ise_lec21_images

November 1, 2019

In [1]:
```
import matplotlib.pyplot as plt
import numpy as np

##############################################################################
###    class Matrix
###    This is our own Matrix class with an additional display() method for image visual
###    Also, functions 'save' and 'load' enable working with .bitmap files
###    Note that __repr__ avoids printing very large matrices (would stuck IDLE)
##############################################################################

class Matrix:
    """
    Represents a rectangular matrix with n rows and m columns.
    """

    def __init__(self, n, m, val=0):
        """
        Create an n-by-m matrix of val's.
        Inner representation: list of lists (rows)
        """
        assert n > 0 and m > 0
        #self.rows = [[val]*m]*n #why this is bad?
        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

    def copy(self):
        ''' brand new copy of matrix '''
```

```
        n,m = self.dim()
        new = Matrix(n,m)
        for i in range (n):
            for j in range (m):
                new[i,j] = self[i,j]
        return new

    # 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

    #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

    # arithmetic operations
    #######################
    def entrywise_op(self, other, op):
        if not isinstance(other, Matrix):
            return NotImplemented
        assert self.dim() == other.dim()
        n,m = self.dim()
        M = Matrix(n,m)
        for i in range(n):
            for j in range(m):
                M[i,j] = op(self[i,j], other[i,j])
```

```
        return M

    def __add__(self, other):
        return self.entrywise_op(other,lambda x,y:x+y)


    def __sub__(self, other):
        return self.entrywise_op(other,lambda x,y:x-y)

    def __neg__(self):
        n,m = self.dim()
        return Matrix(n,m) - self


    def __mul__(self, other):
        if isinstance(other, Matrix):
            return self.multiply_by_matrix(other)
        elif isinstance(other, (int, float, complex)):
            return self.multiply_by_scalar(other)
        else:
            return NotImplemented

    __rmul__ = __mul__

    def multiply_by_scalar(self, val):
        n,m = self.dim()
        return self.entrywise_op(Matrix(n,m,val), lambda x,y :x*y)
###a more efficient version, memory-wise.
##        n,m = self.dim()
##        M = Matrix(n,m)
##        for i in range(n):
##            for j in range(m):
##                M[i,j] = self[i,j] * val
##        return M

    def multiply_by_matrix(self, other):
        assert isinstance(other, Matrix)
        n,m = self.dim()
        n2,m2 = other.dim()
        assert m == n2
        M = Matrix(n,m2)
        for i in range(n):
            for j in range(m2):
                M[i,j] = sum(self[i,k] * other[k,j] for k in range(m))
        return M


    # Input/output
```

```
            ###############
            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()

            @staticmethod
            def load(filename):
                f = open(filename)
                line = f.readline()
                n,m = [int(x) for x in line.split()]
                result = Matrix(n,m)
                for i in range(n):
                    line = f.readline()
                    row = [int(x) for x in line.split()]
                    assert len(row) == m
                    result.rows[i] = row
                return result

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

                plt.show()
```

```
In [2]: m=Matrix(5,6)
```

```
In [3]: m
```

```
Out[3]: <Matrix [[0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0],
```

```
In [4]: 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')
```

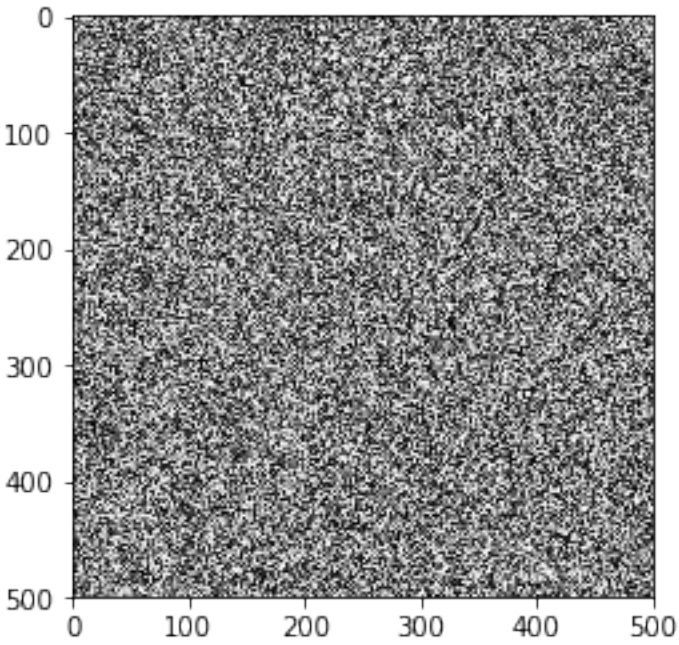
```
In [5]: import random
        n = 500
        m = 500
        mat = Matrix(n,m)

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

