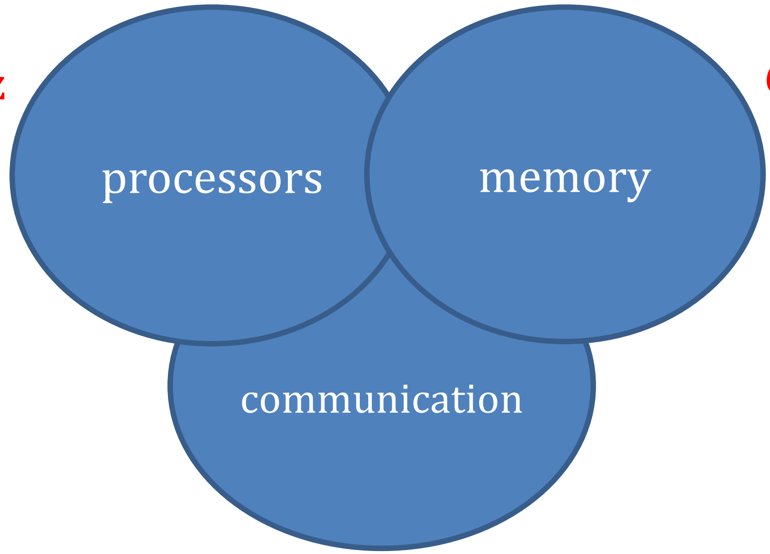
Lecture : Representing images

Lecture 21-22 Plan

- Introduction to Digital Image representation.
  - Grayscale and color image
  - Bit depth, resolution
  - Class Matrix
- Generating synthetic images
- Basics of Digital Image Processing
- Noise, and local noise reductions

Brief "Historical" Technological Context

transistors	speed	RAM	Hard Disk
29 K	4.77 MHz	640 KB	5 MB
1.4 G	3.7 GHz	16 GB	2 TB



e-mail, simple text (128 ascii chars)
tons of data, inc. images (next slide)

- Early 1980's
- Today (2013)

A Brief Historical Context, 30 Years Later

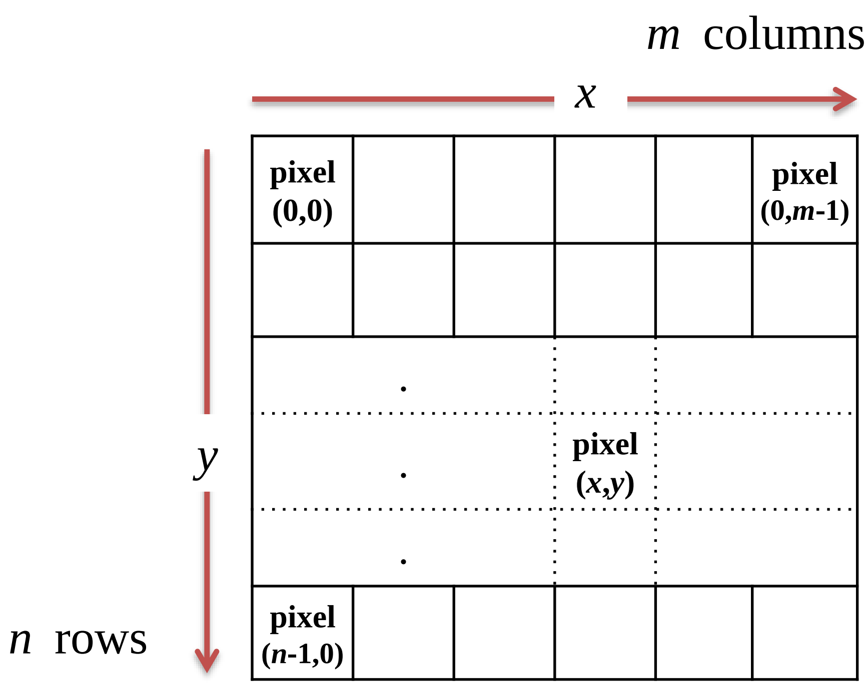
- With the proliferation of
  (1) larger and faster memory,
  (2) strong, inexpensive processors,
  (3) faster internet,

  it became possible to efficiently
  (1) store,
  (2) process, and
  (3) transmit large digital images.

- Facebook: ~350 million photos DAILY (Sep. 2013).
- Instagram: ~ 16 billion. ~55 million photos daily (Dec. 2013). (Instagram was launched on Oct. 2010!!).
- This dramatic technological progress is reflected by the following saying, often attributed (apparently incorrectly) to Bill Gates, in 1981: "640KB ought to be enough for anybody".

Basic Model of a Digital Image

- A digital image is typically encoded as a n-by-m rectangle, or matrix, M, of either grey-level or color values.

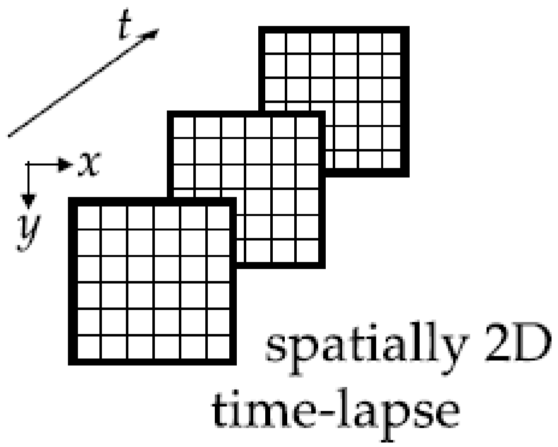


***n* x *m* matrix**

Video

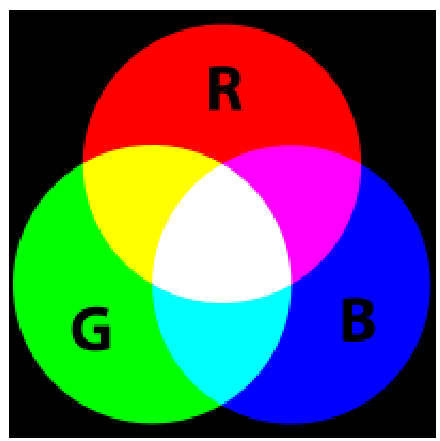
- A 2D image is encoded as a n-by-m matrix M

