Algebra Lineal

In [1]:

```
#La linea de abajo es para graficar aqui mismo los plots
%pylab inline
import numpy as np
import matplotlib.pyplot as plt
import scipy.linalg # Biblioteca para algebra lineal
```

```
Populating the interactive namespace from numpy and matplotlib
```

Algebra de matrices

Los arreglos de **numpy** no se comportan como las *matrices* de sus clases de algebra lineal.

En lugar de ello, hacen *broadcasting*, como hemos visto en las clases pasadas. Recordemos que *broadcasting* es mapear las operaciones a cada uno de los elementos del arreglo (*array*).

¿Pero que pasa si queremos hacer operaciones matriciales? Bueno, **numpy** nos ofrece las siguientes opciones.

Definamos el arreglo $\mathbf{A}$

In [2]:

```
A = array([[n+m*10 for n in range(1,5)] for m in range(1,5)])

A
```

Out[2]:

```
array([[11, 12, 13, 14],
       [21, 22, 23, 24],
       [31, 32, 33, 34],
       [41, 42, 43, 44]])
```

El arreglo $\mathbf{A}$, es eso, un arreglo (*array*), es el mismo objeto que hemos visto con anterioridad. **Numpy** soporta (en beneficio de los usuarios de `matlab` / `GNU Octave`) el objeto `matrix` .

In [3]:

```
#Lo hace una matriz
Am = np.matrix(A)
Am
```

Out[3]:

```
matrix([[11, 12, 13, 14],
        [21, 22, 23, 24],
        [31, 32, 33, 34],
        [41, 42, 43, 44]])
```

Como probablemente en un futuro se topen con cosas de `matlab / GNU Octave` les recomiendo esta [liga](https://numpy.org/doc/stable/user/numpy-for-matlab-users.html)

Nada nuevo en cuanto las dimensiones de $\mathbf{A}$ y A_m:

In [4]:

```
print(A.shape)
print(Am.shape)
```

```
(4, 4)
(4, 4)
```

Pero recordemos de la clase pasada que el *slicing* devuelve arreglos unidimensionales

In [5]:

```
y = A[:, 0]
print(y)
print(y.shape)
```

```
[11 21 31 41]
(4,)
```

En lugar de arreglos bidimensionales (Recuerden sus clases de algebra lineal y piensen en lo que llaman *vectores*...)

In [6]:

```
ym = Am[:,0]
print(ym)
print(ym.shape)
```

```
[[11]
 [21]
 [31]
 [41]]
(4, 1)
```

Obviamente este comportamiento se puede simular con arreglos y *slicing*, pero es más elaborado:

In [7]:

```
y = A[:,:1]
print(y)
print(y.shape)
```

```
[[11]
 [21]
 [31]
 [41]]
(4, 1)
```

Las operaciones en matrices (usando la clase `matrix`) son como sigue:

In [10]:

```
Am*ym
#Multiplicacion de matrices
```

Out[10]:

```
matrix([[1350],
        [2390],
        [3430],
        [4470]])
```

In [11]:

```
ym*Am #No puede por las dimensiones
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-11-8111b5980667> in <module>
----> 1 ym*Am #No puede por las dimensiones

/opt/conda/lib/python3.8/site-packages/numpy/matrixlib/defmatrix.py in __mul__(self, other)
    216         if isinstance(other, (N.ndarray, list, tuple)) :
    217             # This promotes 1-D vectors to row vectors
--> 218             return N.dot(self, asmatrix(other))
    219         if isscalar(other) or not hasattr(other, '__rmul__') :
    220             return N.dot(self, other)

<__array_function__ internals> in dot(*args, **kwargs)

ValueError: shapes (4,1) and (4,4) not aligned: 1 (dim 1) != 4 (dim 0)
```

Ejercicio ¿Por qué no funcionó?