```
Matrix too large, specify submatrix
```

```
In [6]: mat[3:5, 4:8]
```

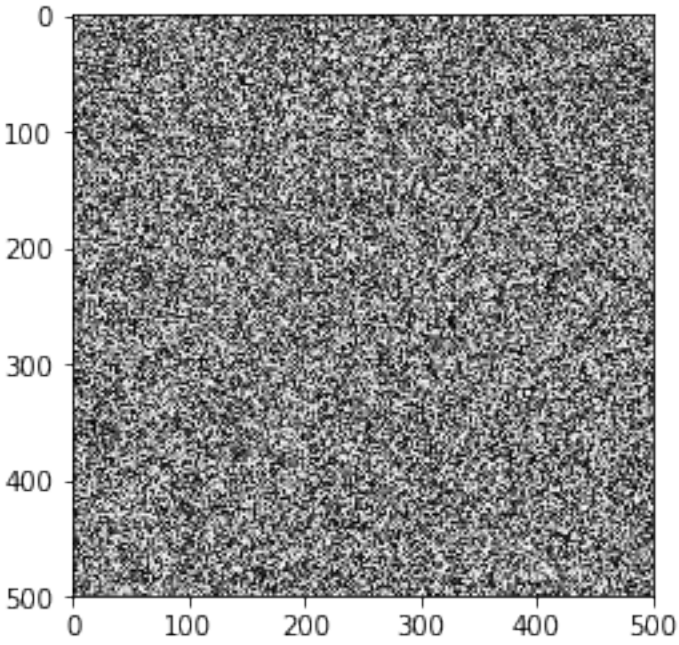
```
Out[6]: <Matrix [[142, 90, 164, 149], [50, 78, 222, 235]]>
```