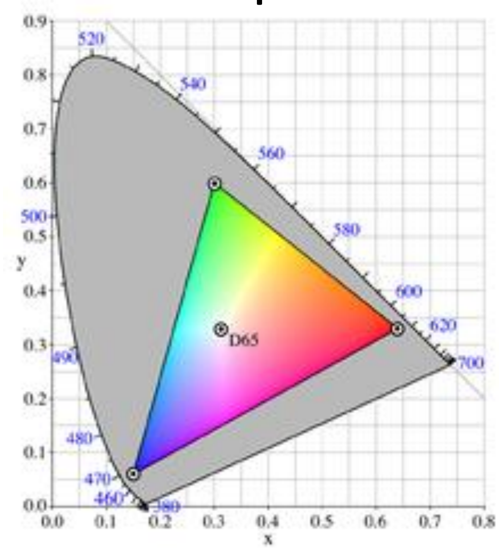
- For videos (movies), there is a third dimension, "time". For each point *t* sampled in time, the frame at time *t* is nothing but a "regular" image.



Color images (RGB)

- Each element M[x, y] of the image is called a pixel, shorthand for picture element.
- For grey level images, M[x, y] is a non negative real number, representing the light intensity at the pixel.
- For standard (RGB) color images, M[x, y] is a triplet of values, representing the red, green, and blue components of the light intensity at the pixel.





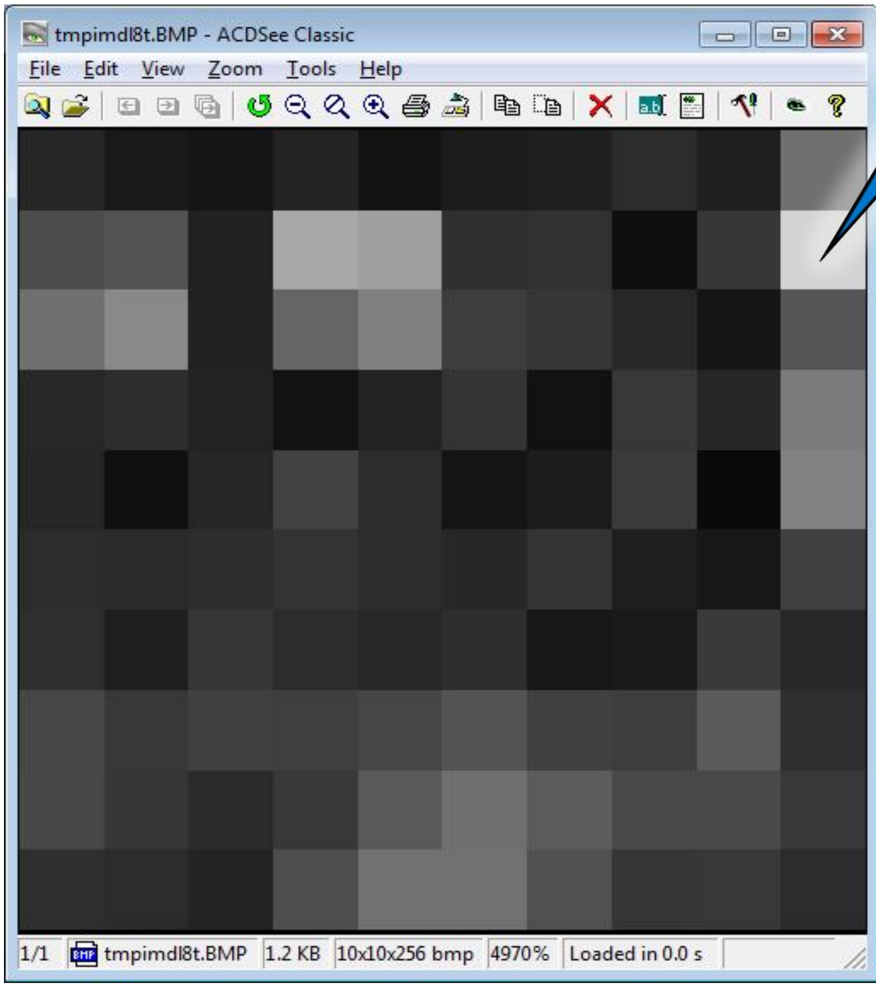
(images from Wikipedia)

Gray scale images

- We discuss grey scale images only (for simplicity).
- Real numbers expressing grey levels have to be discretized.
- A good quality photograph (human visual inspection) has 256 grey-level values (8 bits) per pixel.
- The value 0 represents black, 255 represents white.
- In some applications (e.g., medical imaging) 4096 grey levels (12 bits) are used.

Gray scale image

- 8 bits per pixel (2^8=256 gray levels): 0 = black, 255 = white



whiteish

value

38,	26,	21,	36,	19,	28,	33,	44,	31,	112,
77,	83,	34,	168,	159,	48,	50,	14,	55,	211,
112,	137,	34,	101,	129,	62,	54,	40,	21,	86,
41,	46,	35,	19,	35,	52,	18,	57,	39,	123,
38,	16,	38,	67,	45,	21,	29,	59,	10,	130,
45,	43,	46,	51,	44,	39,	53,	31,	24,	64,
47,	30,	54,	45,	40,	46,	23,	26,	58,	40,
71,	57,	66,	63,	70,	84,	65,	62,	91,	49,
72,	55,	43,	57,	90,	111,	92,	73,	74,	56,
47,	45,	36,	78,	114,	113,	81,	54,	57,	44

Bit Depth

- Number of bits per pixel.