In [12]:

```
ym.T #Transpuesta
```

Out[12]:

```
matrix([[11, 21, 31, 41]])
```

In [13]:

```
Am.T
```

Out[13]:

```
matrix([[11, 21, 31, 41],
        [12, 22, 32, 42],
        [13, 23, 33, 43],
        [14, 24, 34, 44]])
```

Una operación común es el producto $y^T A y$ (esto es simplemente el *producto interno*)

In [14]:

```
ym.T*Am*ym
```

Out[14]:

```
matrix([[354640]])
```

In [15]:

```
Am*Am #Elevar al cuadrado la matriz (multiplicacion)
```

Out[15]:

```
matrix([[1350, 1400, 1450, 1500],
        [2390, 2480, 2570, 2660],
        [3430, 3560, 3690, 3820],
        [4470, 4640, 4810, 4980]])
```

In [16]:

```
Am**2 # Esto es equivalente a Am * Am
```

Out[16]:

```
matrix([[1350, 1400, 1450, 1500],
        [2390, 2480, 2570, 2660],
        [3430, 3560, 3690, 3820],
        [4470, 4640, 4810, 4980]])
```

In [17]:

```
Am + ym
```

Out[17]:

```
matrix([[22, 23, 24, 25],
        [42, 43, 44, 45],
        [62, 63, 64, 65],
        [82, 83, 84, 85]])
```

In [18]:

```
Am**(-1) #No puede calcular la inversa porque es singular
```

```
---------------------------------------------------------------------------
LinAlgError                               Traceback (most recent call last)
<ipython-input-18-e7013444d8a7> in <module>
----> 1 Am**(-1) #No puede calcular la inversa porque es singular

/opt/conda/lib/python3.8/site-packages/numpy/matrixlib/defmatrix.py in __pow__(self, other)
    229
    230     def __pow__(self, other):
--> 231         return matrix_power(self, other)
    232
    233     def __ipow__(self, other):

<__array_function__ internals> in matrix_power(*args, **kwargs)

/opt/conda/lib/python3.8/site-packages/numpy/linalg/linalg.py in matrix_power(a, n)
    642
    643     elif n < 0:
--> 644         a = inv(a)
    645         n = abs(n)
    646

<__array_function__ internals> in inv(*args, **kwargs)

/opt/conda/lib/python3.8/site-packages/numpy/linalg/linalg.py in inv(a)
    544     signature = 'D->D' if isComplexType(t) else 'd->d'
    545     extobj = get_linalg_error_extobj(_raise_linalgerror_singular)
--> 546     ainv = _umath_linalg.inv(a, signature=signature, extobj=extobj)
    547     return wrap(ainv.astype(result_t, copy=False))
    548

/opt/conda/lib/python3.8/site-packages/numpy/linalg/linalg.py in _raise_linalgerror_singular(err, fl
ag)
     86
     87 def _raise_linalgerror_singular(err, flag):
---> 88     raise LinAlgError("Singular matrix")
     89
     90 def _raise_linalgerror_nonposdef(err, flag):

LinAlgError: Singular matrix
```

In [19]:

```
Id = matrix(np.identity(4))
Id
```

Out[19]:

```
matrix([[1., 0., 0., 0.],
        [0., 1., 0., 0.],
        [0., 0., 1., 0.],
        [0., 0., 0., 1.]])
```

Soluciones de sistemas de ecuaciones

Los sistemas de ecuaciones lineales se pueden plantear como un problema matricial, del tipo $\mathbf{Ax} = \mathbf{B}$, por ejemplo:

$$3x + 6y - 5z = 12$$

$$x - 3y + 2z = -2$$

$$5x - y + 4z = 10$$

La solución de las ecuaciones matriciales $\mathbf{Ax} = \mathbf{B}$, es $\mathbf{x} = \mathbf{A}^{-1}\mathbf{B}$ (Si la matriz $\mathbf{A}$ es invertible, claro está)

In [20]:

```
A = np.matrix([[3,6,-5],
              [1,-3,2],
              [5,-1,4]])
A
```

Out[20]:

```
matrix([[ 3,  6, -5],
        [ 1, -3,  2],
        [ 5, -1,  4]])
```

In [21]:

```
B = np.matrix([[12],
               [-2],
               [10]])
B
```

Out[21]:

```
matrix([[12],
        [-2],
        [10]])
```

In [22]:

```
#Para solucionar el sistema
x = A**(-1)*B
print(x)
```

```
[[1.75]
 [1.75]
 [0.75]]
```

In [23]:

```
A*x
```

Out[23]:

```
matrix([[12.],
        [-2.],
        [10.]])
```

Es importante tener en mente que las matrices generalmente no son invertibles, por lo que este método de solución, no siempre funciona.