```
In [7]: mat.display()
```

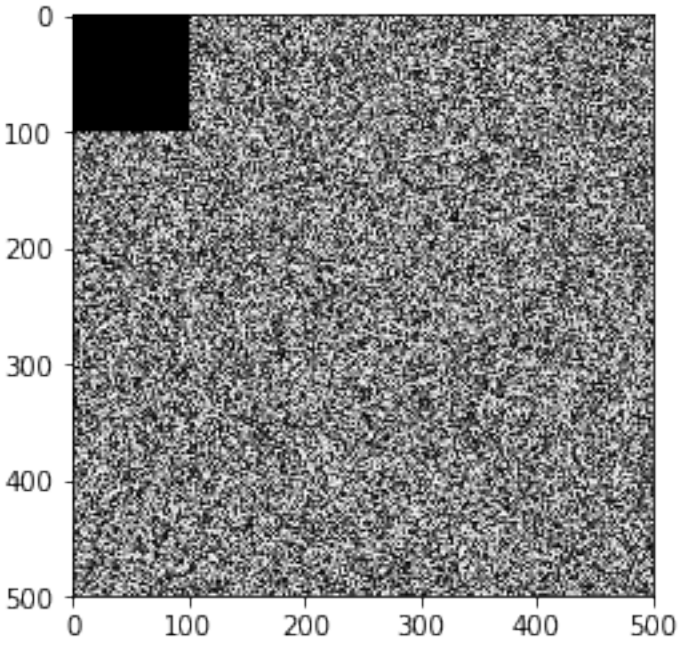



```
In [8]: mat.save("./rand_image.bitmap")
```

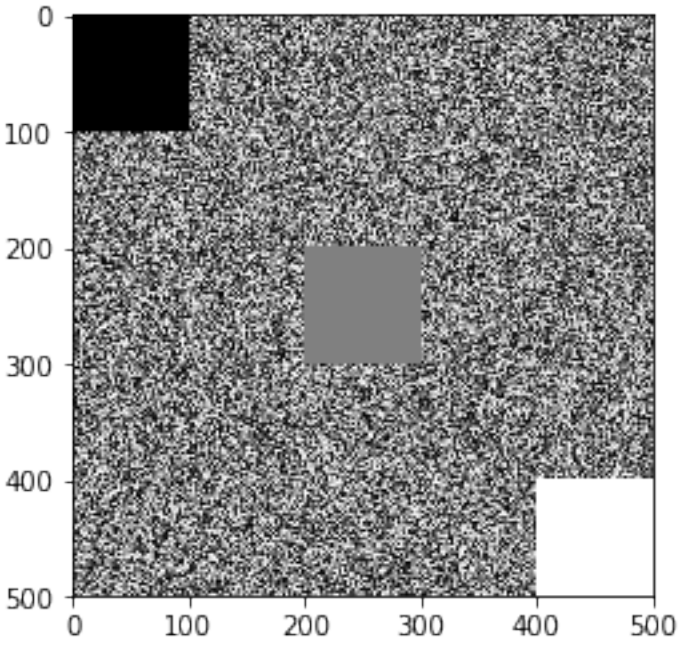
```
In [9]: mat2 = Matrix.load("./rand_image.bitmap")
        mat2.display()
```



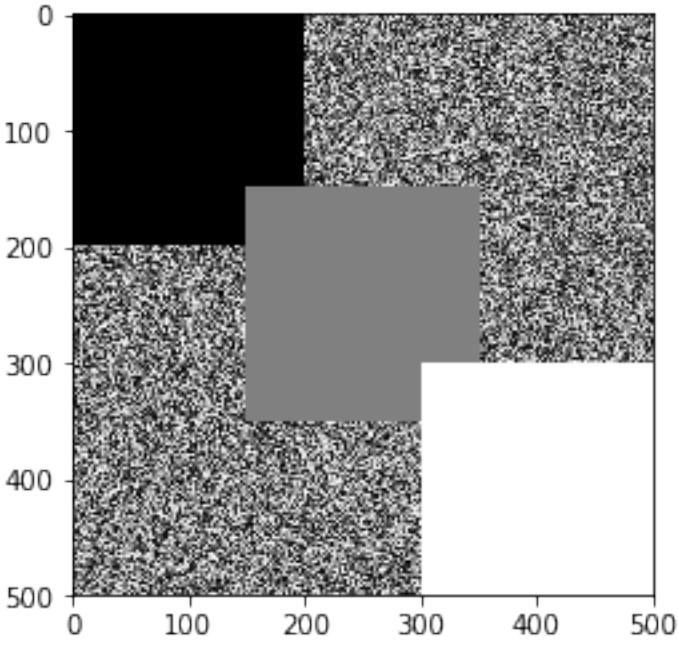

```
In [10]: def black_square(mat):
             ''' add a black square at upper left corner '''
             n,m = mat.dim()
             if n<100 or m<100:
                 return None
             else:
                 new = mat.copy()
                 for i in range(100):
                     for j in range(100):
                         new[i,j] = 0
                 return new

         black_square(mat).display()
```




```
In [29]: 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()
```



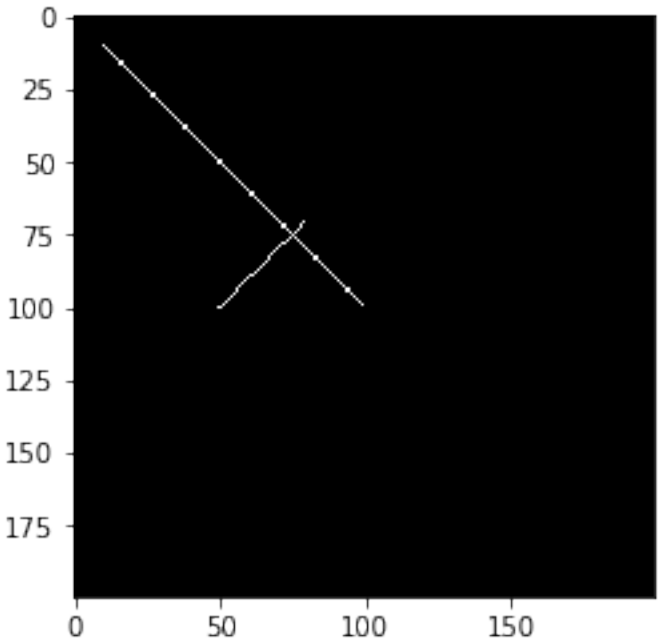

