can assume a maximum number of columns W (W: fixed image width), and allocate W * h (h: variable image height) elements → which can be significantly better than declaring a static array `a[H][W]` (H: fixed image height)



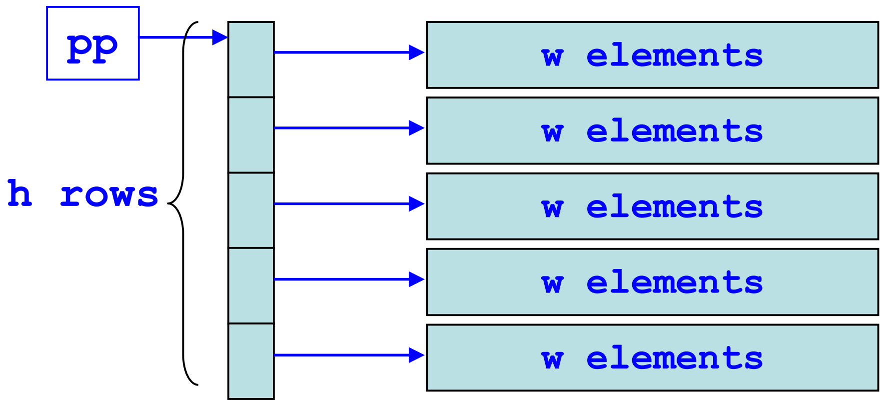


Allocating array of fixed width

Using fixed-height/width image array

Techniques to Mimic a Multidimensional Array on Heap Data

- Pointer to a pointer (allocate rows separately)
  - `int ** pp = malloc(sizeof(int *) * h);`
  - `for(i = 0; i < h; i ++)  pp[i] = malloc(sizeof(int) * w);`
    - `pp` points to an array of `h` elements of `(int *)`
    - `pp[i]` points to an array of `w` elements of `(int)`
    - `pp[i][j]` : equivalent to `*(*(pp + i) + j)`
  - We do need to pay an overhead of having `h` elements of `(int *)`, but the image size can be precisely allocated to the required size (`w * h`)
  - One problem here is that row arrays can be (usually are) segmented in the heap space → may hurt cache performance which replies on data locality



Techniques to Mimic a Multidimensional Array on Heap Data

- Pointer to a pointer (allocate the entire 2D array)
  - `int ** pp = malloc(sizeof(int *) * h);`
  - `int * temp = malloc(sizeof(int) * h * w);`
  - `for(i = 0; i < h; i ++)  pp[i] = temp + w * i;`
    - `pp` points to an array of `h` elements of `(int *)`
    - `temp` points to an array of `h * w` elements of `(int)`
    - `pp[i]` points to `(w * i)`–th element of `temp`
    - `pp[i][j]` : equivalent to `*(*(pp + i) + j)`
  - Here, you will be "freeing" `pp[0]` which is pointing to `temp` (`temp` does not need to be saved)
  - This technique is often used in reference codes for image/video standards

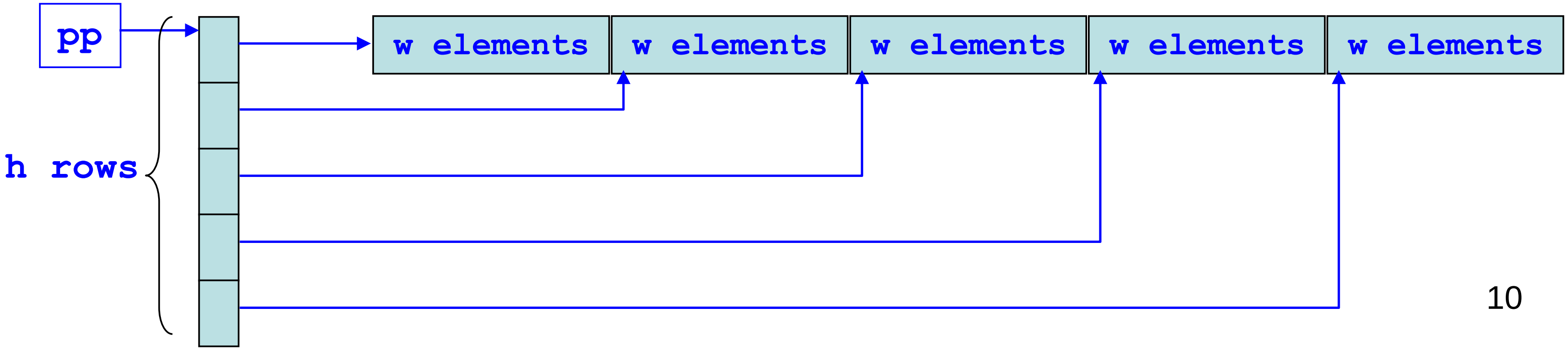


Image processing on 1D Array

- Even though accessing the image data as 2D array can make the program easier to understand, it can add substantial overhead in calculating memory addresses
  - Some redundant address calculation can be eliminated by smart compilers, but this really is compiler dependent and very hard to tune on the code
- So in order to write efficient (fast) image processing programs, we often use 1D array representation and keep track of row address and column address in the program → This is how we will do it in this lecture

BMP Image File Format

- File name : *.bmp → standard image file format used on Windows
- Colormap(pallete): an array of RGB values that translates an index value into RGB data (R,G,B: 8-bit value [0, 255])
- Pixel: represented in 1, 4, 8, 24 bits
  - 1-bit format : 1-bit index to 2-element colormap
  - 4-bit format : 4-bit index to 16-element colormap
  - 8-bit format : 8-bit index to 256-element colormap
  - 24-bit format : 24-bit RGB values (does not use colormap)

  → 4-bit/8-bit formats can use simple run-length encoding for compressing the index values (effective for graphics images where there are large regions with same RGB values)
- File format:
  - BMP header:
    - "BM" identifier, file size, bitmap data location
  - Bitmap info:
    - Image size (width, height), pixel format, compression type, colormap size, etc.
  - Colormap data: 1/4/8-bit formats only
  - Bitmap data: colormap index values (1/4/8-bit) or RGB values

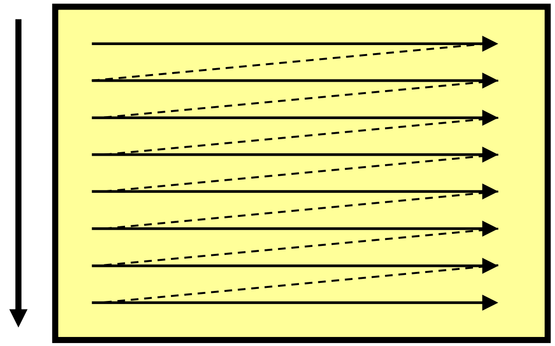
The basic file structure is binary (as opposed to a text file) and is broken into the following four sections:

- The File Header (14 bytes)
    - Confirms that the file is (at least probably) a BMP file.
    - Tells exactly how large the file is.
    - Tells where the actual image data is located within the file.
- The Image Header (40 bytes in the versions of interest)
    - Tells how large the image is (rows and columns).
    - Tells what format option is used (bits per pixel).
    - Tells which type of compression, if any, is used.
    - Provides other details, all of which are seldom used.
- The Color Table (length varies and is not always present)
    - Provides the color palette for bit depths of 8 or less.
    - Provides the (optional) bit masks for bit depths of 16 and 32.
    - Not used for 24-bit images.
- The Pixel Data
    - Pixel by pixel color information
    - Row-by-row, bottom to top.
    - Rows start on double word (4-byte) boundaries and are null padded if necessary.
    - Each row is column-by-column, left to right.
    - In 24-bit images, color order is Red, Green, Blue.
    - In images less than 8-bits, the higher order bits are the left-most pixels.

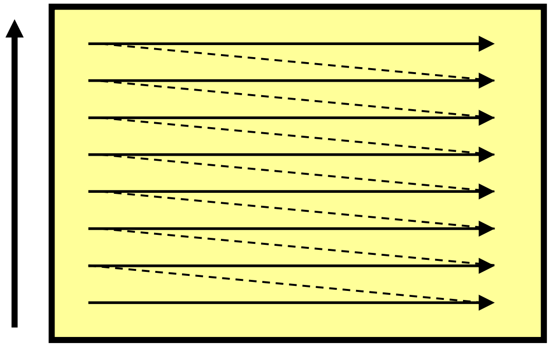
BMP Image File Format

- Image pixel position coordinate:
  - pixel(0, 0) is located at the top-left corner of the image
  - pixel(i, j) is at the i-th row, j-th column
- Image scanning order:

  (W: image width, H: image height)
  - General scanning order: top-left to bottom-right, column-first
    - (0, 0), (0, 1), (0, 2) ... , (0, W – 1), (1, 0), (1, 1), ...
  - BMP scanning order: bottom-left to top-right, column-first
    - (H – 1, 0), (H – 1, 1), ... , (H – 1, W – 1), (H – 2, 0), ...



General scanning order



BMP scanning order

Simple Data Structure and APIs for Handling BMP Images (bmp.h)

```
typedef struct pixel
{
      unsigned char R, G, B, I;
} PIX;

typedef struct bmp_info
{
      PIX * bitmap;
      PIX colormap[256];
      int colormap_size;
      int size_x, size_y, size_xy;
      int out_pix_bits;
} BMPIMG;

void BMPIMG_initialize(BMPIMG * img);
void BMPIMG_destroy(BMPIMG * img);
int BMPIMG_save(BMPIMG * img);
int BMPIMG_open(BMPIMG * img, const char * fname);
```

PIX : holds RGB pixel values (R,G,B) as well as colormap index values (I)

PIX * bitmap : points to an array of **PIX** which holds the image data → this array is dynamically created by **malloc()**

size_x : image width
size_y : image height
size_xy : = **size_x * size_y**
out_pix_bits : bits/pixel for output image

BMPIMG_initialize()
BMPIMG_open()

```
void BMPIMG_initialize(BMPIMG * img)
{
        img->bitmap = 0;
        img->colormap_size = 0;
        img->size_x = 0;
        img->size_y = 0;
        img->size_xy = 0;
        img->out_pix_bits = 24;
}
```

During initialization, **`bitmap`** array is not allocated (because the image size is unknown)
`out_pix_bits` is set to 24 (bits)

```
int BMPIMG_open(BMPIMG * img, const char * fname)
{
        FILE * fp;
        if(!check_bmp_file_extension(fname)){
           printf("file <%s> does not have .bmp file extension\n", fname);
           return 0;
        }
        fp = fopen(fname, "rb");
        if(fp == 0){
           printf("cannot open file <%s>\n", fname);
           return 0;
        }
        return BMPIMG_openBMP(img, fp);
}
```

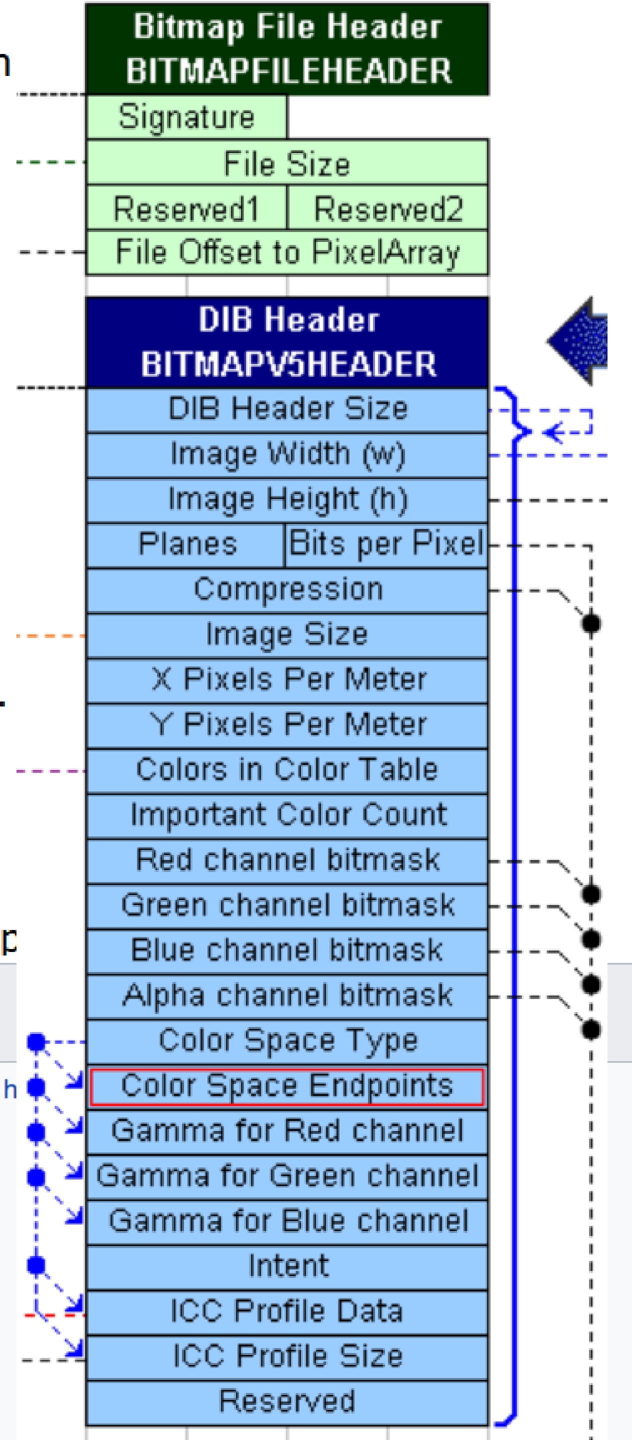
`BMPIMG_openBMP()` : core function for reading BMP file

The basic file structure is binary (as opposed to a text file) and is broken

- The File Header (14 bytes)
  - Confirms that the file is (at least probably) a BMP file.
  - Tells exactly how large the file is.
  - Tells where the actual image data is located within the file.
- The Image Header (40 bytes in the versions of interest)
  - Tells how large the image is (rows and columns).
  - Tells what format option is used (bits per pixel).
  - Tells which type of compression, if any, is used.
  - Provides other details, all of which are seldom used.
- The Color Table (length varies and is not always present)
  - Provides the color palette for bit depths of 8 or less.
  - Provides the (optional) bit masks for bit depths of 16 and 32.
  - Not used for 24-bit images.
- The Pixel Data
  - Pixel by pixel color information
  - Row-by-row, bottom to top.
  - Rows start on double word (4-byte) boundaries and are null p

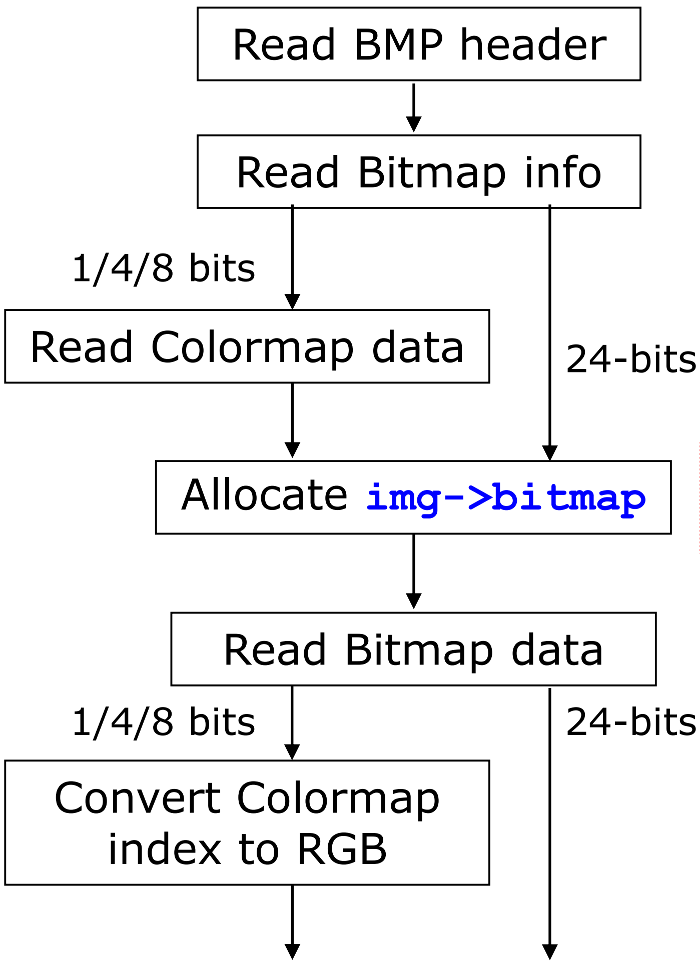
Offset hex	Offset dec	Size	Purpose
00	0	2 bytes	The header field used to identify the BMP and DIB file is `0x42 0x4D` in h entries are possible: **BM** Windows 3.1x, 95, NT, ... etc. **BA** OS/2 struct bitmap array **CI** OS/2 struct color icon