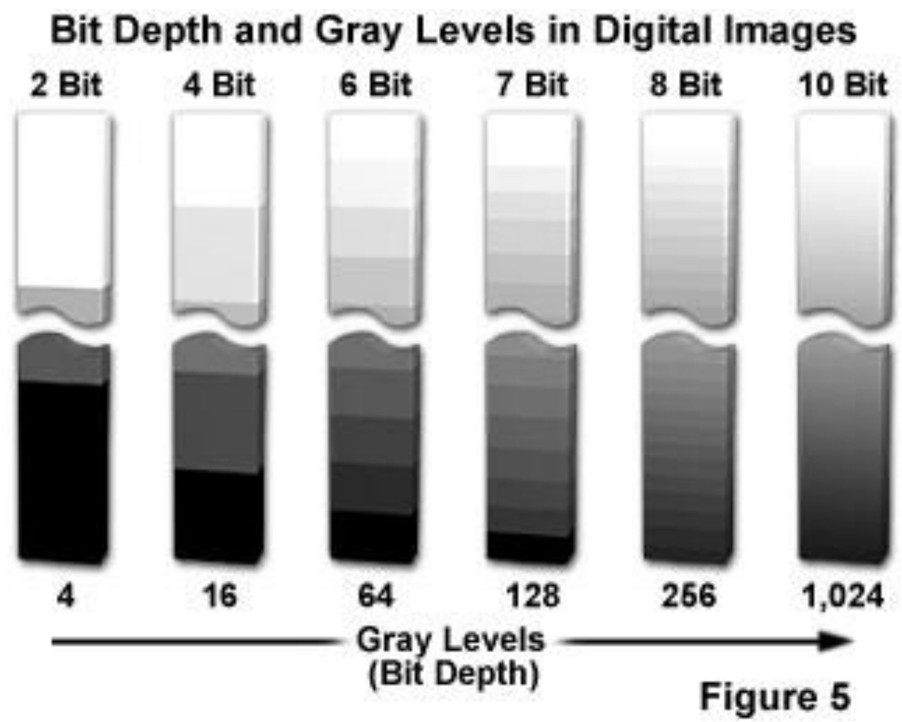


Image from:
http://micro.magnet.fsu.edu/

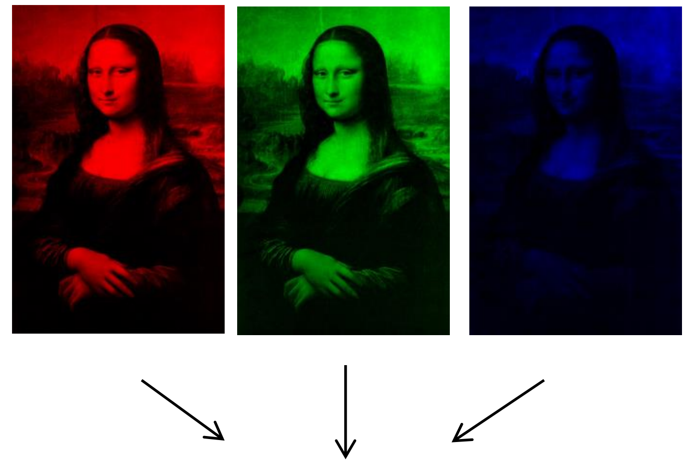
- A human observer sees at most a few hundreds shades of gray
- Higher bit depths images: typically for automated analysis by a computer.

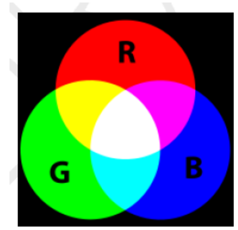
BW / Grayscale / RGB - summary

- Black & white / gray-level / RGB

Black & white
(1 bpp)

256 gray level image
(8 bpp)





"true color" image
(8+8+8 = 24 bpp)

Images from: http://www.csse.uwa.edu.au/~wongt/matlab.html

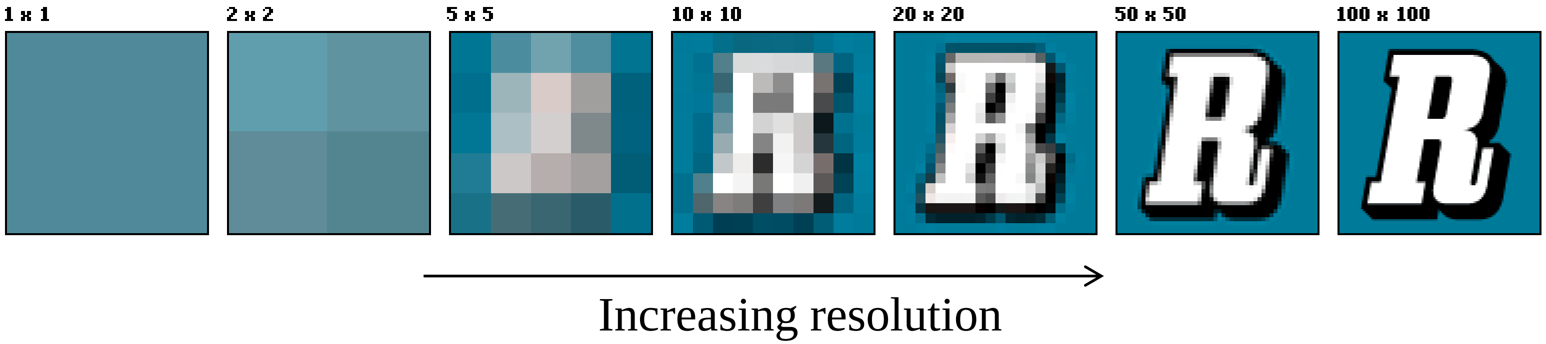
Color Bit Depth and Resolution

Resolution and Pixel Physical Size

- Resolution is the capability of the sensor to observe or measure the smallest object clearly with distinct boundaries.

- Resolution depends on the physical size of a pixel.

  Higher resolution = more pixels per area = **lower** pixel size.