```
In [28]: three_squares(mat, 200).display()
```



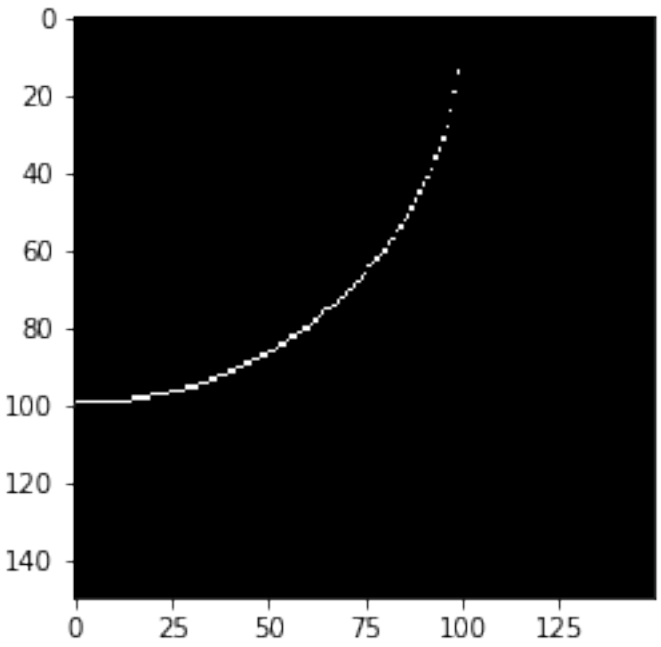