Check for Bit Map format



BMPIMG_openBMP()

```
int BMPIMG_openBMP(BMPIMG * img, FILE * fp);
```



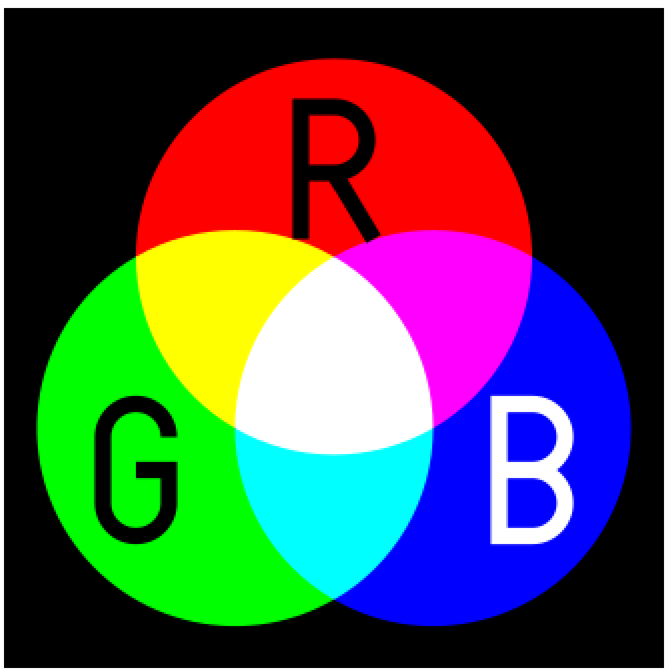
File size, Bitmap location

Image size, pixel format, colormap size, etc
Set **BMPIMG** fields:
size_x, size_y, size_xy, colormap_size

```
img->bitmap =
   (PIX *) malloc(sizeof(PIX) * img->size_xy);
```

```
/* read the file type (first two bytes) */
c = getc(fp);  c1 = getc(fp);    // Check if it Bit Map format or not
if (c!='B' || c1!='M') {
    printf("File type is not 'BM' (%c%c)", c, c1);
    fclose(fp);
    return 0;
}

bfSize = getint(fp);
getshort(fp);          /* reserved and ignored */
getshort(fp);
bfOffBits = getint(fp);
biSize         = getint(fp);
```

```c
typedef struct pixel
{
    unsigned char R, G, B, I;
} PIX;

typedef struct bmp_info
{
    PIX * bitmap;
    PIX colormap[256];
    int colormap_size;
    int size_x, size_y, size_xy;
    int out_pix_bits;    /* valid values are: 24, 8, 4, 1 */
} BMPIMG;
```



RGB color format & calculation

RGB code has 24 bits format (bits 0..23):

RED[7:0]		GREEN[7:0]		BLUE[7:0]	
23	16	15	8	7	0

```
RGB = (R*65536)+(G*256)+B , (when R is RED, G is GREEN and B is BLUE)
```

Calculation examples

White RGB Color

```
White RGB code = 255*65536+255*256+255 = #FFFFFF
```

Blue RGB Color

```
Blue RGB code = 0*65536+0*256+255 = #0000FF
```

BMPIMG_save()
BMPIMG_destroy()

```
int BMPIMG_save(BMPIMG * img);
```

- Output file name: fixed as “result.bmp”
- `img->out_pix_bits` determines the pixel format of result.bmp
  - 1/4/8 bits: stores colormap index values for each pixel
    - `colormap_size == (1 << img->out_pix_bits)` must be satisfied
    - Appropriate colormap values must be set before saving BMPIMG
  - 24 bits: stores RGB values for each pixel
    - colormap_size is ignored

```
int BMPIMG_destroy(BMPIMG * img){
    if (img->bitmap != 0){
        free(img->bitmap);
        img->bitmap = 0;
    }
}
```

  - Deallocate image array (to avoid memory leak)

Typical Program Flow Using BMPINFO API

```
int main(int argc, char * argv[])
{
   BMPIMG img;
   BMPIMG_initialize(&img);
   if(!BMPIMG_open(&img, argv[1])){
       printf("open file <%s> failed\n", argv[1]);
       return 0;
   }

   /** do some image processing here **/
   BMPIMG_save(&img); /** writes to result.bmp **/
   BMPIMG_destroy(&img);
   return 1;
}
```

If you don’t do any processing here, result.bmp will be identical to the original file (but in 24-bit format)

Let's Enjoy Image Processing!!

- Greyscale conversion
- Simple filtering (averaging)
- Binarization
- Color quantization
- Dithering

Greyscale Conversion

- Greyscale image is a black-and-white image without any colors
- Each pixel consists of 1 component which represents “light intensity” (brightness)
- Conversion from RGB to greyscale
  - Simple scheme: average RGB values
    - pix->I = (pix->R + pix->G + pix->B) / 3;
  - A formal conversion equation
    - pix->I = (unsigned char) (pix->R * 0.299 + pix->G * 0.587 + pix->B * 0.114);
- We will use the “I” field of PIX to represent the intensity, which is used to index the colormap
- Greyscale colormap:
  - White is R = G = B = 255
  - Black is R = G = B = 0
  - So at **`colormap[i]`**, RGB values should all be **`i (i = 0, 1, ..., 255)`**

Grey-scale Conversion

```
void BMPIMG_greyscale1(BMPIMG * img)
{
     int i;
     PIX * pix;
     /* set greyscale colormap */
     for(i = 0; i < 256; i ++){
        img->colormap[i].B = i;
        img->colormap[i].G = i;
        img->colormap[i].R = i;
     }
     img->out_pix_bits = 8;
     img->colormap_size = 256;
     pix = img->bitmap;
     for(i = 0; i < img->size_xy; i ++, pix ++){
#if 1   /* definition of Y (luminance) */
        pix->I = (unsigned char) (pix->R * 0.299 + pix->G * 0.587
               + pix->B * 0.114);
#else   /* simple average of RGB */
        pix->I = (pix->R + pix->G + pix->B) / 3;
#endif
     }
}
```

This part forces the output image (result.bmp) to be stored in 8-bit format and use the colormap

Here, make sure that:
`(1<<out_pix_bits) == colormap_size`

Grey-scale Conversion

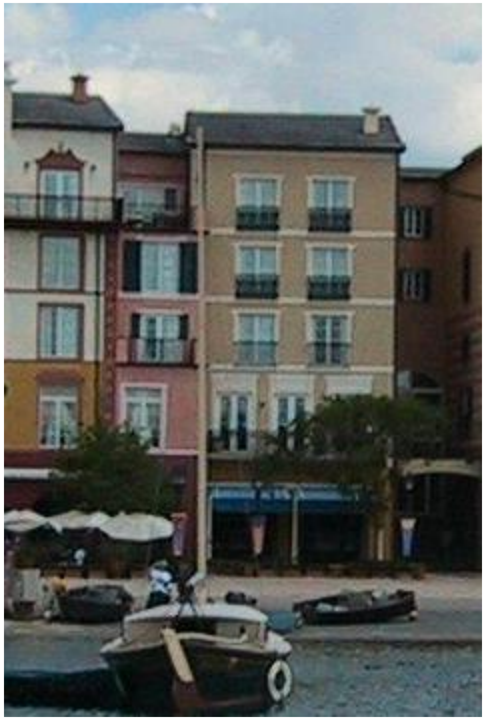
```
void BMPIMG_greyscale1(BMPIMG * img)
{
      ......
      pix = img->bitmap;
      for(i = 0; i < img->size_xy; i ++, pix ++){
#if 1   /* definition of Y (luminance) */
         pix->I = (unsigned char) (pix->R * 0.299 + pix->G * 0.587
                + pix->B * 0.114);
#else   /* simple average of RGB */
         pix->I = (pix->R + pix->G + pix->B) / 3;
#endif
      }
}
```

used by PAL and NTSC, the rec601 luma (Y') component is computed as

$$Y' = 0.299R' + 0.587G' + 0.114B'$$

Notice here that we only have one loop to do this processing → pixel-wise processing is very easy to code with 1D array image structure

Grey-scale Conversion

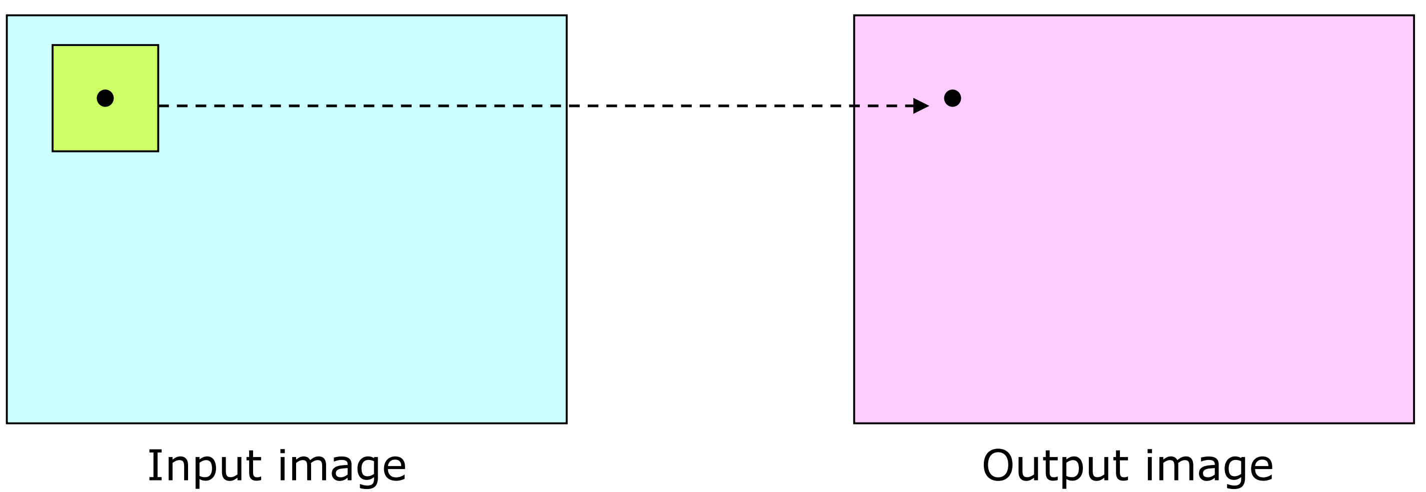


original image

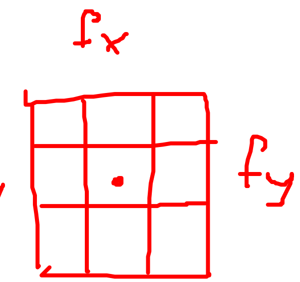
result

Filtering

- Here, we consider a non-recursive filter: the output pixel is computed only from the pixels in the input image → FIR (finite impulse response) filter
- The filter works on a "window" centered around the current pixel and computes an output pixel
  - $out\_pix(i, j) = \Sigma_k \Sigma_l w(k, l) * pix(i - k, j - l)$
  - Here, we will do simply average the window pixels



Average Filter




```
void BMPIMG_average(BMPIMG * img, int fx, int fy)
{
      int i, j, k, l;
      int sx = (fx + 1) / 2, sy = (fy + 1) / 2;
      int area = fx * fy;
      PIX * pix, * out_pix;
      PIX * out_buf = (PIX *) malloc(img->size_xy * sizeof(PIX));
      if(out_buf == 0){
         printf("malloc failed in BMPIMG_lowpass!\n");
         return;
      }
      pix = img->bitmap;
      out_pix = out_buf;

      /*** filtering takes place here (next page) ***/
      /*** (img) --> (out_buf) ***/

      pix = img->bitmap;
      img->bitmap = out_buf;
      free(pix);
}
```

Allocate the output image array

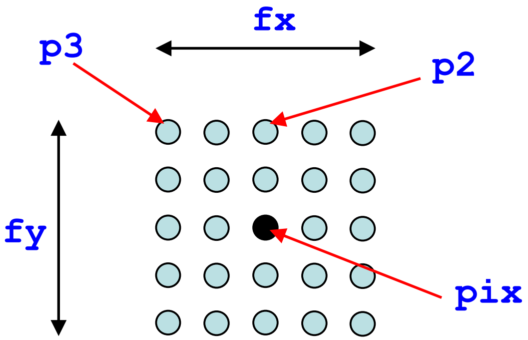
Next page →

Set the output image array to **img->bitmap**, and deallocate the original input image array → No need to copy back the image!!

Average Filter (core processing)

```
pix = img->bitmap;
out_pix = out_buf;
for(i = 0; i < img->size_y; i ++){
      for(j = 0; j < img->size_x; j ++, pix ++, out_pix ++){
         PIX * p2 = pix + (sy - fy) * img->size_x;
         int r = 0, g = 0, b = 0;
         for(k = sy - fy; k < sy; k ++, p2 += img->size_x){
                  PIX * p3 = p2 + (sx - fx);
                  if(i + k < 0 || i + k >= img->size_y)
                           continue;
                  for(l = sx - fx; l < sx; l ++, p3 ++){
                           if(j + l < 0 || j + l >= img->size_x)
                                    continue;
                           r += p3->R;
                           g += p3->G;
                           b += p3->B;
                  }
         }
         out_pix->R = r / area;
         out_pix->G = g / area;
         out_pix->B = b / area;
      }
}
```

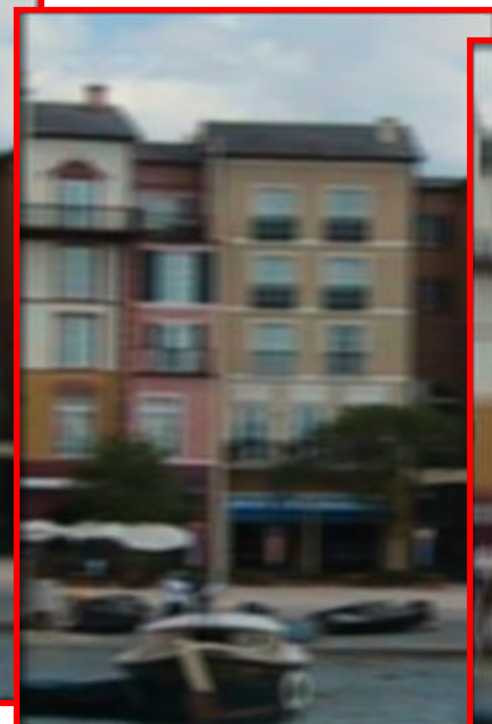
```
int area = fx * fy;
```



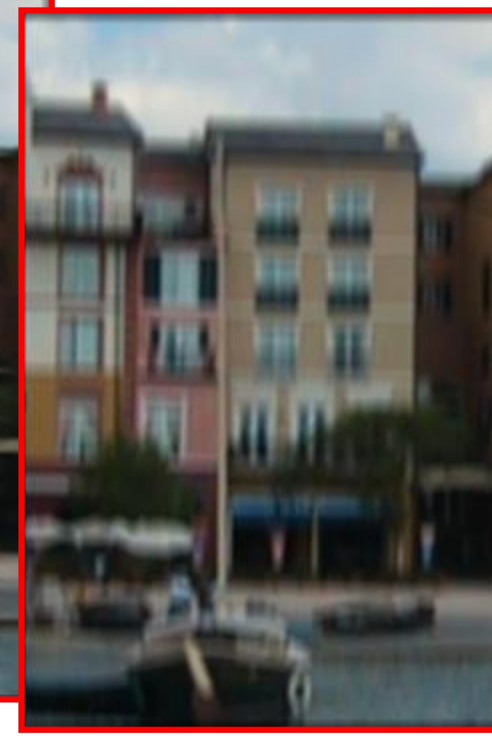
Average Filter

original image

fx = 5, fy = 5



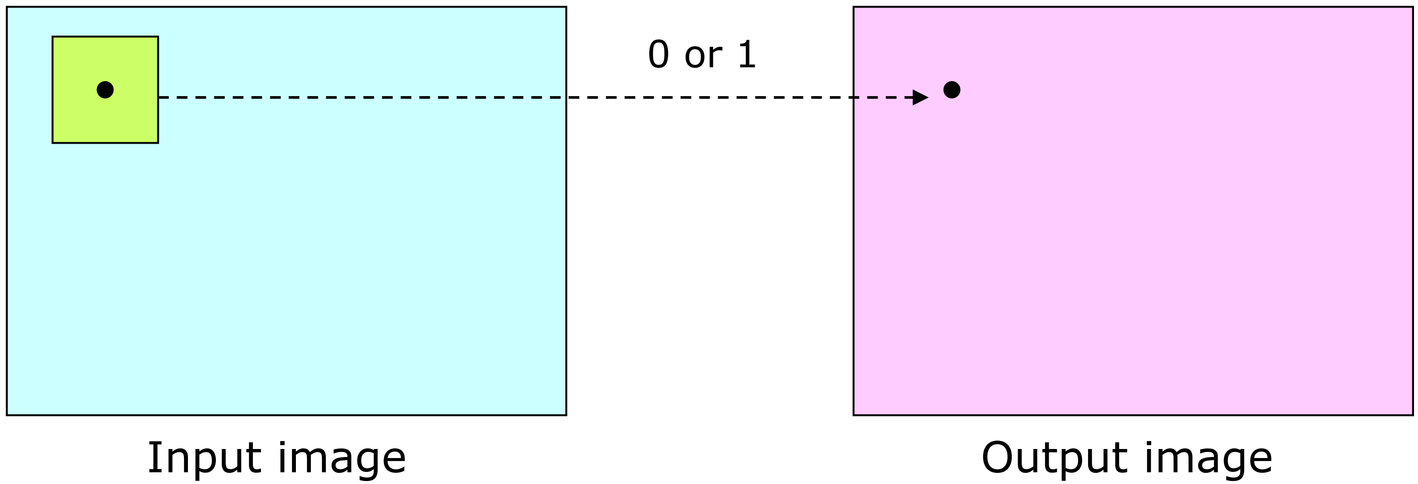
fx = 10, fy = 3



fx = 3, fy = 10

Binarization

- Binarization is a process of coverting the greyscale image into 1-bit pixel image (1→ white, 0 → black)
- Binarization has many applications in image recognition, and computer vision
- Criteria for binarization depends on the application, but here, we will consider a simple case of applying a threshold to determine 0 or 1
- Here, threshold is calculated as the average intensity of the window region surrounding the pixel