Source: wikipedia

Our implementation: The Class Matrix

- Class Matrix, implemented as a list of lists:

```
class Matrix:
    def __init__(self, n, m, val=0):
        assert n > 0 and m > 0
        self.rows = [[val]*m for i in range(n)]
```

fill
with val

The Class Matrix (2)

```
class Matrix:
    def __init__(self, n, m, val=0):
        assert n > 0 and m > 0
        self.rows = [[val]*m for i in range(n)]
    def dim(self):
        return len(self.rows), len(self.rows[0])

    def __repr__(self):
        if len(self.rows)>10 or len(self.rows[0])>10:
            return "Matrix too large, specify submatrix"
        return "<Matrix {}>".format(self.rows)

    def __eq__(self, other):
        return isinstance(other, Matrix) and \
               self.rows == other.rows
```

calls eq
of class list

The Class Matrix (3)

- Additional methods (we will only show how to use them):
  - ❑ **copy**
  - ❑ **Arithmetical** operations, e.g., `mat1 + mat2`
  - ❑ **__getitem__** : receives a tuple (i,j)
  - ❑ **__setitem__** : receives a tuple (i,j) and val

    i and j can be both integers or both slices
  - ❑ **display**: shows the image represented by a matrix, uses the Python standard (no installation needed) package tkinter
  - ❑ **save** and **load**: enable storing and reading images from files

class Matrix - item access and assignment

```
>>> m = Matrix(10, 10)   # 10x10 matrix of zeros
>>> m[4,5]               # same as m.__getitem__((4,5))
0
>>> m[4,5] = 45          # same as m.__setitem__((4,5),45)
>>> m[4,5]
45
```

Note: the code file contains an additional feature: accessing and assignment of a whole slice.

```
>>> m[3:5, 4:8]          # here i and j are both slices
<Matrix [[0 , 0, 0, 0], [0, 45, 0, 0]] >
```

class Matrix - Indexing

```
# cell/sub-matrix access/assignment
####################################
```

```
#ij is a tuple (i,j). Allows m[i,j] instead m[i][j]
def __getitem__(self, ij):
    i,j = ij
    if isinstance(i, int) and isinstance(j, int):
        return self.rows[i][j]
    elif isinstance(i, slice) and isinstance(j, slice):
        M = Matrix(1,1) # to be overwritten
        M.rows = [row[j] for row in self.rows[i]]
        return M
    else:
        return NotImplemented
```

both ints

both slices

int & slice

```python
#ij is a tuple (i,j). Allows m[i,j] instead m[i][j]
def __setitem__(self, ij, val):
    i,j = ij
    if isinstance(i,int) and isinstance(j,int):
        assert isinstance(val, (int, float, complex))
        self.rows[i][j] = val
    elif isinstance(i,slice) and isinstance(j,slice):
        assert isinstance(val, Matrix)
        n,m = val.dim()
        s_rows = self.rows[i]
        assert len(s_rows) == n and len(s_rows[0][j]) == m
        for s_row, v_row in zip(s_rows,val.rows):
            s_row[j] = v_row
    else:
        return NotImplemented
```

both ints

both slices

ZIP - example

```
numberList = [1, 2, 3]
strList = ['one', 'two', 'three']

# Two iterables are passed
result = zip(numberList, strList)

for e in result:
    print(e)
```

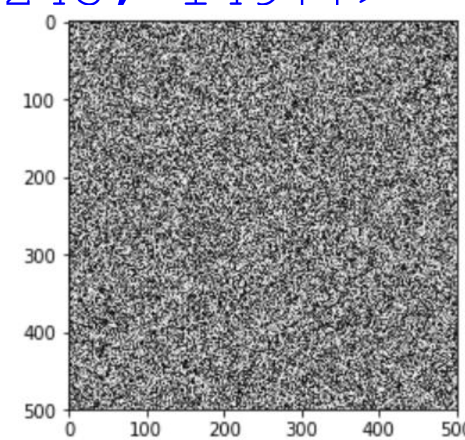
```
(1, 'one')
(2, 'two')
(3, 'three')
```

Display image

```
# display - for image visualization - using plt
###############################################
def display(self, title=None, zoom=None):
    X = np.array(self.rows)
    plt.imshow(X, cmap="gray")
    plt.show()
```

Displaying an image

```
n = 500
m = 500
mat = Matrix(n,m)

for i in range(n):
    for j in range(m):
        mat[i,j] = random.randint(0,255)

>>> mat
Matrix too large, specify submatrix

>>> mat[3:5, 4:8]
<Matrix [[216, 213, 114, 208], [2, 4, 245, 149]]>

>>> mat.display()
```



save to and load from file

```
>>> mat.save("./rand_image.bitmap")
```

A new file rand_image.bitmap will be created.
Although we gave it the extension .bitmap, this is only a text file:

rand_image.bitmap

```
500 500
230 168 178 213 28 159 121 …
222 133 165 152 8 236 188  …
51 152 152 93 120 117 208  …
…
```

```
>>> mat2 = Matrix.load("./rand_image.bitmap")
```

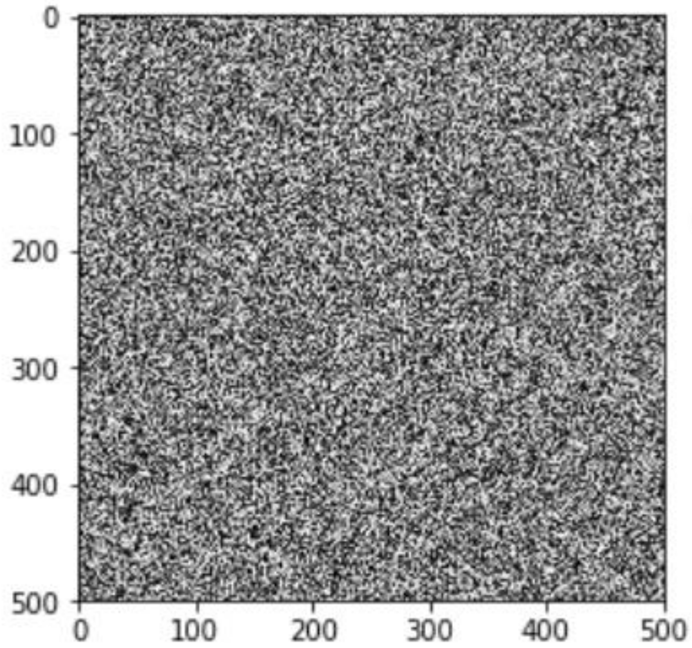
save image to file

```
def save(self, filename):
    f = open(filename, 'w')
    n,m = self.dim()
    print(n,m, file=f)
    for row in self.rows:
        for e in row:
            print(e, end=" ", file=f)
        print("",file=f) #newline
    f.close()
```