```
In [13]: def draw_pixel(mat, y, x):
             mat[y, x]=255
```

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

"""
def draw_line_steps(mat, xstart, ystart, xend, yend):
    #xstart,xend = np.minimum(xstart,xend),np.maximum(xstart,xend)
    #ystart,yend = np.minimum(ystart,yend),np.maximum(ystart,yend)
    xrange = xend-xstart
    yrange = yend-ystart
    if xrange==0 or yrange==0:
        if xrange==0:
            ys = np.arange(ystart, yend)
            xs = np.ones(len(ys))*xstart
        if yrange==0:
            xs = np.arange(xstart, xend)
            ys = np.ones(len(xs))*ystart
        draw_line(mat,xs,ys)
        return
    x_cur, y_cur = xstart, ystart
    x_step = np.minimum(1, xrange/yrange)
    y_step = np.minimum(1, yrange/xrange)
    while (x_cur<=xend and y_cur<=yend):
        #print("Drawing at:", x_cur, y_cur)
        draw_pixel(mat,int(y_cur),int(x_cur))
        y_cur +=y_step
        x_cur +=x_step
draw_line_steps(mat1, 0,0,100,100)
"""
"""
def draw_function2(mat, xs, f):
    rows,cols = mat.dim()
    fxs = np.array( [f(x) for x in xs] )
    for ix,x in enumerate(xs):
        fx = f(x)
        if 0<=x<=cols and 0<=fx<rows:
            draw_pixel(mat, int(fx), x)
            if ix<len(xs)-1:
                next_x = xs[ix+1]
                next_fx = fxs[ix+1]
                if 0<=next_x<=cols and 0<=next_fx<rows:
                    draw_line_steps(mat, int(fx), x, int(next_fx), next_x)

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

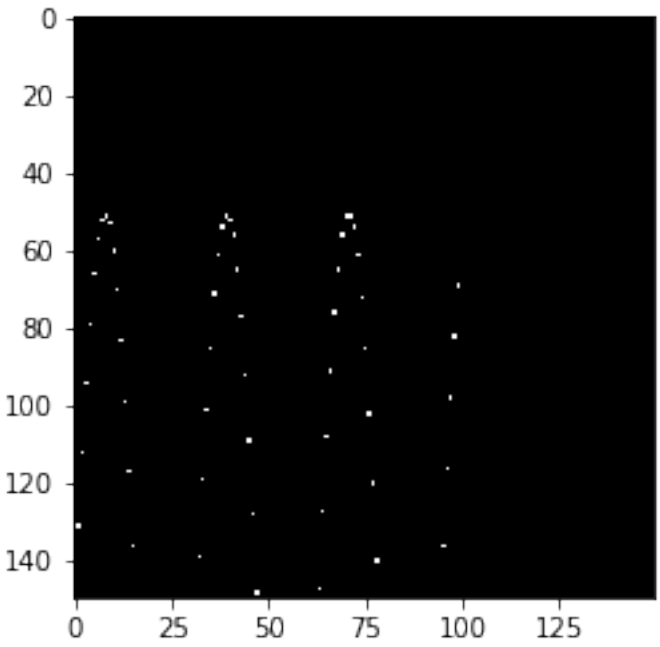
```
mat1.display()
```



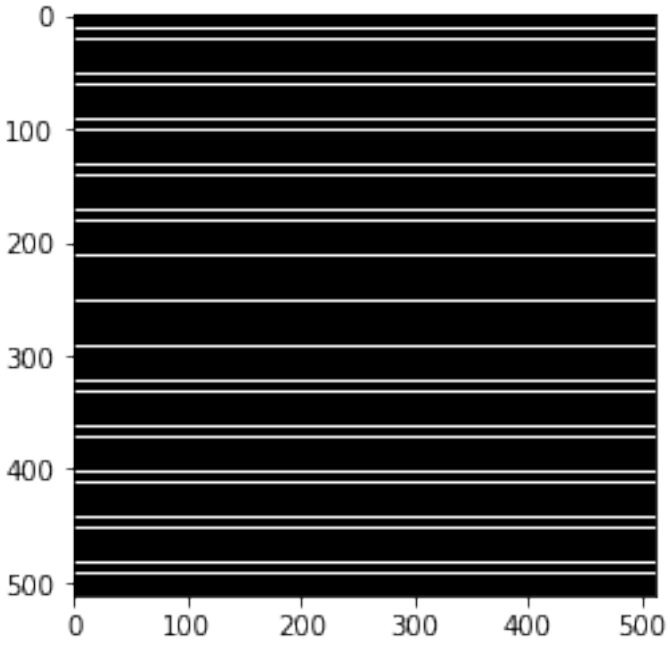
In [14]:
```
def draw_pixel(mat, y, x):
    mat[y, x]=255

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



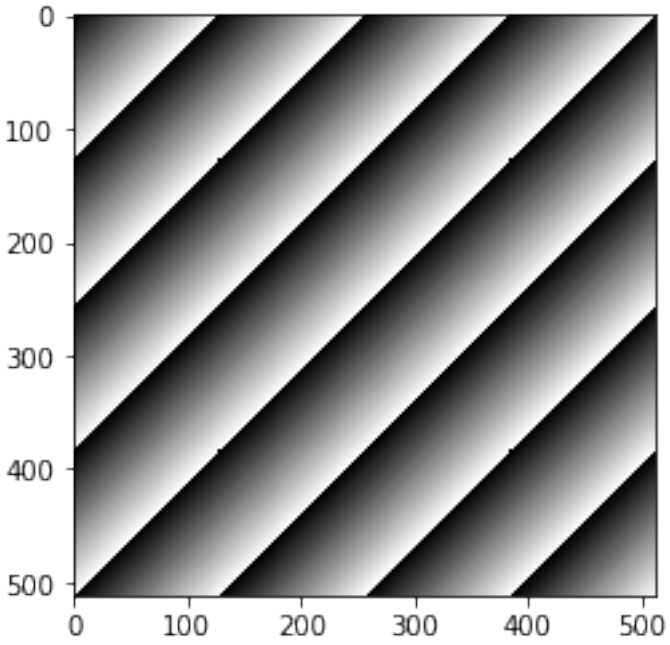

```
In [15]: def draw_pixel(mat, y, x):
             rows,cols = mat.dim()
             if 0<y<rows and 0<x<cols:
                 mat[rows-y, x]=255

         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()
```