Binarization (core processing)

```
BMPIMG_greyscale1(img);
pix = img->bitmap;
out_pix = out_buf;
for(i = 0; i < img->size_y; i ++){
      for(j = 0; j < img->size_x; j ++, pix ++, out_pix ++){
         PIX * p2 = pix + (sy - fy) * img->size_x;
         int sum = 0;
         for(k = sy - fy; k < sy; k ++, p2 += img->size_x){
                  PIX * p3 = p2 + (sx - fx);
                  if(i + k < 0 || i + k >= img->size_y)
                           continue;
                  for(l = sx - fx; l < sx; l ++, p3 ++){
                           if(j + l < 0 || j + l >= img->size_x)
                                    continue;
                           sum += p3->I;
                  }
         }
         out_pix->I = (pix->I >= sum / area) ? 255 : 0;
      }
}
```

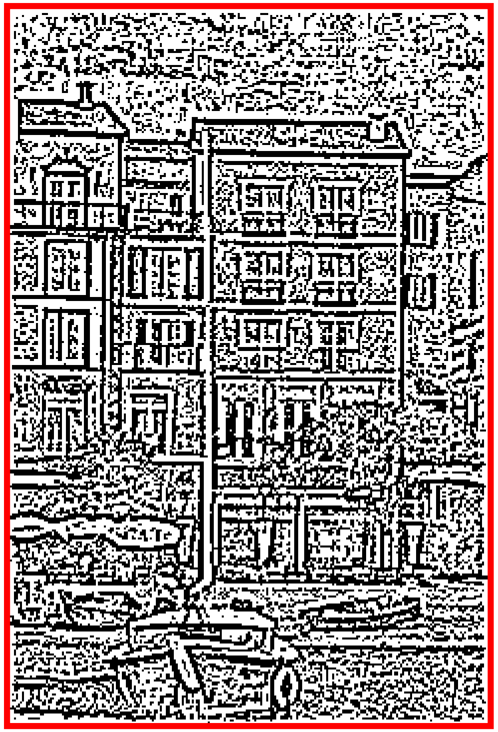
Convert to greyscale image

`sum / area` is the average intensity of the window region

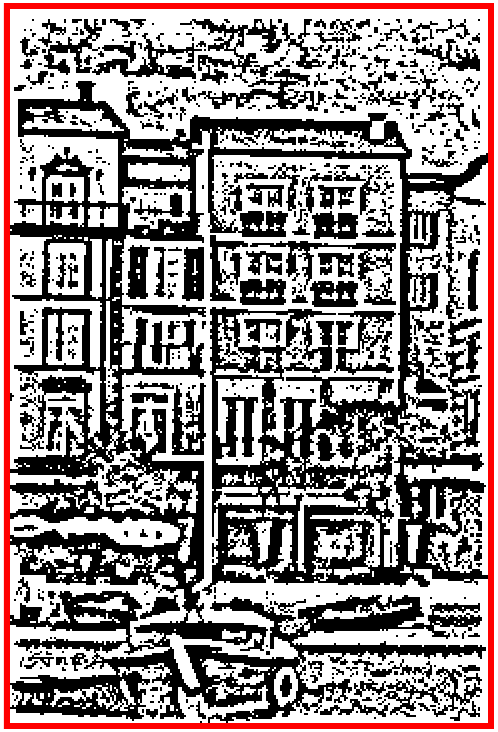
The basic code structure is the same as BMPIMG_average()

Binarization

greyscale



fx = 5, fy = 5



fx = 10, fy = 10

Color Quantization

- Color quantization reduces the number of bits per pixel from the raw 24-bit RGB data by quantizing the RGB components
- Most straightforward scheme of color quantization is to quantize individual RGB components separately
- To quantize to 8 bits per pixel, we can assign the number of bits to each components : R = 3 bits, G = 3 bits, B = 2 bits (3 bits = 8 quantization levels, 2 bits = 4 quantization levels)
- A slightly more generalized scheme is to specify the quantization levels for each RGB component while keeping the number of “combinations” no greater than 256 : R = 6 levels, G = 7 levels, B = 6 levels → 6 * 7 * 6 = 252 < 256
- Quantization equation :
  - Q : quantization level, 0 <= x <= 255, 0 <= x_q <= Q – 1
  - x_q = x * Q / 256 = floor(x * Q / 256.0)
  - Range of x value that quantizes to x_q :
    - min(x → x_q) = ceil(x_q * 256.0 / Q) = (x_q * 256 + Q – 1) / Q
    - max(x → x_q) = min(x → x_q + 1) – 1
= ((x_q + 1) * 256 + Q – 1) / Q – 1

Color Quantization

```
void BMPIMG_create_colormap(BMPIMG * img, int RQ, int GQ, int BQ)
{
        int i, j, k;
        if(RQ * GQ * BQ > 256 || RQ <= 0 || GQ <= 0 || BQ <= 0){
            printf("Q-levels out of range!!! (%d * %d * %d = %d)\n",
                        RQ, GQ, BQ, RQ * GQ * BQ);
            return;
        }
        for(i = 0; i < RQ; i ++){
            int r_min = (i * 256 + RQ - 1) / RQ;
            int r_max = ((i + 1) * 256 + RQ - 1) / RQ - 1;
            for(j = 0; j < GQ; j ++){
                        int g_min = (j * 256 + GQ - 1) / GQ;
                        int g_max = ((j + 1) * 256 + GQ - 1) / GQ - 1;
                        for(k = 0; k < BQ; k ++){
                                    int b_min = (k * 256 + BQ - 1) / BQ;
                                    int b_max = ((k + 1) * 256 + BQ - 1) / BQ - 1;
                                    int cid = i * GQ * BQ + j * BQ + k;
                                    img->colormap[cid].R = (r_min + r_max) / 2;
                                    img->colormap[cid].G = (g_min + g_max) / 2;
                                    img->colormap[cid].B = (b_min + b_max) / 2;
                                    printf("%3d: R(%3d,%3d) G(%3d,%3d) B(%3d,%3d)\n", cid,
                                                r_min, r_max, g_min, g_max, b_min, b_max);
                        }
            }
        }
}
```

Colormap value is simply calculated as the middle point between min and max

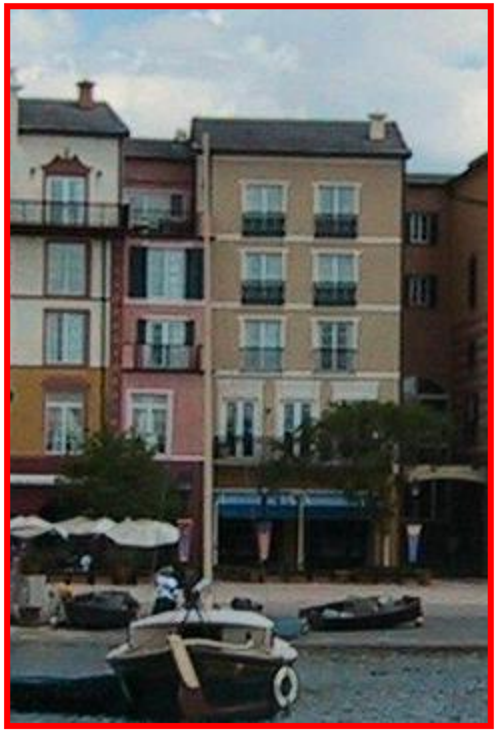
Colormap space is configured as 3D array: colormap[RQ][GQ][BQ]

Color Quantization

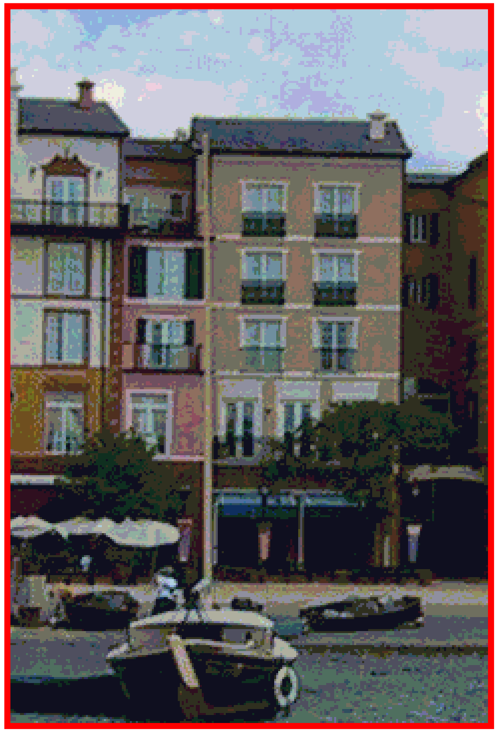
```
void BMPIMG_convert_to_colormap(BMPIMG * img, int RQ, int GQ, int BQ)
{
      int i;
      PIX * pix;
      for(i = 0, pix = img->bitmap; i < img->size_xy; i ++, pix ++){
         pix->I = (pix->R * RQ / 256) * GQ * BQ +
                  (pix->G * GQ / 256) * BQ +
                  (pix->B * BQ / 256);
      }
      img->out_pix_bits = 8;
      img->colormap_size = 256;
}
```

Colormap space is configured as 3D array: colormap[RQ][GQ][BQ]

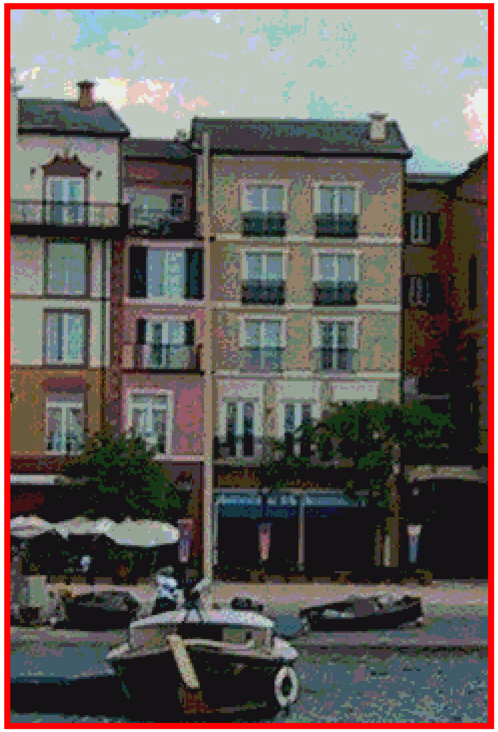
Color Quantization



original image



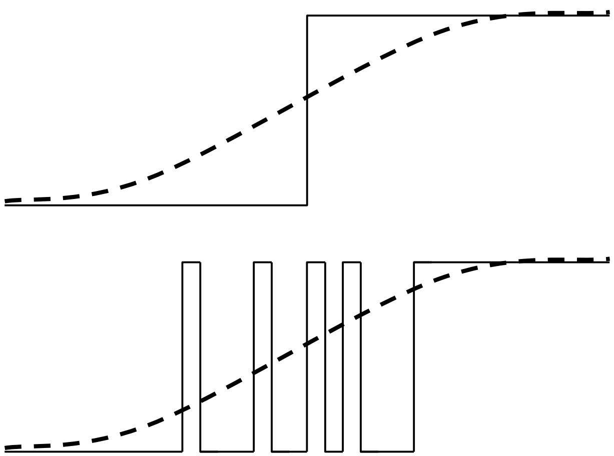
RQ = GQ = 8, BQ = 4
(colormap[256])



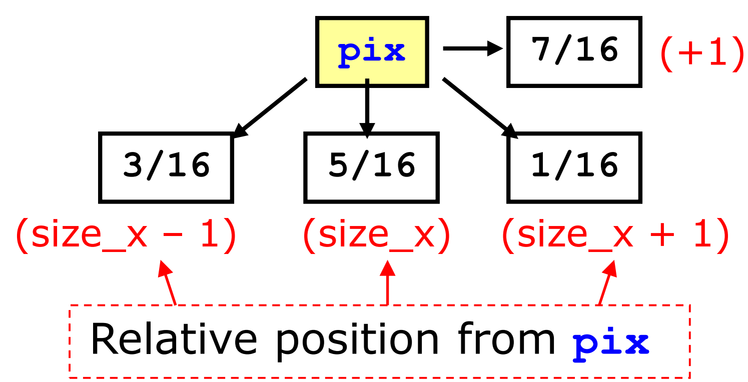
RQ = BQ = 6, GQ = 7
(colormap[252])

Dithering

- Dithering is a technique that is applied to quantized image which reduces the visible contour patterns by scattering the quantization errors to the nearby pixels (error diffusion)
- 2D error diffusion : quantization error is distributed to right, lower-left, lower, lower-right pixels (these pixels are not quantized yet)



Processing proceeds from left to right, top to bottom

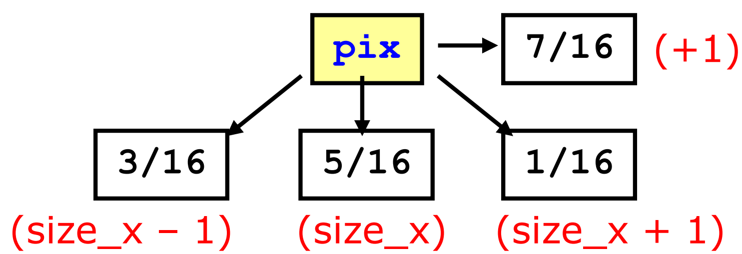


Dithering

```
void BMPIMG_dither(BMPIMG * img, int RQ, int GQ, int BQ)
{
      int i, j, eR, eG, eB;
      PIX * pix;
      for(i = 0, pix = img->bitmap; i < img->size_y; i ++){
         for(j = 0; j < img->size_x; j ++, pix ++){
                pix->I = (pix->R * RQ / 256) * GQ * BQ +
                         (pix->G * GQ / 256) * BQ +
                         (pix->B * BQ / 256);
                eR = pix->R - img->colormap[pix->I].R;
                eG = pix->G - img->colormap[pix->I].G;
                eB = pix->B - img->colormap[pix->I].B;
                if(j < img->size_x - 1) DIF_ERR(1, 7);
                if(i < img->size_y - 1){
                         if(j > 0) DIF_ERROR(img->size_x - 1, 3);
                         DIF_ERROR(img->size_x, 5);
                         if(j < img->size_x - 1)
                                  DIF_ERROR(img->size_x + 1, 1);
                }
         }
      }
      img->out_pix_bits = 8;
      img->colormap_size = 256;
}
```