Using save to and load

```
mat.save("./rand_image.bitmap")
```

```
mat2 = Matrix.load("./rand_image.bitmap")
mat2.display()
```



same as before

from "real" image formats to .bitmap

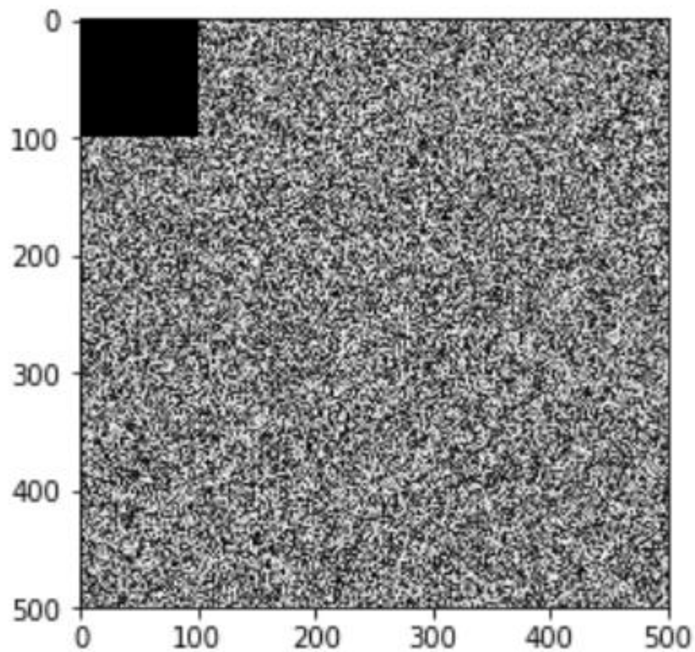
- we provide a way to work with "real" images in known formats such as jpg, bmp, tif etc.
- The file format_conversion.py contains the transformation in both directions.
- To have it work, you first need to install an external Python package called PILLOW – Python Imaging Library, from: https://pypi.python.org/pypi/Pillow

```
>>> image2bitmap("./an_image.jpg") #creates an_image.bitmap

>>> bitmap2image("./an_image.bitmap") #vice versa
```

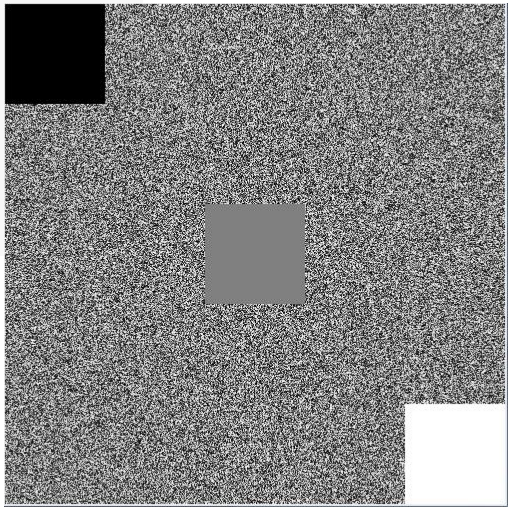
Note: we will not need this later

And now for some nicer stuff

```python
def black_square(mat):
    ''' add a black square at upper left corner '''
    n,m = mat.dim()
    if n<100 or m<100:
        return None
    else:
        new = mat.copy()
        for i in range(100):
            for j in range(100):
                new[i,j] = 0
        return new

black_square(mat).display()
```



```
def three_squares(mat):
    ''' add a black square at upper left corner, grey at
    middle, and white at lower right corner'''
    n,m = mat.dim()
    if n<500 or m<500:
        return None
    else:
        new = mat.copy()
        for i in range (100):
            for j in range (100):
                new[i,j] = 0    # black square
        for i in range (200,300):
             for j in range (200,300):
                 new[i,j] = 128 # grey square
        for i in range (400,500):
            for j in range (400,500):
               new[i,j ]= 255  # white square
    return new

three_squares(mat).display()
```