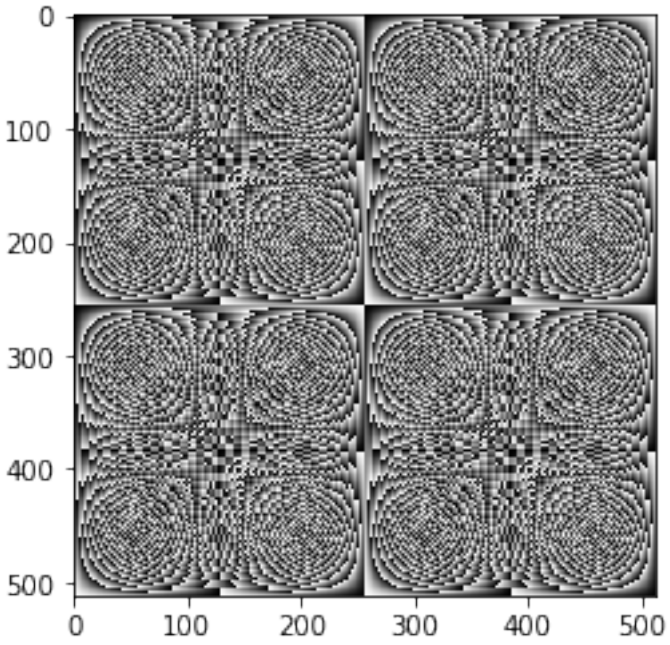

```
In [16]: def horizontal(size=512):
             horizontal_lines = Matrix(size,size)

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

             return horizontal_lines

         horizontal().display()
```



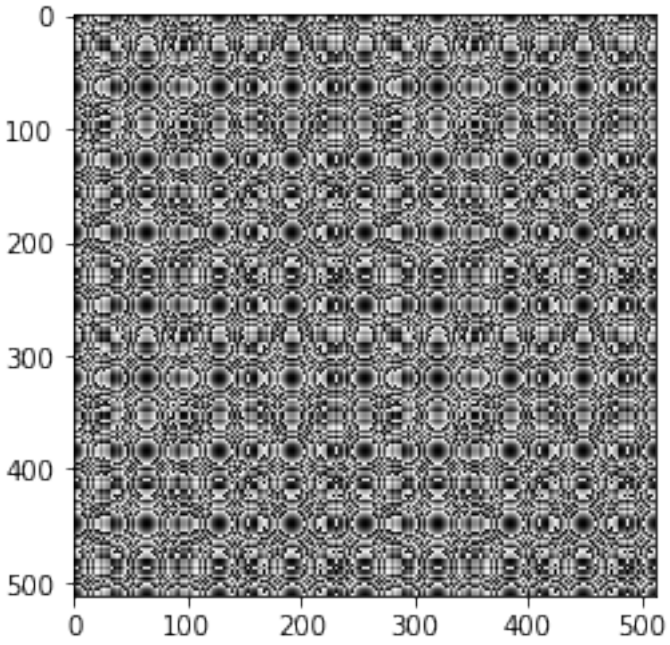

```
In [17]: def diagonals(c=1):
             surprise = Matrix(512,512)

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

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



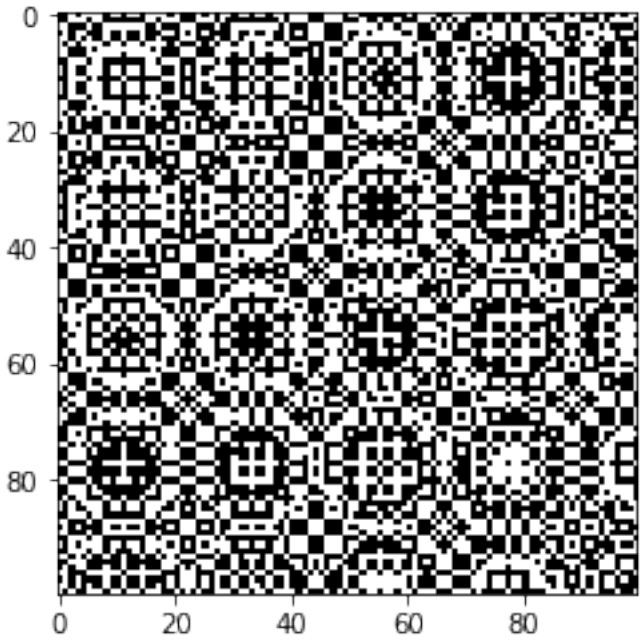