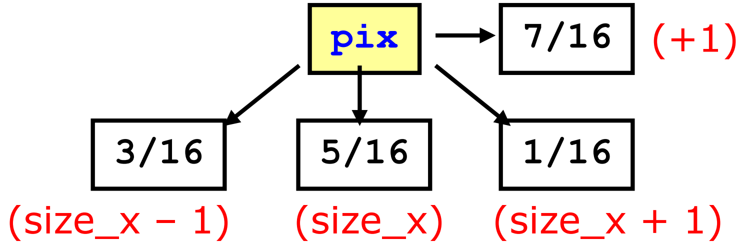
eR, eG, eB : quantization error

DIF_ERROR(pos, coef)
→ macro call

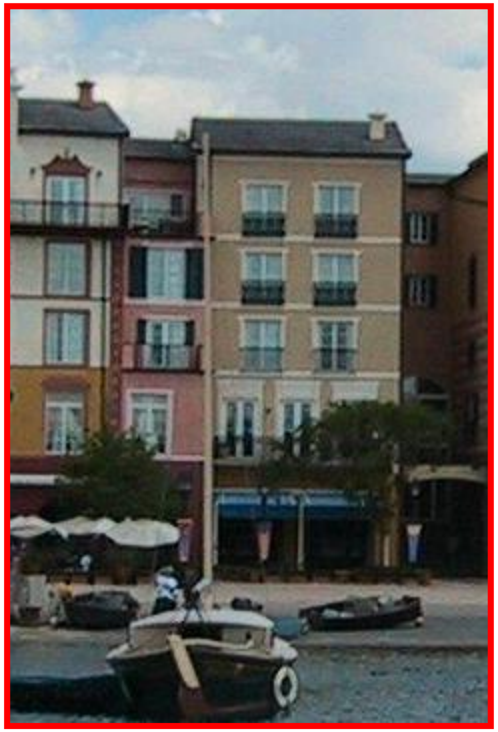


Dithering

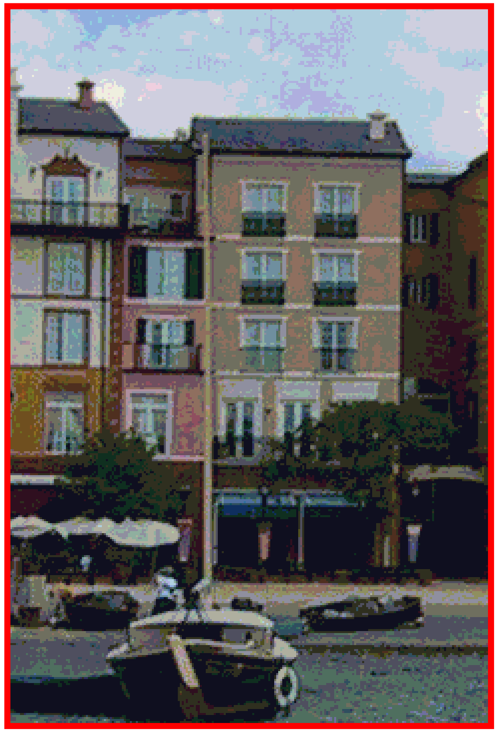
```
#define DIF_ERR(pos, coef)                              \
        {(pix + pos)->R += eR * coef / 16;    \
        (pix + pos)->G += eG * coef / 16;     \
        (pix + pos)->B += eB * coef / 16;}

        ......

        if(j < img->size_x - 1) DIF_ERR(1, 7);
        if(i < img->size_y - 1){
                if(j > 0) DIF_ERROR(img->size_x - 1, 3);
                DIF_ERROR(img->size_x, 5);
                if(j < img->size_x - 1)
                        DIF_ERROR(img->size_x + 1, 1);
        }
        ......
```



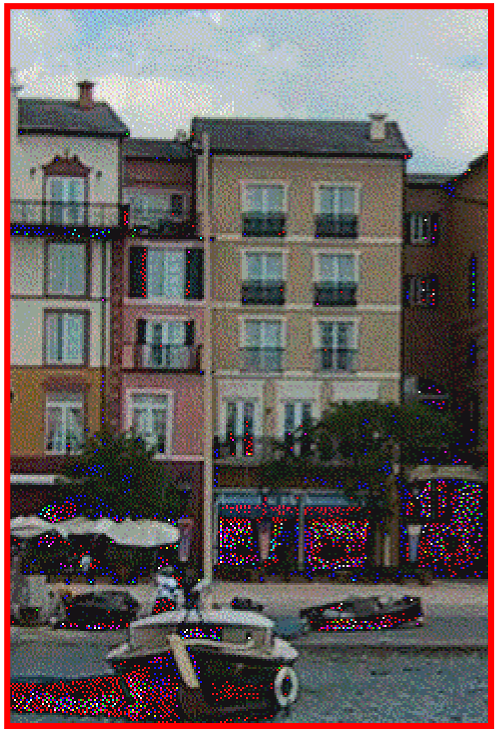
Dithering



original image

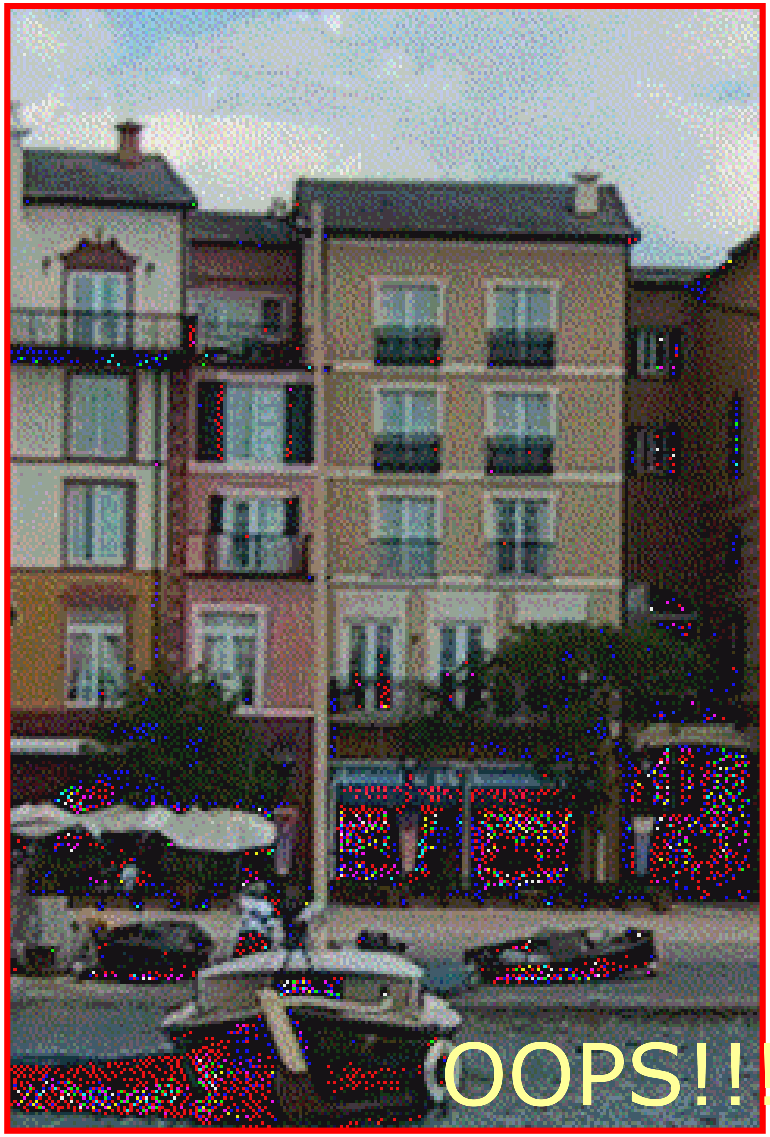


RQ = BQ = 6, GQ = 7
(no dithering)



RQ = BQ = 6, GQ = 7
(with dithering)
OOPS!!!

Dithering



This kind of bug often happens in image processing programs

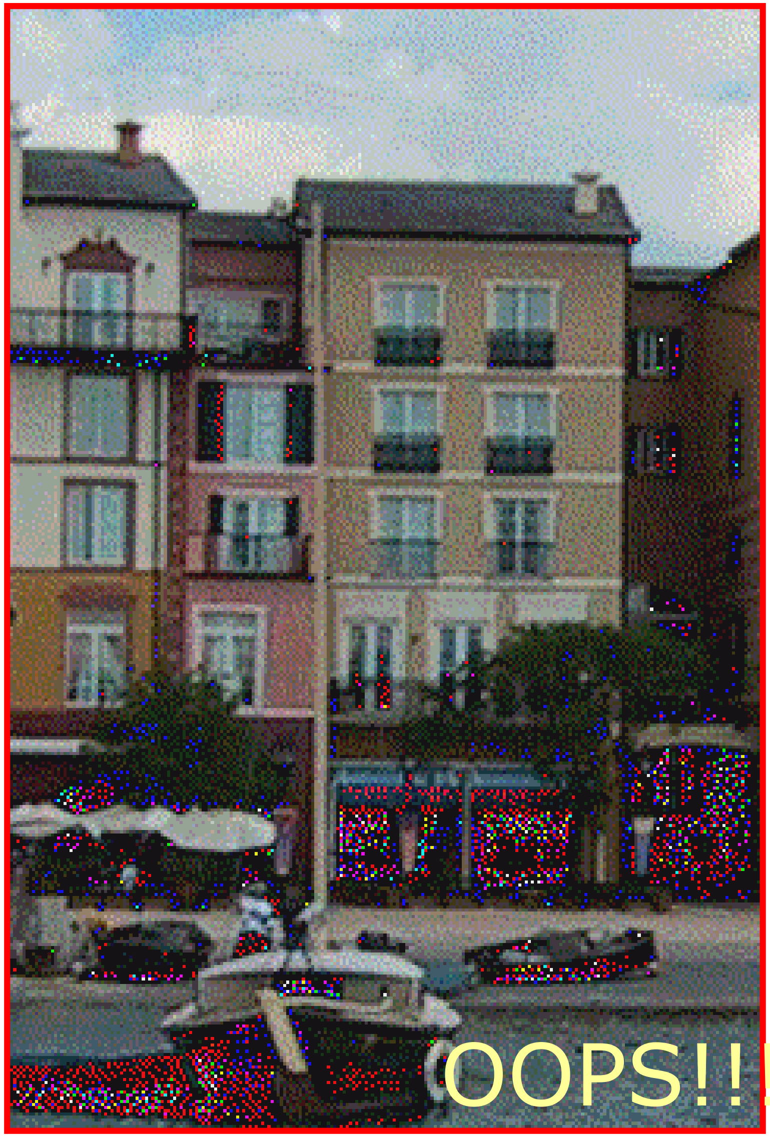
Be careful with the dynamic range of pixels!!

Dithering

```
#if 1
#define CLAMP(pp) ((pp > 255) ? 255 : (pp < 0) ? 0 : pp)

#define DIF_ERR(pos, coef)                              \
        { int pp;                                       \
        pp = (pix + pos)->R + eR * coef / 16;           \
        (pix + pos)->R = CLAMP(pp);                     \
        pp = (pix + pos)->G + eG * coef / 16;           \
        (pix + pos)->G = CLAMP(pp);                     \
        pp = (pix + pos)->B + eB * coef / 16;           \
        (pix + pos)->B = CLAMP(pp);    }
#else
#define DIF_ERR(pos, coef)                              \
        {(pix + pos)->R += eR * coef / 16;              \
        (pix + pos)->G += eG * coef / 16;               \
        (pix + pos)->B += eB * coef / 16;}
#endif
```

`CLAMP(pp)`: keeps `pp` between 0 and 255

Dithering



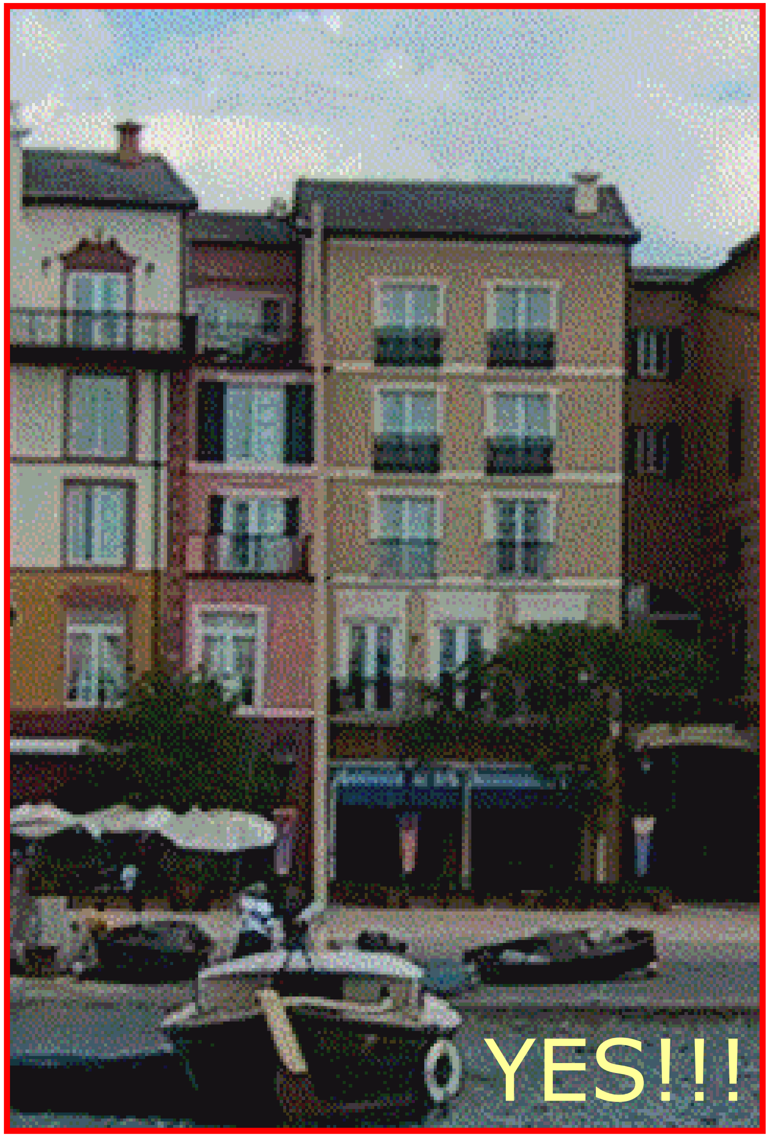


Image processing in Python?

Python provides lots of libraries for image processing, including −

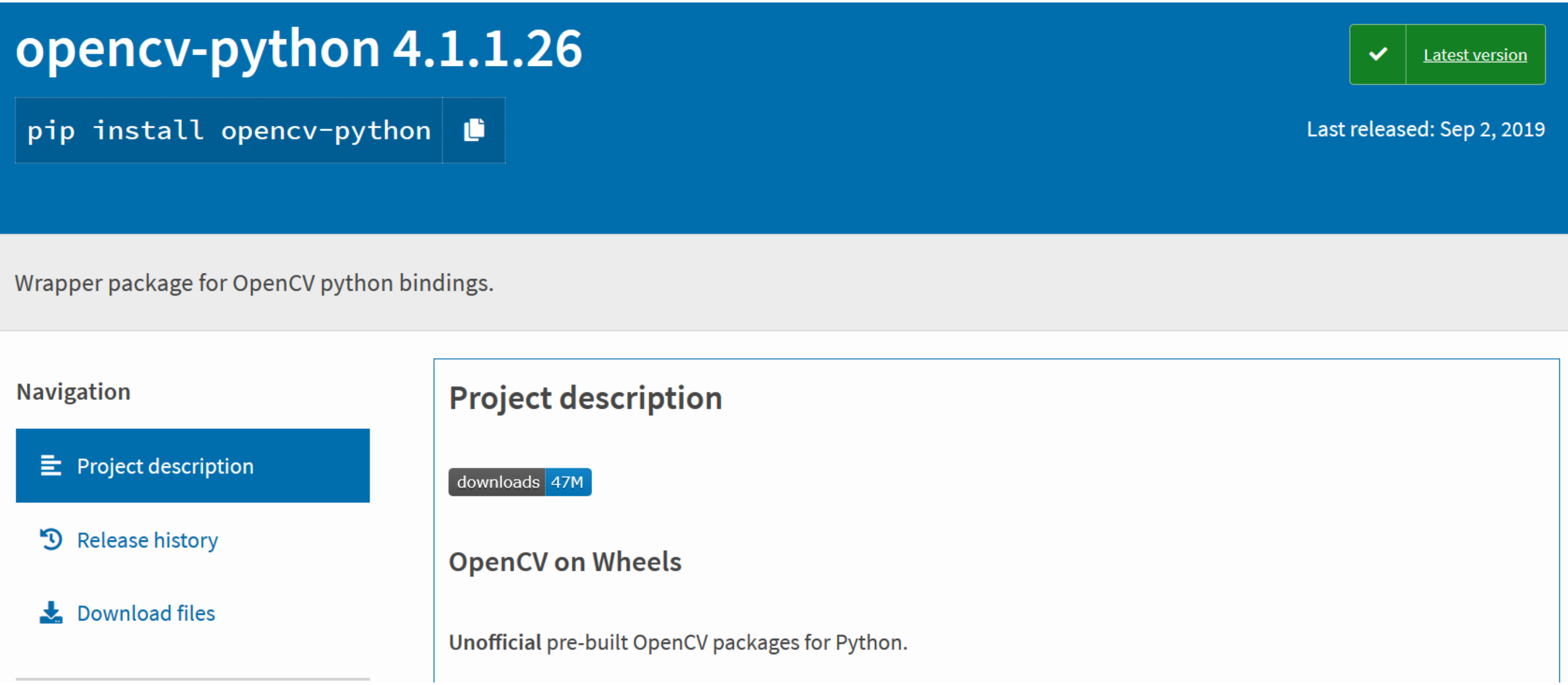
•**OpenCV** − Image processing library mainly focused on real-time computer vision with application in wide-range of areas like 2D and 3D feature toolkits, facial & gesture recognition, Human-computer interaction, Mobile robotics, Object identification and others.