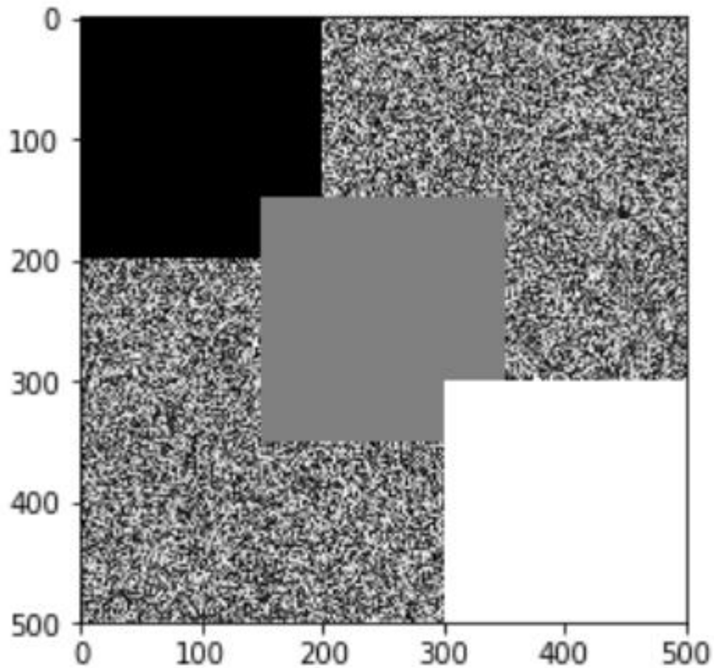


3 Squares: more generic version

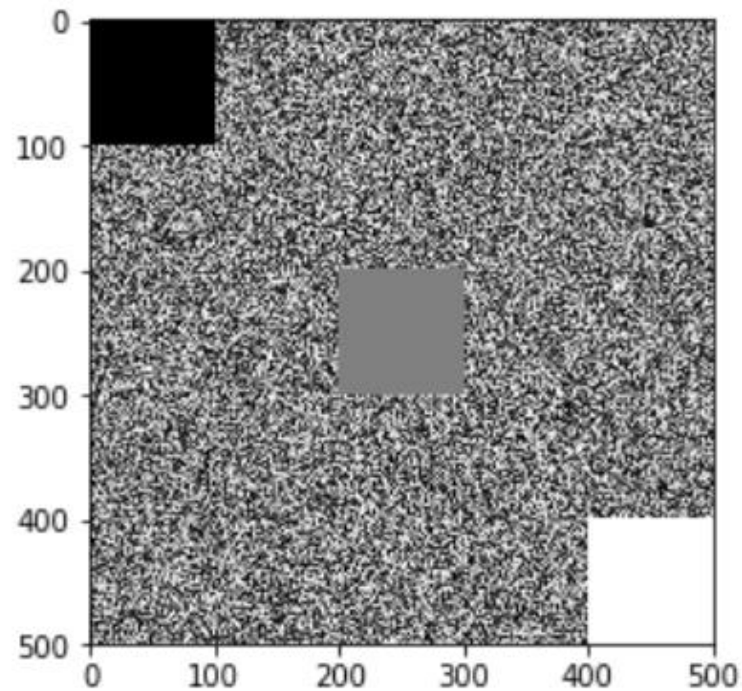
```
def three_squares(mat, size=100):
    ''' add a black square at upper left corner, grey at
    middle, and white at lower right corner'''
    n,m = mat.dim()
    if n<500 or m<500:
        return None
    else:
        new = mat.copy()
        for i in range (size):
           for j in range (size):
               new[i,j] = 0    # black square
        mid_m = int(m/2 - size/2)
        mid_n = int(n/2 - size/2)

        for i in range (mid_m, mid_m+size):
            for j in range (mid_n, mid_n+size):
                new[i,j] = 128 # grey square

        for i in range (n-size,n):
           for j in range (m-size,m):
              new[i,j]= 255   # white square
    return new

three_squares(mat).display()
```

3 Squares: more generic version

```
three_squares(mat, 200).display()
```



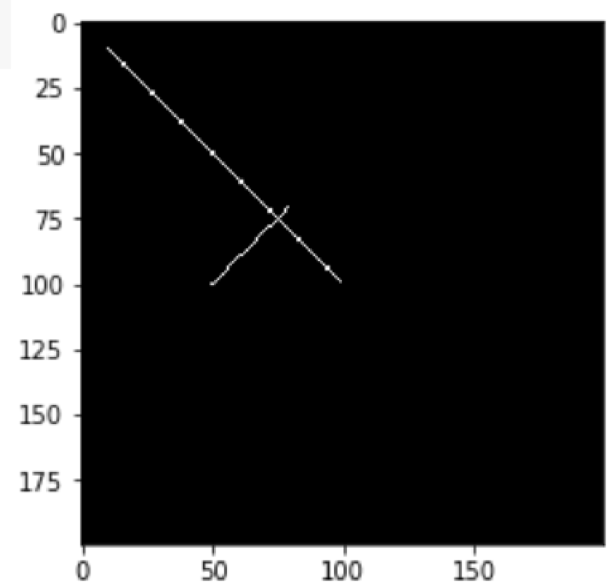

```
three_squares(mat).display()
```



Simple Synthetic Images: Points and Lines

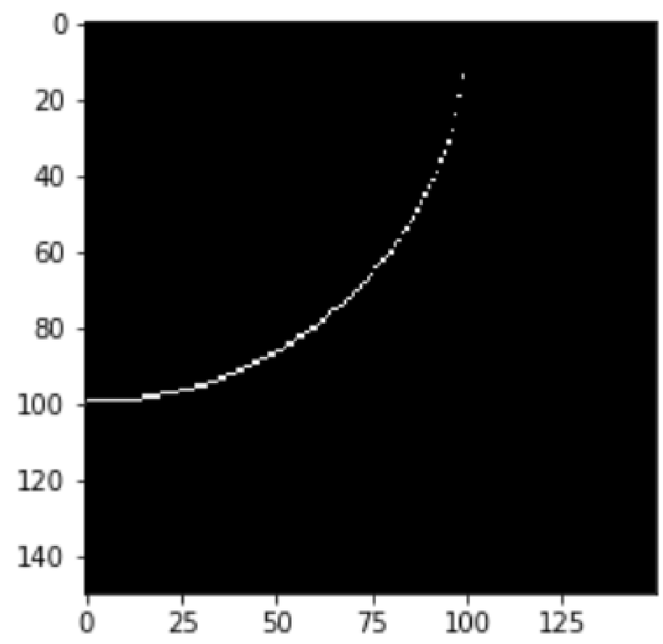
```
def draw_pixel(mat, y, x):
    mat[y, x]=255

def draw_line(mat, xs, ys):
    for x,y in zip(xs,ys):
        draw_pixel(mat,y,x)
```

```
mat1 = Matrix(200,200)
draw_line(mat1, range(10,100), range(10,100))
draw_line(mat1, range(50,80), range(100,70,-1))
mat1.display()
```