```
In [18]: def product(c=1):
             surprise = Matrix(512,512)

             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()
```



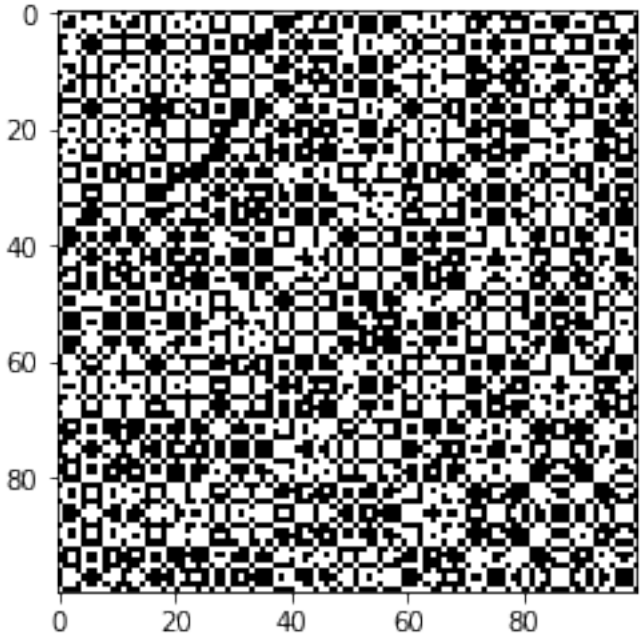

```
In [19]: def circles(c=2):
             surprise = Matrix(512,512)

             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()
```



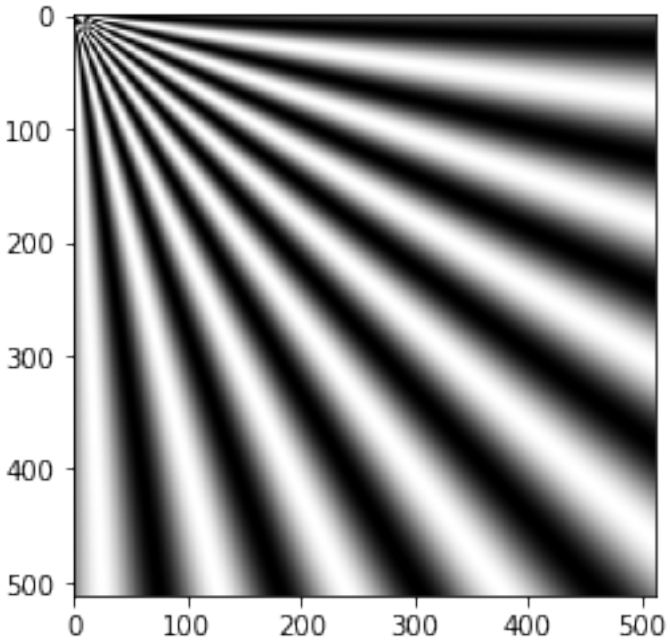

```
In [20]: import math
         def synthetic(n, m, func):
             """ produces a synthetic image "upon request" """
             new = Matrix(n,m)
             for i in range(n):
                 for j in range(m):
                     new[i,j] = func(i,j)/256
             return new

         def sin_mul(i,j):
             return math.sin((i**2+j**2))%256
         synthetic(100,100,sin_mul).display()
```