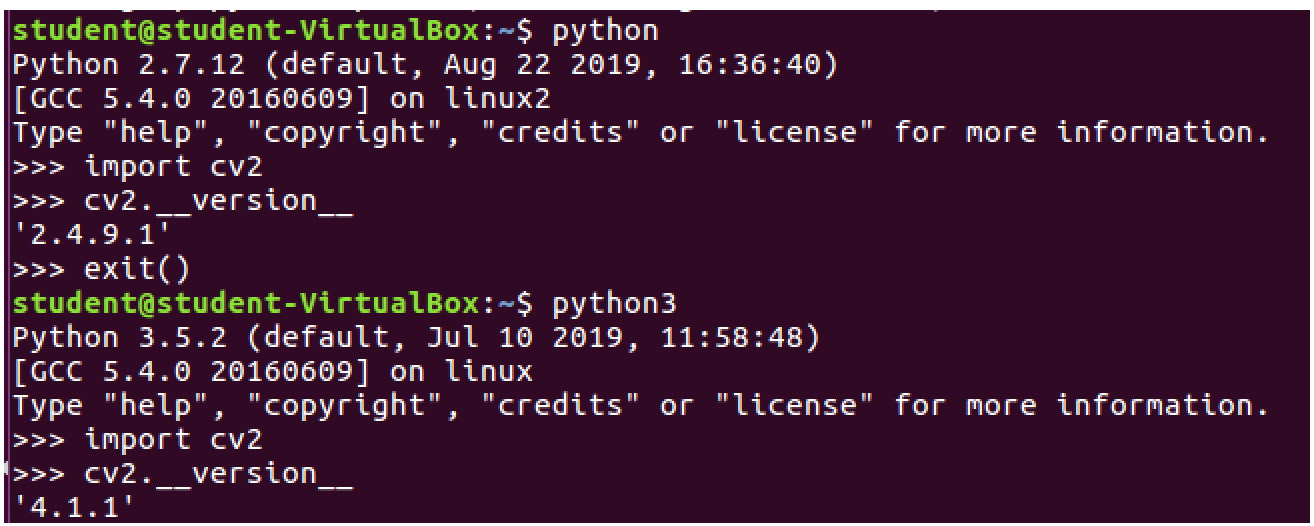
•**Numpy and Scipy libraries** − For image manipuation and processing.

•**Sckikit** − Provides lots of alogrithms for image processing.

•**Python Imaging Library (PIL)** − To perform basic operations on images like create thumnails, resize, rotation, convert between different file formats etc.

sudo apt-get install python-pip python3-pip
pip install pillow **or** pip3 install pillow
sudo apt-get install python-opencv **or** pip3 install opencv-python



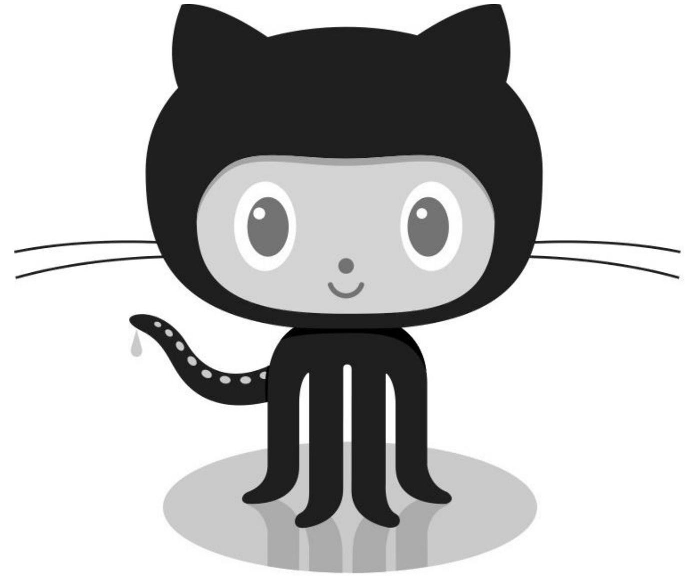


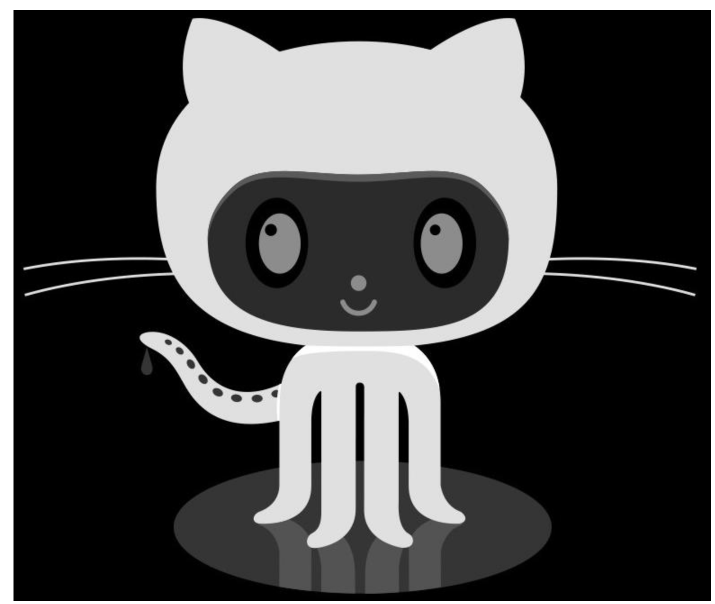
Cannot use pip to install opencv-python for python 2.7 due to error of numpy package

If need newer version of opencv in python 2.7 need to build from source process

```
student@student-VirtualBox:~/class/lecture3$ python main_img.py --file GitLogo.png
```

```
<Image File Location>
GitLogo.png
File loaded
-----------------------------------------------------
Image Information
Dimension w x h = 800 x 665
Channels = 3
Total Pixels of input image = 532000 pixels which 3 channels
-----------------------------------------------------
File Saved
```





```python
import cv2
```

Import opencv lib

```python
import argparse


#Usage python main_img.py --file Scenic009smaller.bmp #

img_location = 'DonKi.jpg'
parser = argparse.ArgumentParser(description='input image name')
parser.add_argument('--file', help="location of the image file")
```

Pass file name

```python
args = parser.parse_args()
if args.file:
        img_location = args.file
print('<Image File Location>')
print(img_location)

img = cv2.imread(img_location)
```

Import image file (GRB) to image (numpy) array

```python
print('File loaded')

h,w,c = img.shape
```

Get image information (height, Width, Channel)

```python
print('-----------------------------------------------------')
print("Image Information")
print("Dimension w x h = %s x %s" % (w,h))
print("Channels = %s" %(c))
print("Total Pixels of input image = %s pixels which %s channels" % (w*h,c))
print('-----------------------------------------------------')


gray_img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
```

Change BGR image to Gray Scale image

```python
Invert_img = 255-gray_img
```

Invert Gray Scale image

```python
cv2.imshow('Input Image',img)
cv2.imshow('Gray Image',gray_img)
cv2.imshow('Invert Image',Invert_img)
```

Display image

```python
cv2.imwrite('Invert_img.bmp',Invert_img)
```

Save image array to file

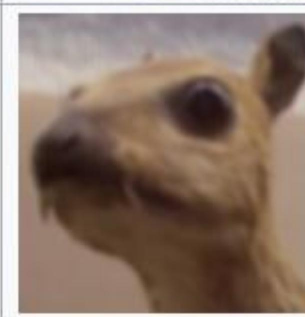
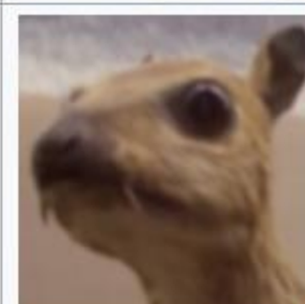
```python
print('File Saved')

cv2.waitKey(0)
```

Wait for user press any key

Image Convolution

$$g(x,y) = \omega * f(x,y) = \sum_{s=-a}^{a} \sum_{t=-b}^{b} \omega(s,t) f(x-s, y-t),$$

where $g(x,y)$ is the filtered image, $f(x,y)$is the original image, ω is the filter kernel.

Sharpen	$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$	
Box blur (normalized)	$\frac{1}{9}\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$ Low pass	
Gaussian blur 3 × 3 (approximation)	$\frac{1}{16}\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$	

Filter (Convolution)

7	23	50	64	14
15	13	31	46	8
42	25	92	31	32
71	44	74	94	92
2	43	51	35	4

Input Image

X

Filter (Kernel)

0	2	0
0	0	0
0	0	0

=

Output Image

-	-	-	-	-
-	46	100	128	-
-	26	62	92	-
-	50	184	62	-
-	-	-	-	-

```python
def convolution2d(image, filter, bias):
    m, n = filter.shape
    if (m == n):
        # Middle of the kernel
        offset = m // 2
        print("offset = %s" %offset)
        print("Applying Filter to Image")
        h, w = image.shape
        nh = h - m + offset
        nw = w - m + offset
        new_image = np.empty((nh,nw), dtype=np.uint8)
        new_image.fill(0)
        #new_image = np.zeros((y,x))
        for j in range(offset, w - offset):
            for i in range(offset, h - offset):
                acc = 0.0
                temp = 0.0
                #print("pixel-location = {%s,%s}" %(i,j))
                for a in range(0,m):
                    for b in range(0,n):
                        pixel_value = float(image[i-offset+a][j-offset+b])
                        filter_value = float(filter[a][b])
                        acc = acc + pixel_value * filter_value

                output_pixel_value = int(acc + bias)
                #print(output_pixel_value)
                if (output_pixel_value) > 255:
                    output_pixel_value =255
                if (output_pixel_value) <0:
                    output_pixel_value =0
                new_image[i-offset][j-offset] = output_pixel_value
    return new_image
```

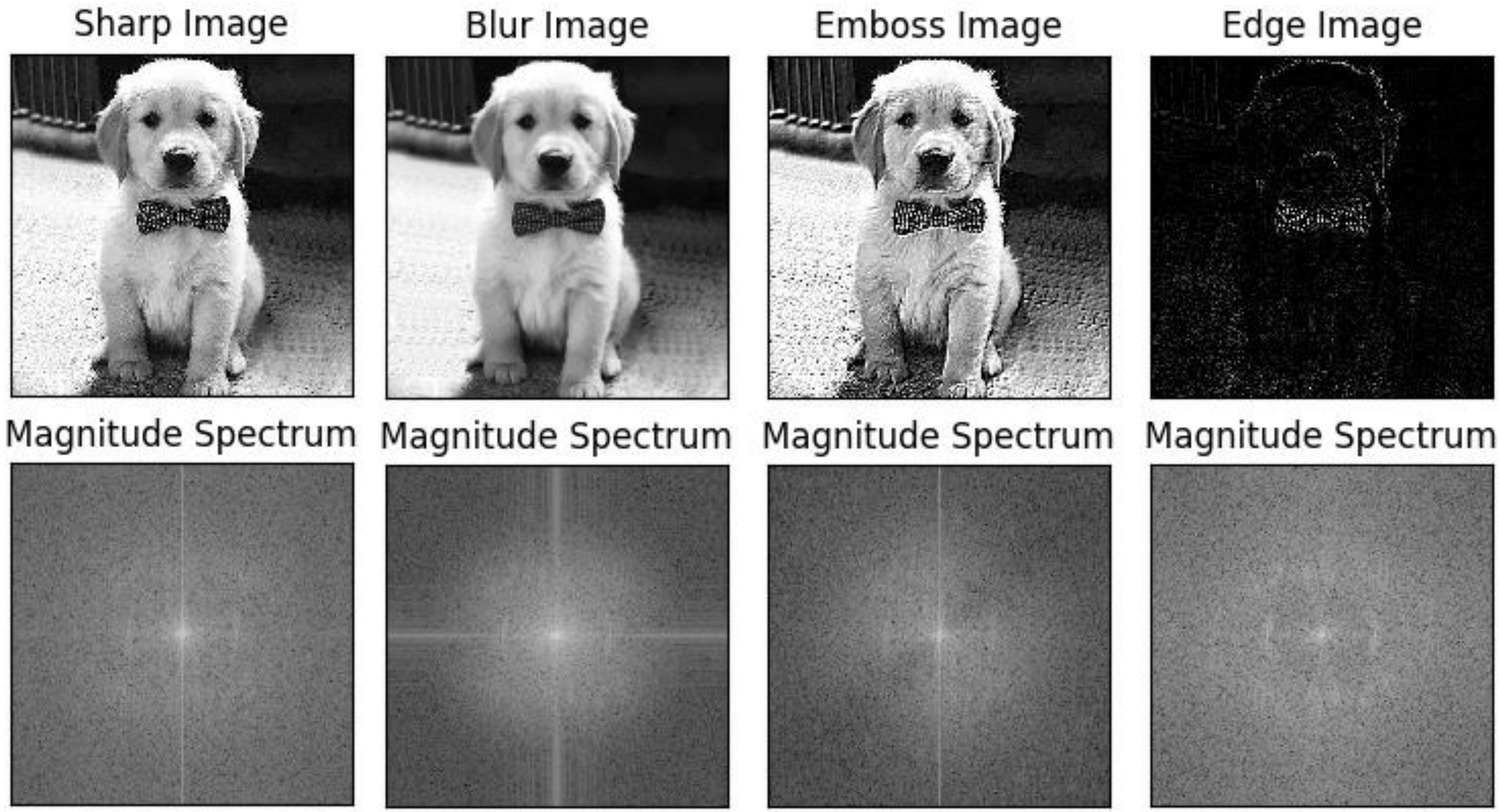
Filter size /2

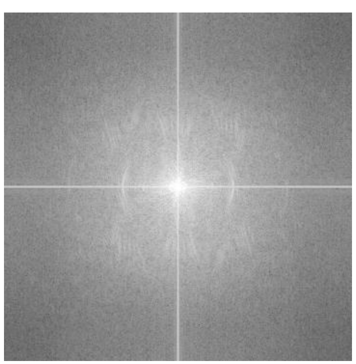
7	23	50	64	14
15	13	31	46	8
42	25	92	31	32
71	44	74	94	92
2	43	51	35	4

X

0	2	0
0	0	0
0	0	0

$$\frac{1}{16}\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix} \quad \begin{bmatrix} 0 & -0.5 & 0 \\ -0.5 & 3 & -0.5 \\ 0 & -0.5 & 0 \end{bmatrix} \quad \begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix} \quad \begin{bmatrix} -2 & -1 & 0 \\ -1 & 0 & 1 \\ 0 & 1 & 2 \end{bmatrix}$$





Original Image Spectrum

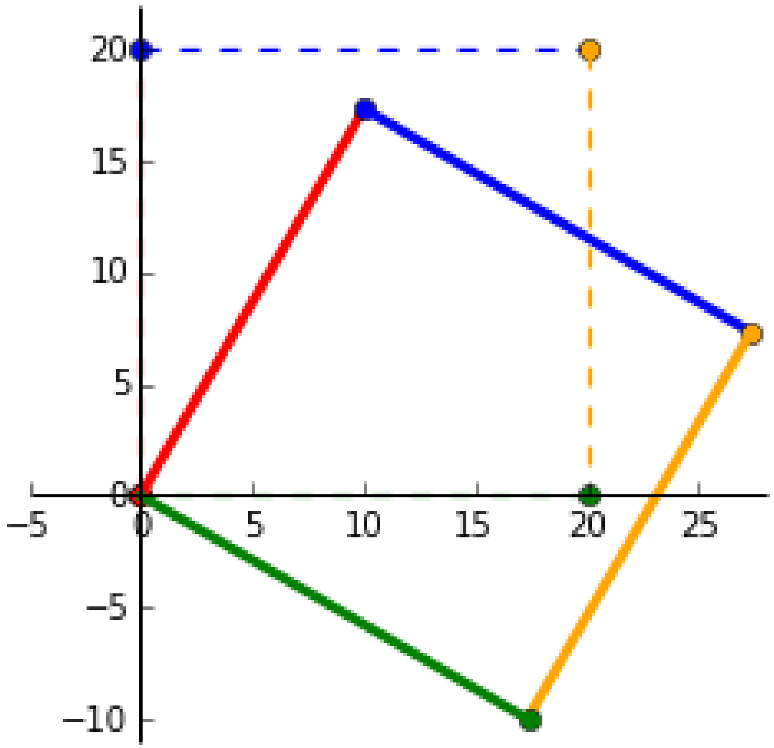
```
student@student-VirtualBox:~/class/lecture3$ python main_img2.py
<Image File Location>
Donki.jpg
File loaded
----------------------------------------------------
Image Information
Dimension w x h = 421 x 404
Total Pixels of input image = 170084 pixels
----------------------------------------------------
---------------------------------------------------------
Applying Blur Filter
offset = 1
Applying Filter to Image
---------------------------------------------------------
Applying High Pass Filter
offset = 1
Applying Filter to Image
---------------------------------------------------------
Applying Edge Filter
offset = 1
Applying Filter to Image
---------------------------------------------------------
Applying Emboss Filter
offset = 1
Applying Filter to Image
---------------------------------------------------------
```