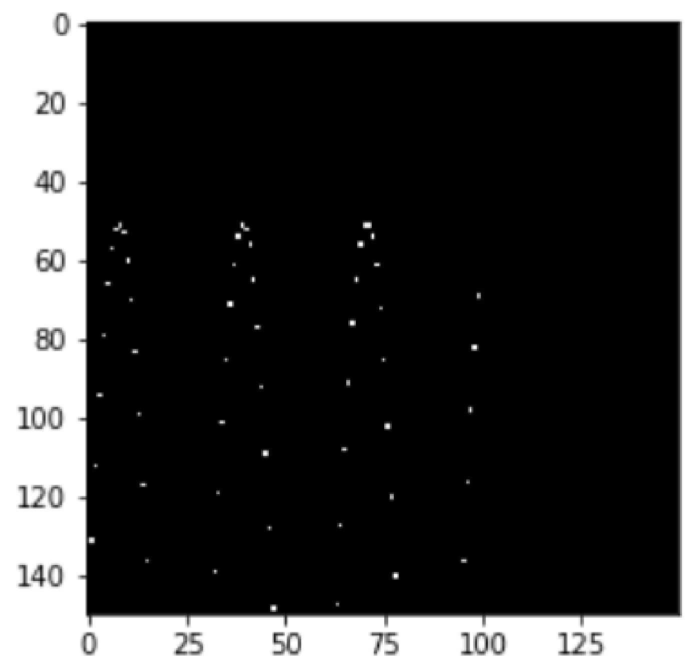
Simple Synthetic Images: Drawing Functions

```
def draw_function(mat, xs, f):
    rows,cols = mat.dim()
    for x in xs:
        if 0<=x<=cols:
            fx = f(x)
            if 0<=fx<=rows:
                draw_pixel(mat, int(fx), x)

mat2 = Matrix(150,150)
def func(x):
    return np.sqrt(10000-x**2)
xs=np.arange(0,100)
draw_function(mat2, range(100), func)
mat2.display()
```



Drawing Functions where “bottom is down”

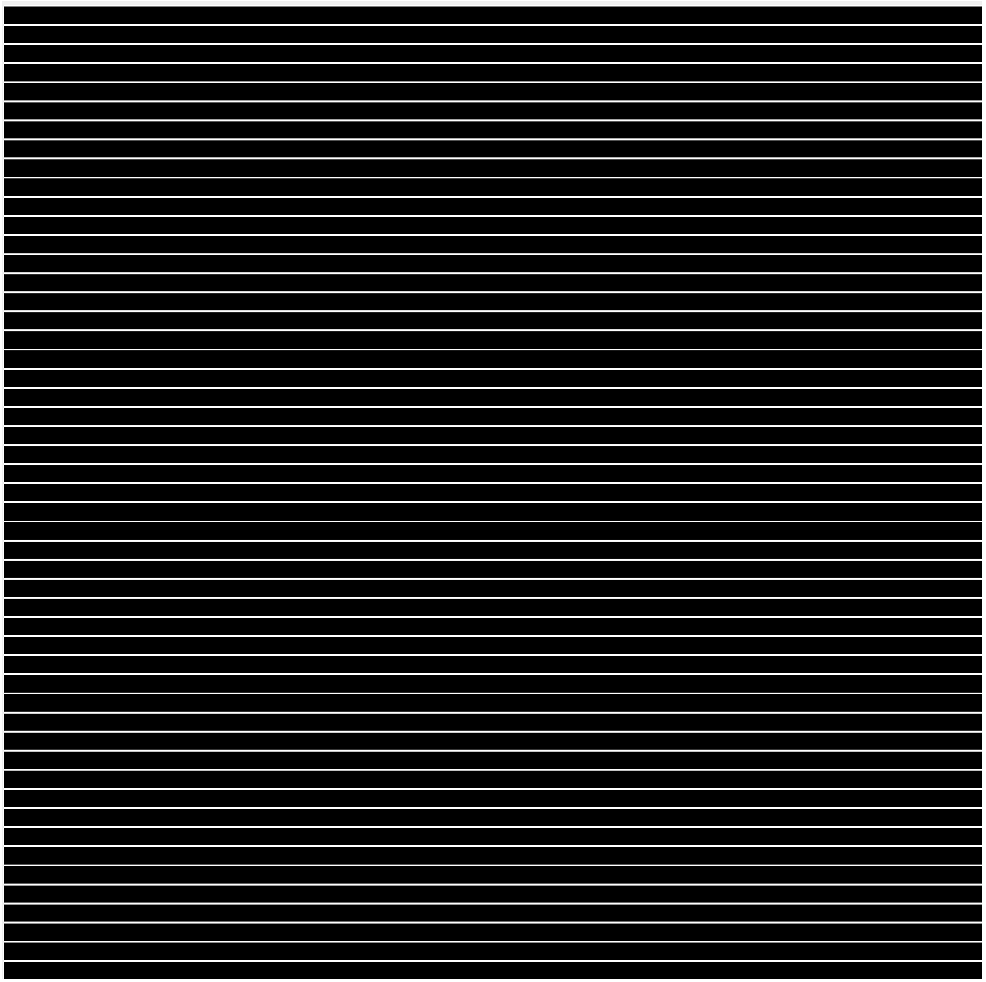
```
def draw_pixel(mat, y, x):
    rows,cols = mat.dim()
    if 0<y<rows and 0<x<cols:
        mat[rows-y, x]=255

def draw_function(mat, xs, f):
    rows,cols = mat.dim()
    for x in xs:
        if 0<=x<=cols:
            fx = f(x)
            if 0<=fx<=rows:
                draw_pixel(mat, int(fx), x)

mat2 = Matrix(150,150)
def func(x):
    return 100*np.sin(0.2*x)
xs=np.arange(0,100)
draw_function(mat2, range(100), func)
mat2.display()
```



Simple Synthetic Images: Lines and More

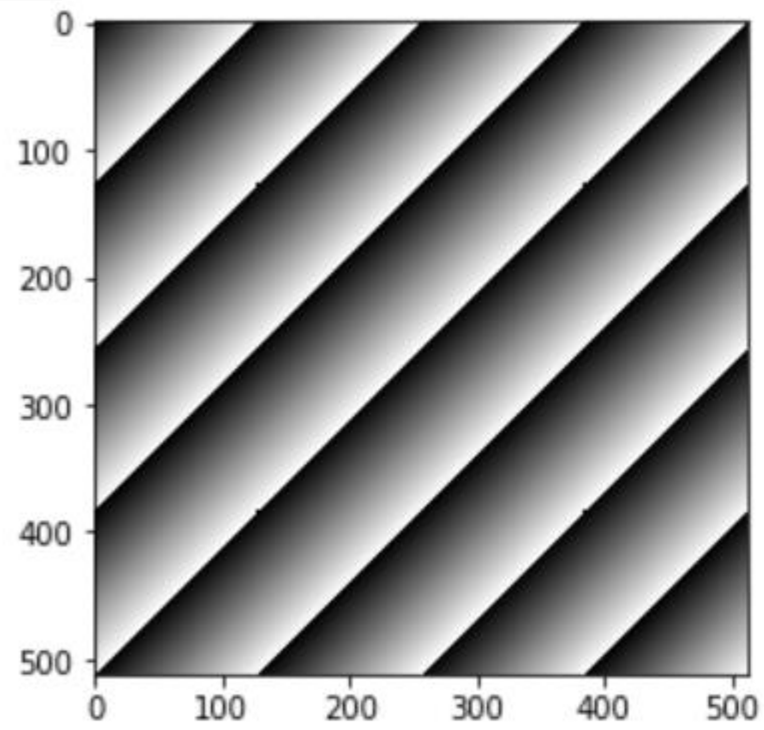
```
def horizontal(size=512):
    horizontal_lines = Matrix(size,size)

    for i in range(1,size):
        if i%10 == 0:
            for j in range(size):
                horizontal_lines[i,j] = 255

    return horizontal_lines

im = horizontal(512)
im.display( )
```