```
In [21]: def cos_mul(i,j):
             return math.cos(4*(i**2 + j**2))%256
         synthetic(100,100,cos_mul).display()
```



In [22]:
```
import cmath
c = synthetic(512, 512, lambda x,y: \
              100*math.sin(32*cmath.phase(complex(x,y))))
c. display ()
```



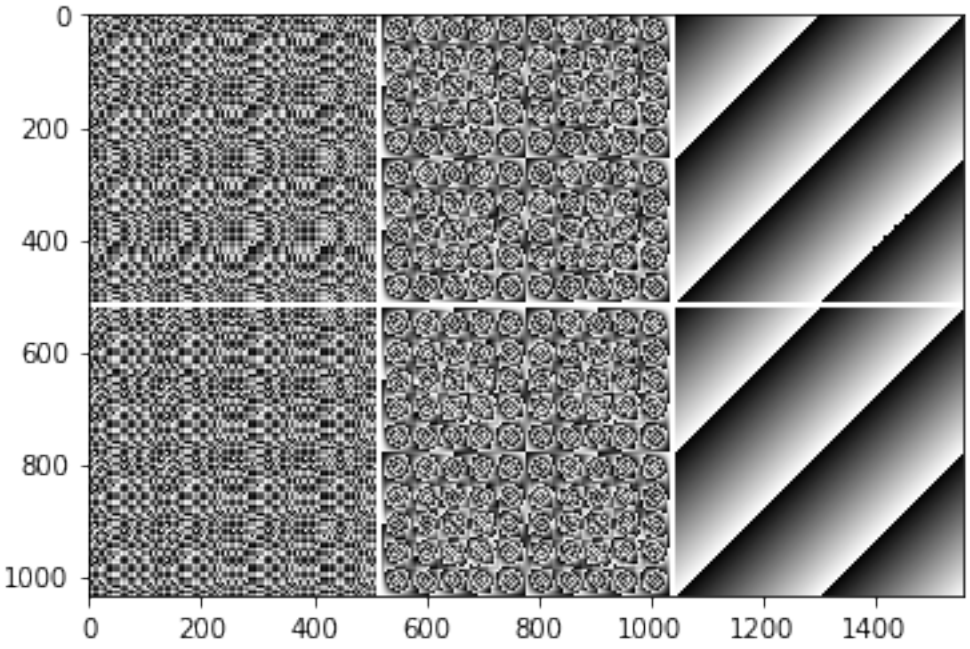
In [23]:
```
#######################
## Useful operation
#######################

def join_h(mat1, mat2):
    """ joins two matrices, side by side with some separation """
    n1,m1 = mat1.dim()
    n2,m2 = mat2.dim()
    m = m1+m2+10
    n = max(n1,n2)
    new = Matrix(n, m, val=255)  # fill new matrix with white pixels

    new[:n1,:m1] = mat1
    new[:n2,m1+10:m] = mat2

    return new
```

In [24]:
```
a = circles()
b = product()
bc = join(a, b, direction='h')
bc.display()
```

```
---------------------------------------------------------------------------

NameError                                 Traceback (most recent call last)

<ipython-input-24-4bc26430e461> in <module>()
  1 a = circles()
  2 b = product()
----> 3 bc = join(a, b, direction='h')
  4 bc.display()


NameError: name 'join' is not defined
```

In [25]:
```
def join_v(mat1, mat2):
    """ joins two matrices, vertically with some separation """
    n1,m1 = mat1.dim()
    n2,m2 = mat2.dim()
    n = n1+n2+10
    m = max(m1,m2)
    new = Matrix(n, m, val=255)  # fill new matrix with white pixels

    new[:n1,:m1] = mat1
    new[n1+10:n,:m2] = mat2

    return new

def join(*mats, direction):
    ''' *mats enables a variable number of parameters.
        direction is either 'h' or 'v', for horizontal
        or vertical join, respectively '''
    func = join_v if direction == 'v' else join_h
    res = mats[0] #first matrix parameter
    for mat in mats[1:]:
        res = func(res, mat)
    return res
```

In [26]:
```
#synthetic(300,300,cos_mul).display()
a = circles()
b = product()
c = diagonals()
abc = join(a, b, c, direction='h')
abcd = join(abc, abc,  direction='v')
abcd.display()
```


0
200
400
600
800
1000
0
200
400
600
800
1000
1200
1400