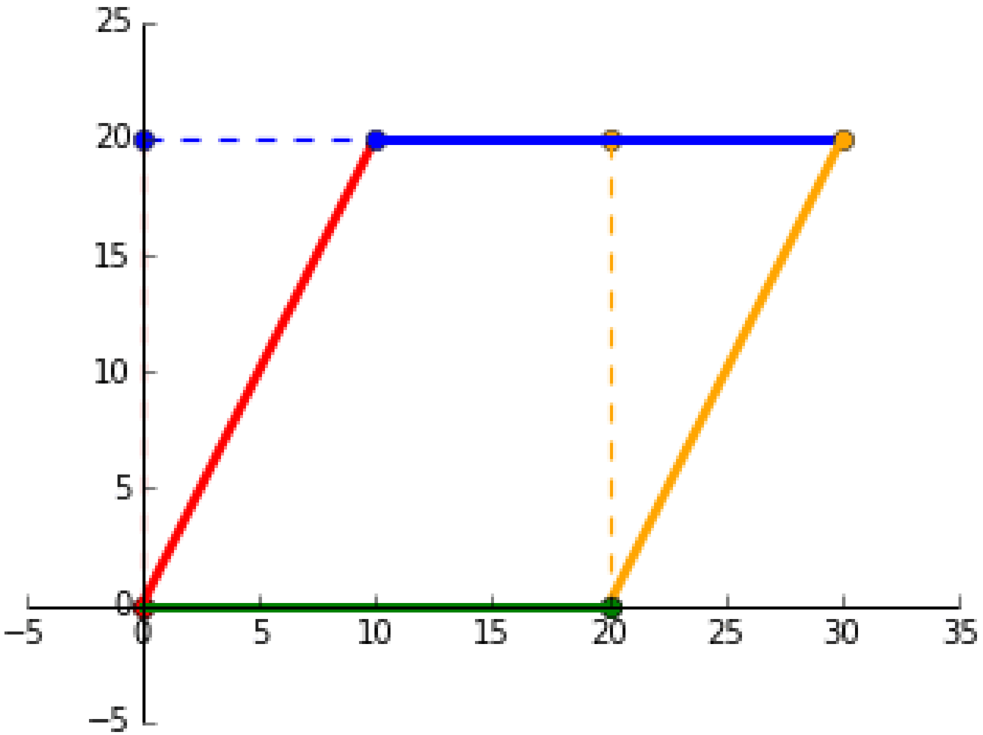
```python
# Box Blur Filter AKA Low-pass Filter
box_blur_filter = np.array([[1/9.0, 1/9.0, 1/9.0], [1/9.0, 1/9.0, 1/9.0], [1/9.0, 1/9.0, 1/9.0]])

# High-pass Filter
sharpen_filter = np.array([[  0.0   , -0.5 ,    0.0 ], [-0.5 ,   3.0   , -0.5 ],[  0.0   , -0.5 ,    0.0 ]])

# Edge Filters
Edge_filter = np.array([[  -1.0   , -1.0 ,    -1.0 ], [-1.0 ,   8.0   , -1.0 ],[  -1.0   , -1.0 ,    -1.0 ]])

# Emboss Filter
Emboss_filter = np.array([[  -2.0   , -1.0 ,    0.0 ], [-1.0 ,   1   , 1.0 ],[  0.0   , 1.0 ,    2.0 ]])
```


Input Image
ドン.キホーテ
Gray Image
ドン.キホーテ
Blur Image
Emboss Image
ドン.キホーテ
Edge Image
ドン.キホーテ
Sharpen Image

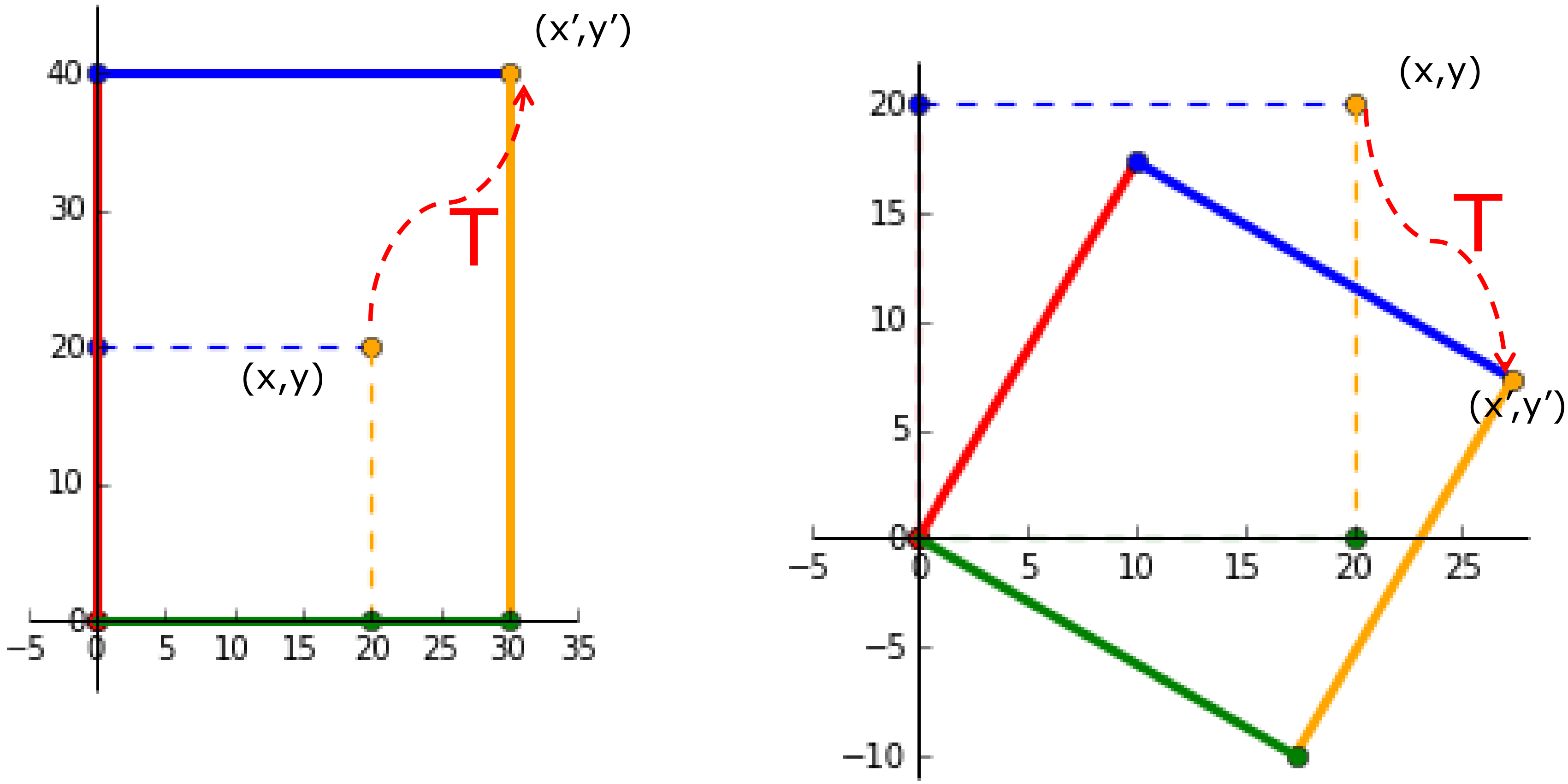
Affine Transformation

Affine transformation is a functional mapping between two geometric (affine) spaces which preserve points, straight and parallel lines as well as ratios between points.



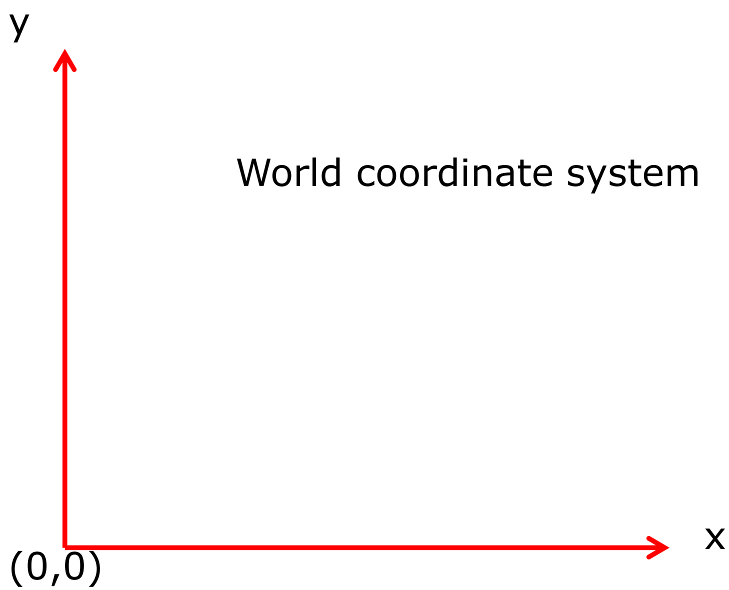


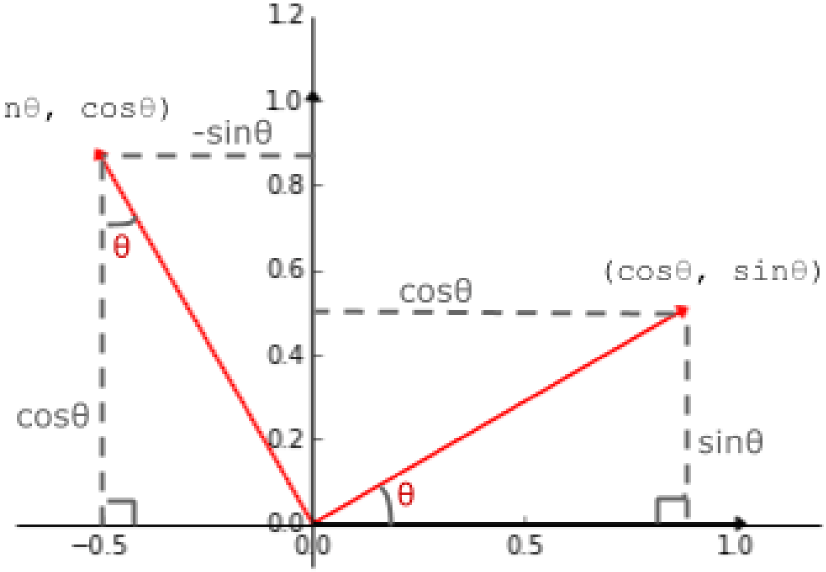
New-location (x’,y’) = Affine Transformation(T) x Old (x,y)



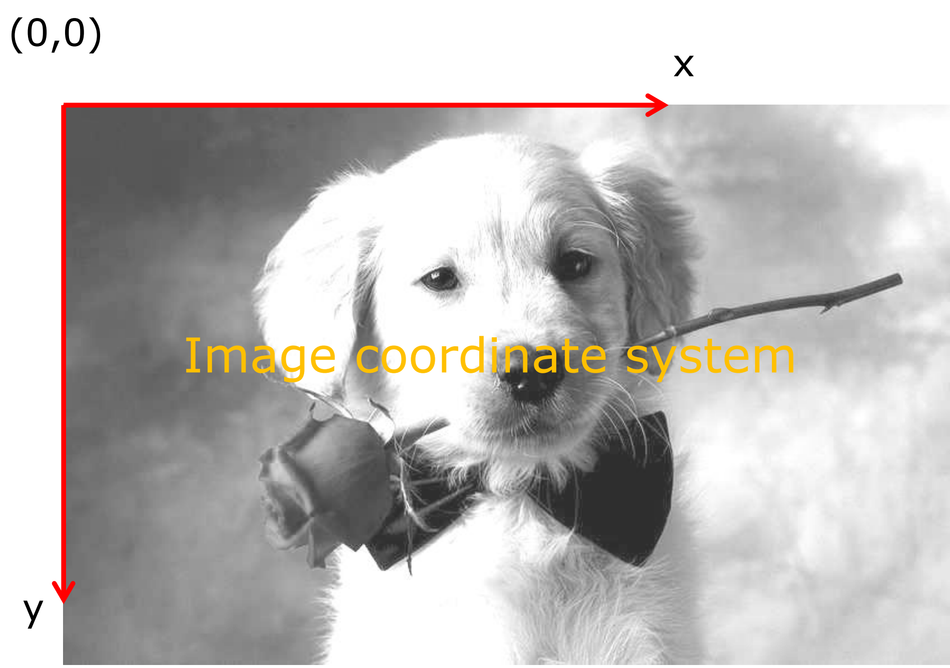
All New location (pixels) = T x All Old location (pixels)

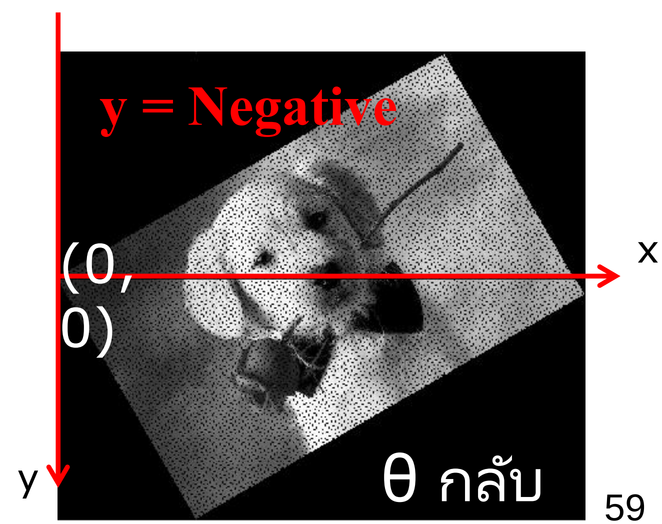
Identity	T=	$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$	$x^{'} = x$ $y^{'} = y$
Scaling	T=	$\begin{bmatrix} c_x & 0 & 0 \\ 0 & c_y & 0 \\ 0 & 0 & 1 \end{bmatrix}$	$x^{'} = c_x * x$ $y^{'} = c_y * y$
Rotation*	T=	$\begin{bmatrix} cos\Theta & sin\Theta & 0 \\ -sin\Theta & cos\Theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$	$x^{'} = x * cos\Theta - y * sin\Theta$ $y^{'} = x * cos\Theta + y * sin\Theta$
Translation	T=	$\begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix}$	$x^{'} = x + t_x$ $y^{'} = y + t_y$

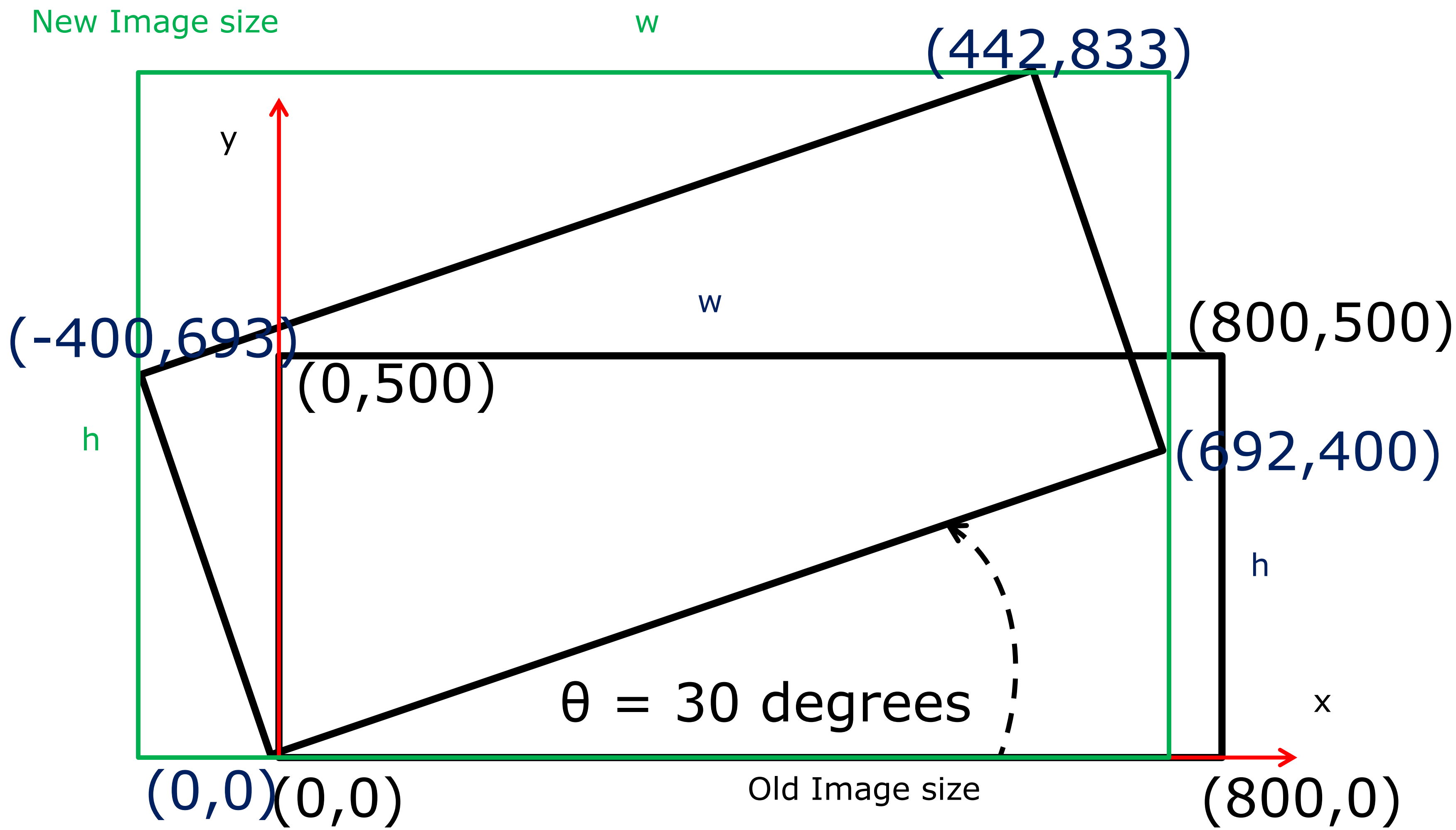




$$\begin{bmatrix} cos\Theta & sin\Theta & 0 \\ -sin\Theta & cos\Theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$