Displaying Synthetic Images: Lines and More



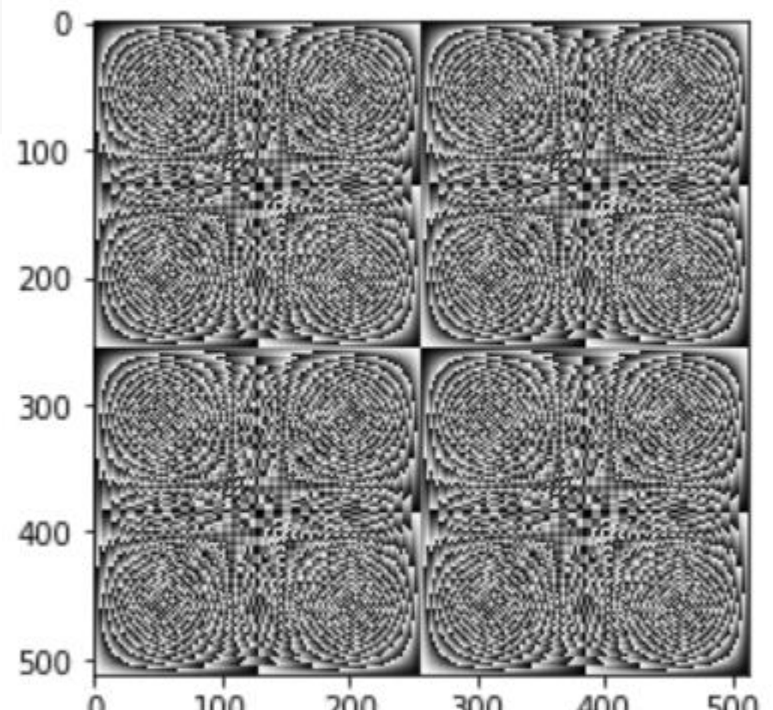
Simple Synthetic Images: Diagonal Lines

```
def diagonals(c=1):
    surprise = Matrix(512,512)

    for i in range(512):
        for j in range(512):
            surprise[i,j] = (c*(i+j)) % 256

    return surprise
diagonals(c=2).display()
```



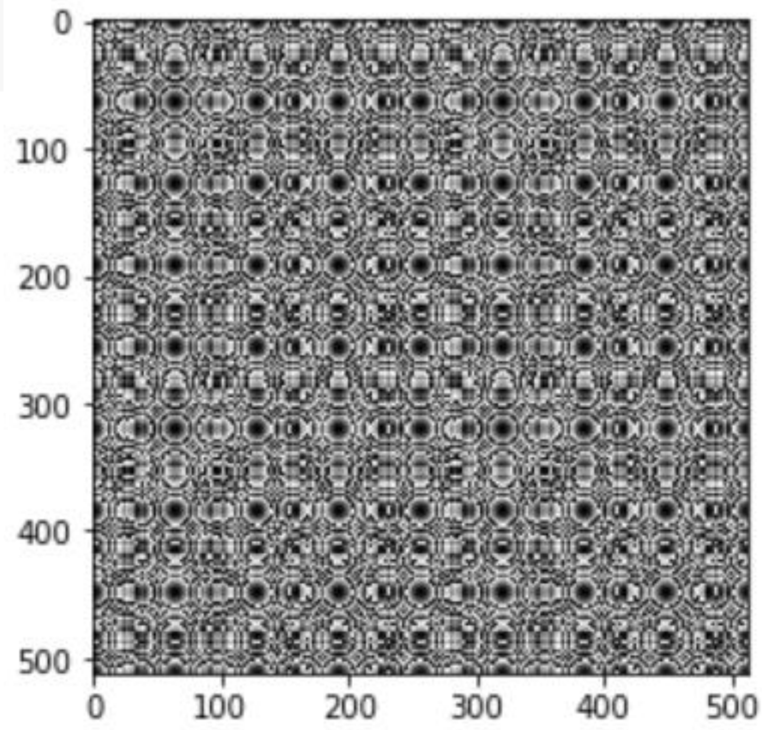
Simple Synthetic Images: Product and Circles

```
def product(c=1):
    surprise = Matrix(512,512)

    for i in range(512):
        for j in range(512):
            surprise[i,j] = (c*(i*j))% 256

    #print(surprise[:10,:10])
    return surprise
product(c=1).display()
```



Simple Synthetic Images: Product and Circles

```
def circles(c=2):
    surprise = Matrix(512,512)

    for i in range(512):
        for j in range(512):
            surprise[i,j] = (c * (i**2 + j**2))% 256
    print(surprise[:10,:10])
    return surprise
circles().display()
```



Exercises:

1. Go to method three squares (slide 29) and change the squares position so the white square is in the upper-left corner (and the black – in the lower right)
2. In the code in slide33, define func2 of x as -10*x^2 + 20x-30, and plot this function instead of func
3. Add a method called setColor(self, num), which sets all pixels of an image to num
   - For example img.setColor(0) will paint it black
4. Add a method called fourSquares(self, size), which draws 4 black squares at the 4 corners.