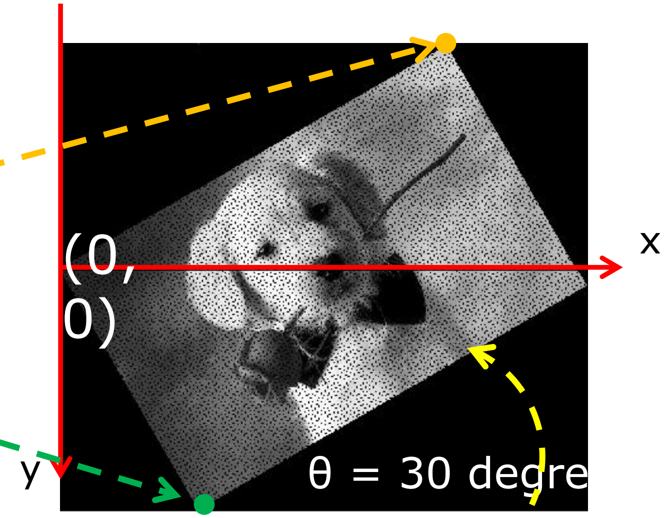


New Image size (w,h) = (max(x) – min(x) , max(y)-min(y))

cx = min(x)

cy = min(y)

```
Transformation Matrix
[[ 0.8660254 -0.5        0.        ]
 [ 0.5        0.8660254  0.        ]
 [ 0.         0.         1.        ]]
Input Image Shape
(500L, 800L)
location of input image corners
[[  0 500 500   0]
 [  0   0 800 800]
 [  1   1   1   1]]
location of transformated image corners
[[   0.         433.01270189   33.01270189 -400.        ]
 [   0.         250.          942.82032303  692.82032303]
 [   1.           1.            1.            1.        ]]
-400
0
Ouput Image Shape
(835L, 944L)
```



Coordinate is different

```
#calculate possible size of output image after transformation using image corners

#coor = np.array([[0, w, w, 0], [0, 0, h, h],[1 , 1 ,1 ,1]])
coor = np.array([[0, h, h, 0], [0, 0, w, w],[1 , 1 ,1 ,1]])
```

old corners

```
print('location of input image corners')
print(coor)
new_coor  = np.dot(T,coor)
```

new corners = T x old corners

```
print('location of transformated image corners')
print(new_coor)
newHeight = int(np.ceil(new_coor[0].max() - new_coor[0].min()))
newWidth = int(np.ceil(new_coor[1].max() - new_coor[1].min()))

#Calculation of correction of output coordinate system
cx, cy = int(np.floor(new_coor[0].min()))  , int(np.floor(new_coor[1].min()))
```

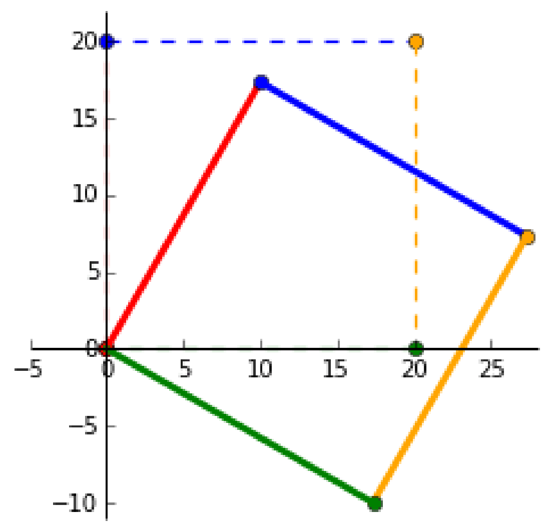
```python
def Get_Rotation_Matrix(theta):
    theta_rad =  -theta * np.pi / 180
    s, c = np.sin(theta_rad), np.cos(theta_rad)
    RMatrix= np.array([[c, s, 0], [-s, c, 0], [0, 0, 1]])
    return RMatrix
```

$$T = \begin{bmatrix} cos\Theta & sin\Theta & 0 \\ -sin\Theta & cos\Theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

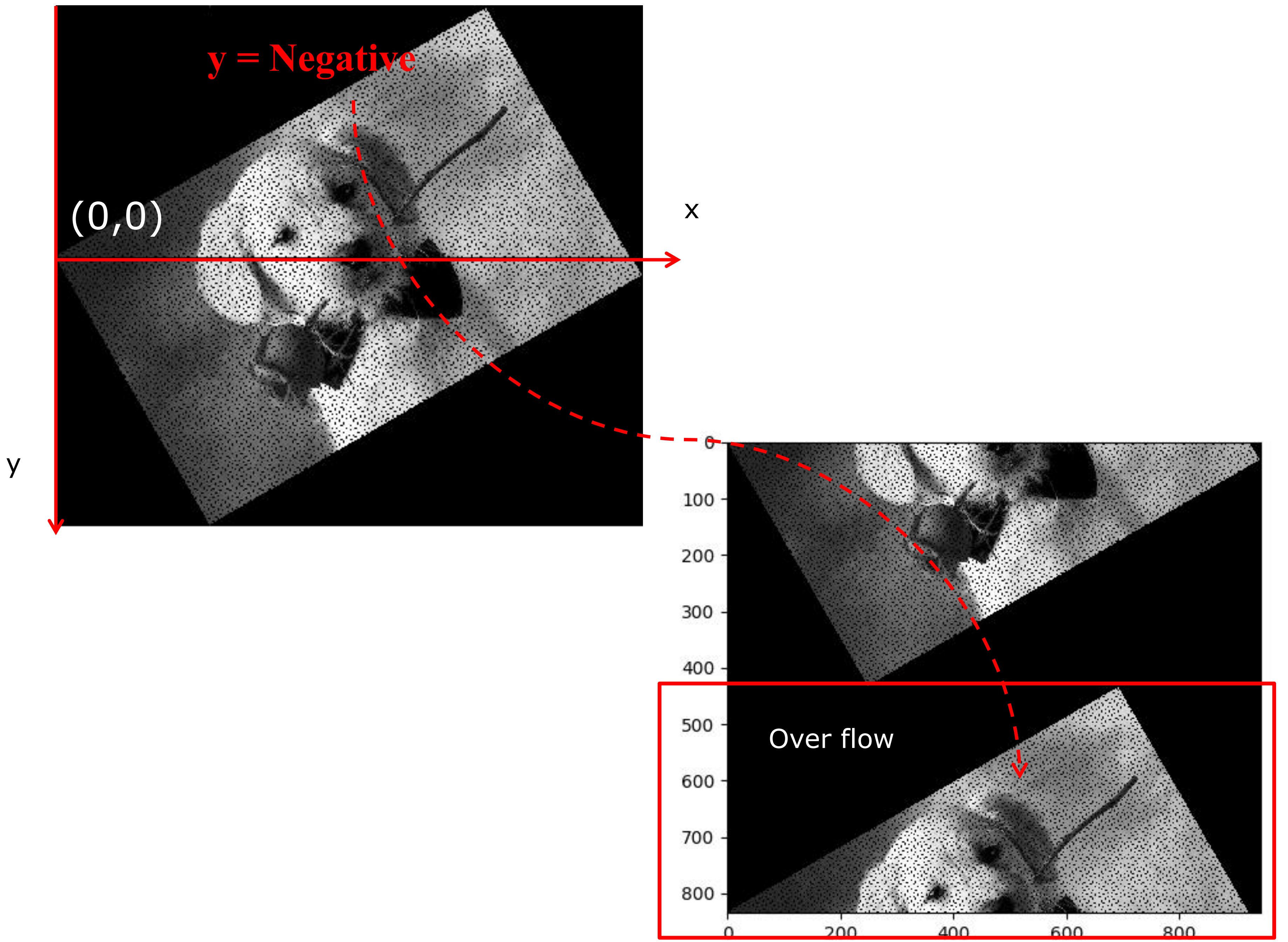
```python
for j in range(0, w):
    for i in range(0, h):
        pixel_data = img[i][j]
        input_coords = np.array([i, j, 1])
        i_out, j_out, _ = np.dot(T,input_coords)

        if int(j_out) >= newWidth:
            #Boundary
            j_out = newWidth
        if int(i_out) >= newHeight:
            #Boundary
            i_out = newHeight
        img_transformed[int(np.floor(i_out)), int(np.floor(j_out))] = pixel_data
```

(x',y')

(x,y)



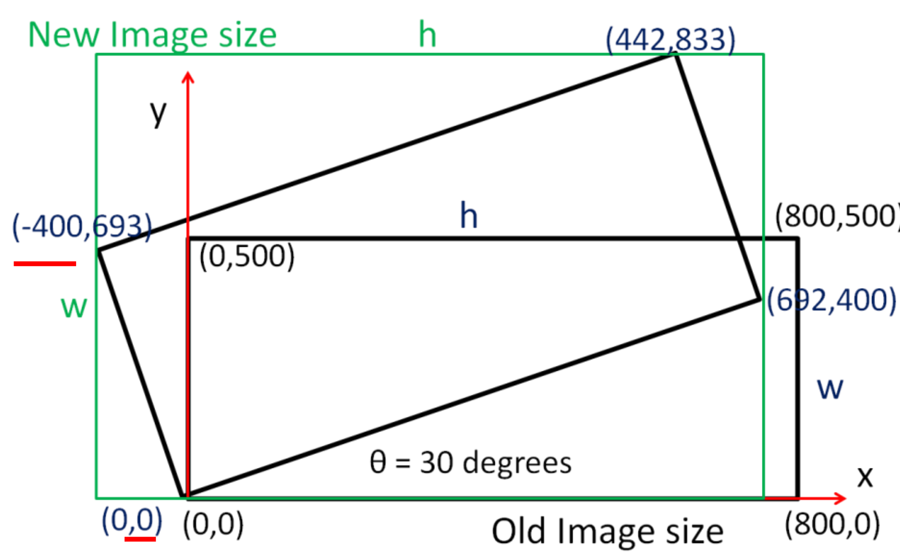
np.dot(A,B) = AxB =Matrix Multiplication

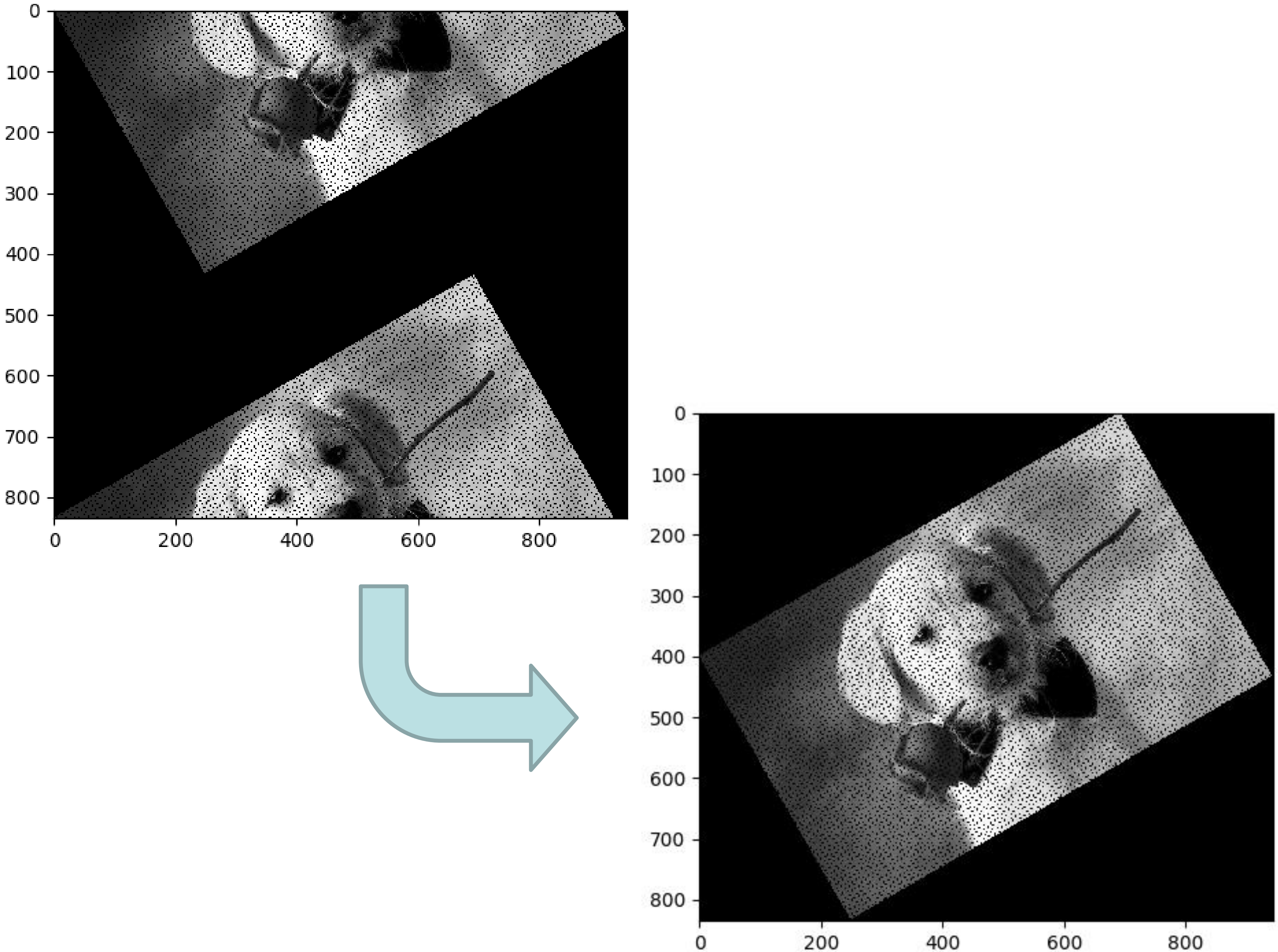


y = Negative
(0,0)
x
y
Over flow
0
100
200
300
400
500
600
700
800
0
200
400
600
800

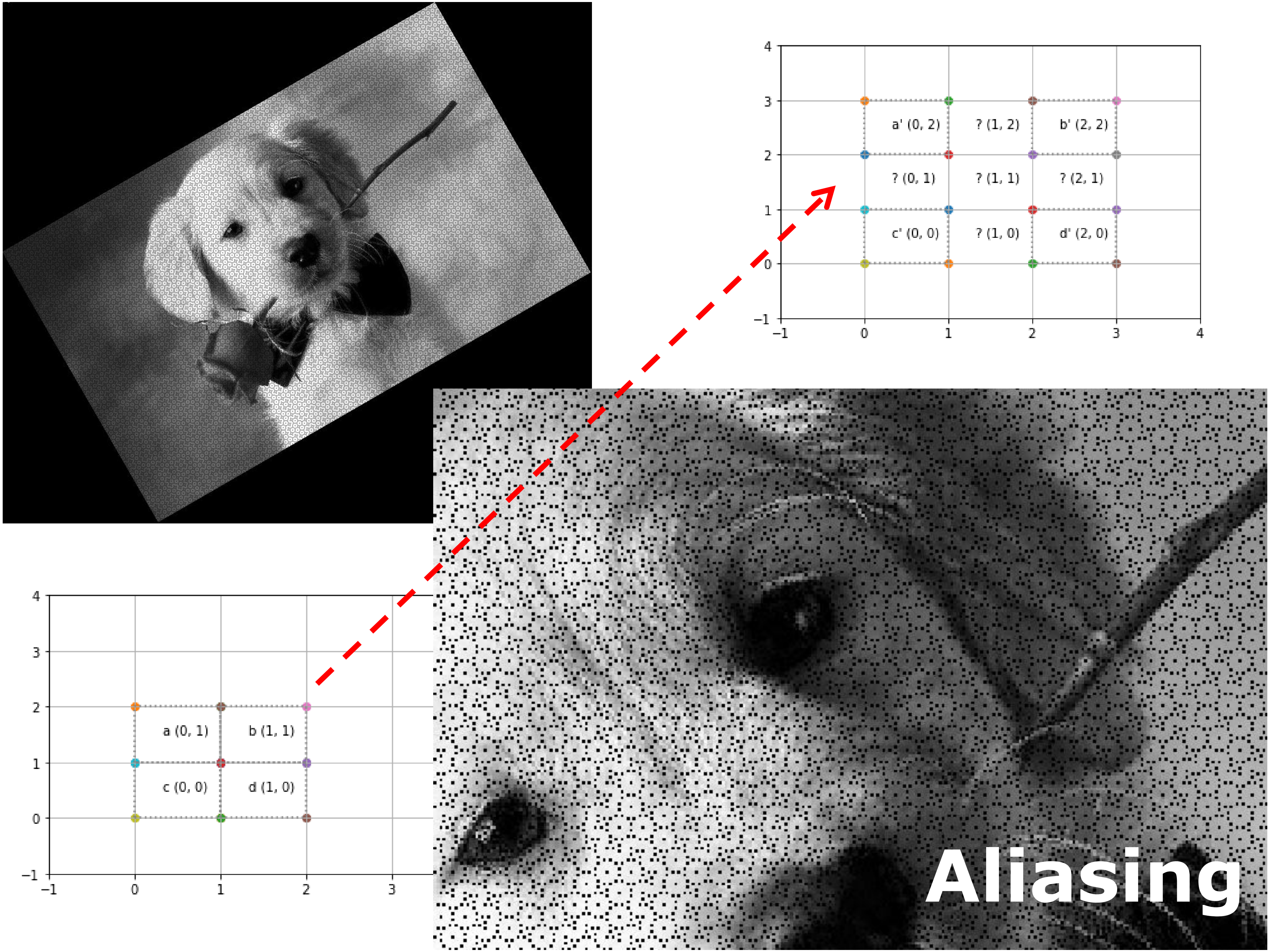
```
for j in range(0, w):
    for i in range(0, h):
        pixel_data = img[i][j]
        input_coords = np.array([i, j, 1])
        i_out, j_out, _ = np.dot(T,input_coords)

        if int(j_out) >= newWidth:
            #Boundary
            j_out = newWidth
        if int(i_out) >= newHeight:
            #Boundary
            i_out = newHeight
        img_transformed[int(np.floor(i_out)), int(np.floor(j_out))] = pixel_data

        i_out = np.floor(i_out - cx)
        j_out = np.floor(j_out - cy)
        img_transformed_corrected[int(i_out), int(j_out)] = pixel_data
```

```
cx, cy = int(np.floor(new_coor[0].min()))  , int(np.floor(new_coor[1].min()))
```

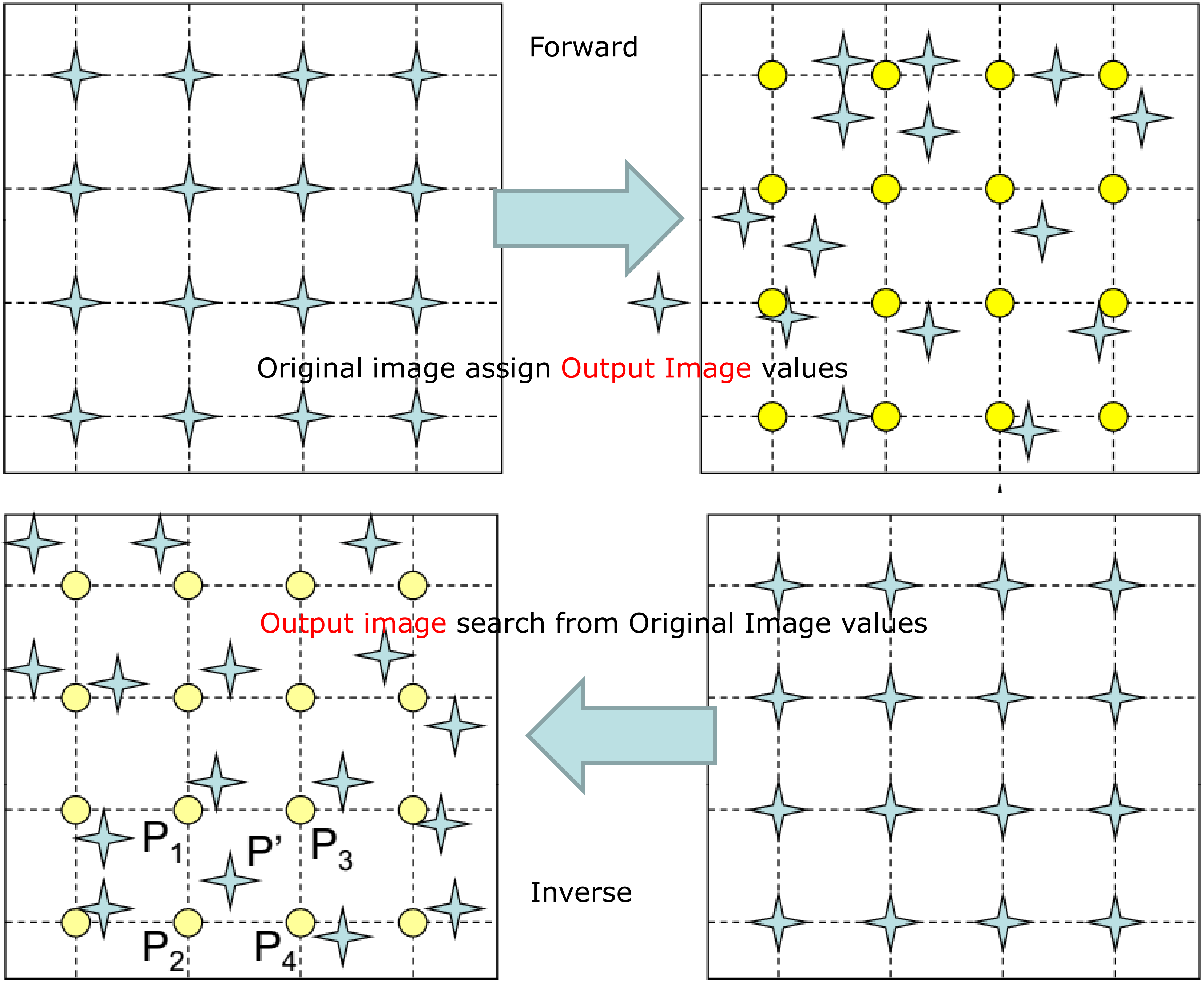




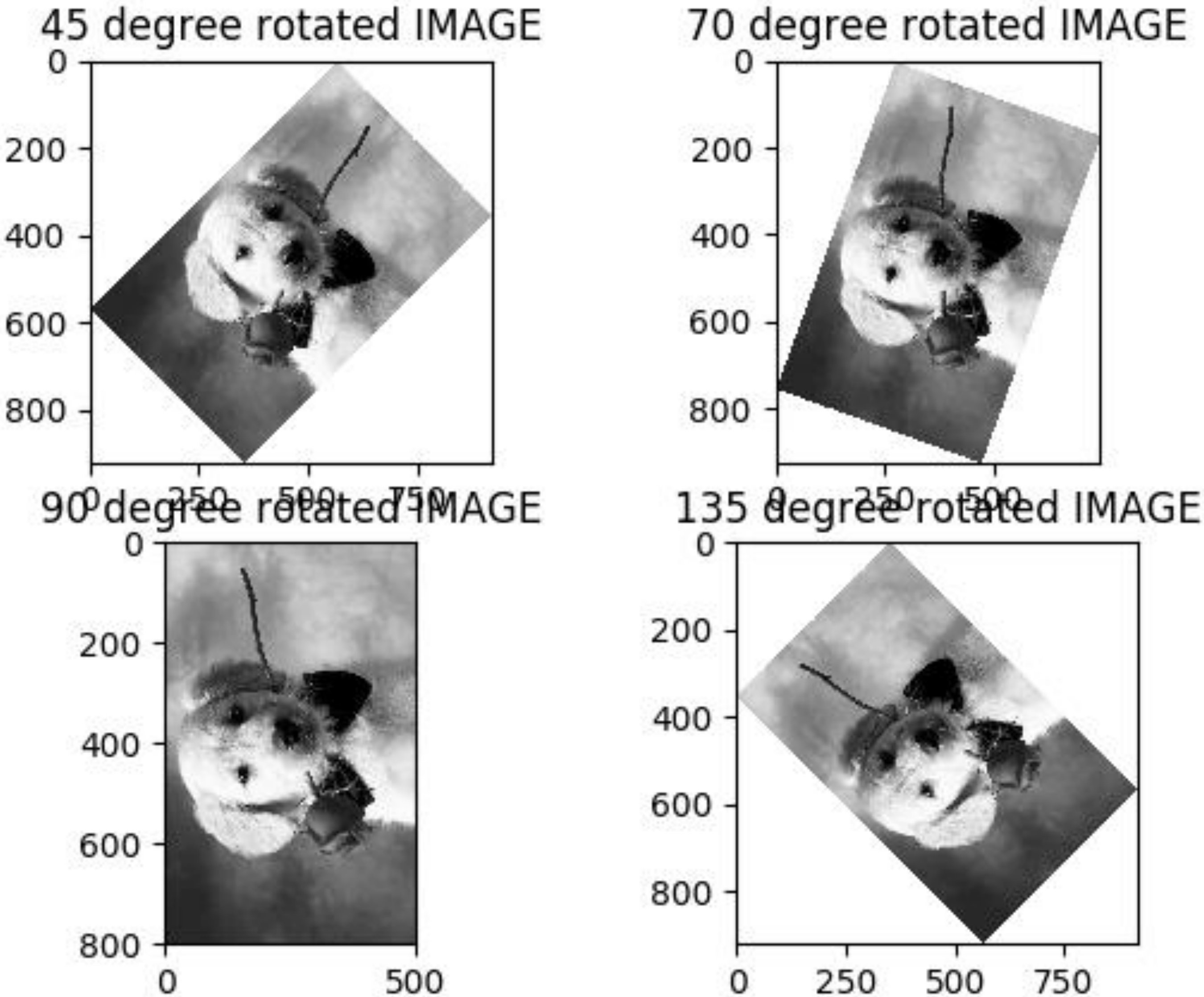
0
100
200
300
400
500
600
700
800
0
200
400
600
800
0
100
200
300
400
500
600
700
800
0
200
400
600
800



a' (0, 2)
? (1, 2)
b' (2, 2)
? (0, 1)
? (1, 1)
? (2, 1)
c' (0, 0)
? (1, 0)
d' (2, 0)
a (0, 1)
b (1, 1)
c (0, 0)
d (1, 0)
Aliasing

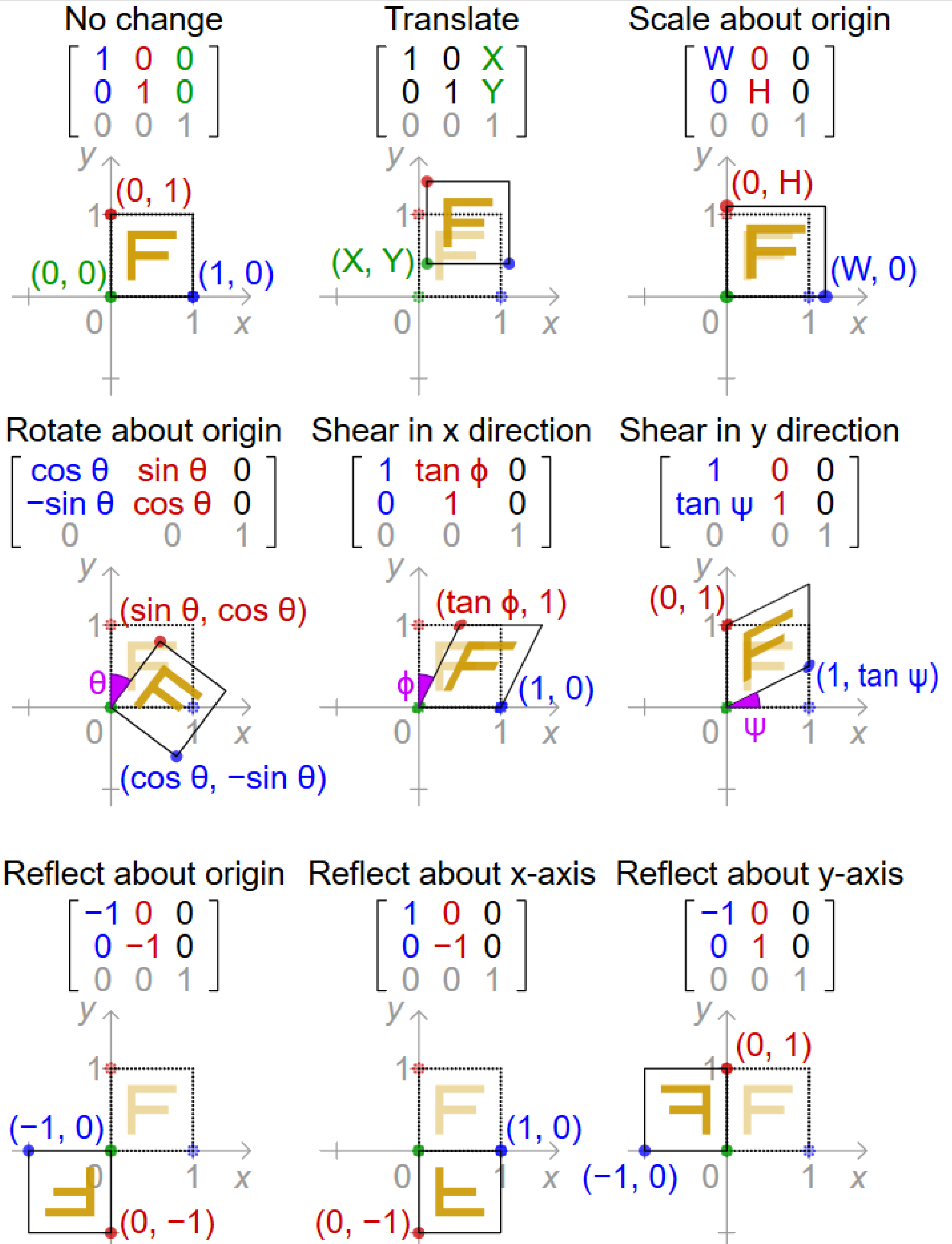

Forward
Original image assign Output Image values
Output image search from Original Image values
P1
P'
P3
P2
P4
Inverse

```python
def rotate_image(src, theta, ox, oy, fill=255):
    """Rotate the image src by theta radians about (ox, oy).
    Pixels in the result that don't correspond to pixels in src are
    replaced by the value fill.
    """
    # Images have origin at the top left, so negate the angle.
    theta = -theta
    # Dimensions of source image.
    # images in row-major order, so shape gives (height, width).
    sh, sw = src.shape
    # Rotated positions of the corners of the source image.
    cx, cy = rotate_coords([0, sw, sw, 0], [0, 0, sh, sh], theta, ox, oy)
    # Determine dimensions of destination image.
    dw, dh = (int(np.ceil(c.max() - c.min())) for c in (cx, cy))
    print('Dimesion of output image')
    print(dw)
    print(dh)
    # Coordinates of pixels in destination image.
    dx, dy = np.meshgrid(np.arange(dw), np.arange(dh))
    # Corresponding coordinates in source image. Since we are
    # transforming dest-to-src here, the rotation is negated.
    sx, sy = rotate_coords(dx + cx.min(), dy + cy.min(), -theta, ox, oy)
    # Select nearest neighbour.
    sx, sy = sx.round().astype(int), sy.round().astype(int)
    # Mask for valid coordinates.
    mask = (0 <= sx) & (sx < sw) & (0 <= sy) & (sy < sh)
    # Create destination image.
    dest = np.empty(shape=(dh, dw), dtype=src.dtype)
    # Copy valid coordinates from source image.
    dest[dy[mask], dx[mask]] = src[sy[mask], sx[mask]]
    # Fill invalid coordinates.
    dest[dy[~mask], dx[~mask]] = fill
    return dest
```



```
img_45 = rotate_image(golden_img, 45 * np.pi / 180, 0,0)
img_70 = rotate_image(golden_img, 70 * np.pi / 180, 0,0)
```

Affine Transform



Exercise 3 (image)

1. Generate a “negative” greyscale image
   - A “negative” image is an image where white and black are reversed
2. Flip the image upside down
3. Rotate the image 90 degrees
4. Shrink the image width to one half
5. Enlarge the image width by 1.5 times
6. In the color quantization example, each colormap value was simply calculated as the average of min/max range values. Consider improving the quantized image quality by setting the colormap value as the average RGB values of all pixels that map to that colormap
7. We can improve the quantized image quality further by adaptively setting each quantization steps and levels. Most popular method is called “median-cut” algorithm. Try programming this method. To learn how the median-cut algorithm work, refer to http://micro.magnet.fsu.edu/primer/java/digitalimaging/processing/colorreduction/index.html