El invertir matrices es un proceso largo y pesado que además puede ser demasiado cálculo para lo que se requiere.

Transformaciones

In [24]:

```
A = np.matrix("1,2,3;4,5,6")
A
```

Out[24]:

```
matrix([[1, 2, 3],
        [4, 5, 6]])
```

In [25]:

```
C = matrix([[1j, 2j], [3j, 4j]])
C
```

Out[25]:

```
matrix([[0.+1.j, 0.+2.j],
        [0.+3.j, 0.+4.j]])
```

El conjugado de una matriz compleja $\mathbf{C}$

In [26]:

```
conjugate(C)
```

Out[26]:

```
matrix([[0.-1.j, 0.-2.j],
        [0.-3.j, 0.-4.j]])
```

El *hermitianno* de una matriz (es decir, el *conjugado* y la *traspuesta*)

In [27]:

```
C.H
```

Out[27]:

```
matrix([[0.-1.j, 0.-3.j],
        [0.-2.j, 0.-4.j]])
```

In [28]:

```
(conjugate(C)).T
```

Out[28]:

```
matrix([[0.-1.j, 0.-3.j],
        [0.-2.j, 0.-4.j]])
```

El *hermitiano* de una matriz real (como $\mathbf{A}$) es simplemente la *traspuesta*

In [29]:

```
print( A.H )
print (A.T)
```

```
[[1 4]
 [2 5]
 [3 6]]
[[1 4]
 [2 5]
 [3 6]]
```

La parte $\Re$ e $\Im$ de una matriz es

In [30]:

```
real(C) # también funciona C.real
```

Out[30]:

```
matrix([[0., 0.],
        [0., 0.]])
```

In [31]:

```
imag(C) # también funciona C.imag
```

Out[31]:

```
matrix([[1., 2.],
        [3., 4.]])
```

In [32]:

```
A.imag
```

Out[32]:

```
matrix([[0, 0, 0],
        [0, 0, 0]])
```

La inversa de una matriz

In [33]:

```
inv(C)
```

Out[33]:

```
matrix([[0.+2.j , 0.-1.j ],
        [0.-1.5j, 0.+0.5j]])
```

In [34]:

```
C.I
```

Out[34]:

```
matrix([[0.+2.j , 0.-1.j ],
        [0.-1.5j, 0.+0.5j]])
```

In [35]:

```
C**(-1) #Otra forma para calcular la inversa
```

Out[35]:

```
matrix([[0.+2.j , 0.-1.j ],
        [0.-1.5j, 0.+0.5j]])
```

In [36]:

```
C*C.I
```

Out[36]:

```
matrix([[1.0000000e+00+0.j, 0.0000000e+00+0.j],
        [8.8817842e-16+0.j, 1.0000000e+00+0.j]])
```

In [37]:

```
inv(C)*C
```

Out[37]:

```
matrix([[1.00000000e+00+0.j, 0.00000000e+00+0.j],
        [1.11022302e-16+0.j, 1.00000000e+00+0.j]])
```

Determinantes

In [38]:

```
A = np.matrix([[1,2],[3,4]])
A
```

Out[38]:

```
matrix([[1, 2],
        [3, 4]])
```

In [39]:

```
np.linalg.det(A)
```

Out[39]:

```
-2.0000000000000004
```

In [40]:

```
B = np.arange(1,10).reshape(3,3)
B = np.matrix(B)
B
```

Out[40]:

```
matrix([[1, 2, 3],
        [4, 5, 6],
        [7, 8, 9]])
```

In [41]:

```
np.linalg.det(B)
```

Out[41]:

```
0.0
```

Sean las matrices $\mathbf{A}$ y $\mathbf{B}$ definidas abajo, compruebe las propiedades $1-6$ de los determinantes como se muestran en la página de la [Wikipedia](http://en.wikipedia.org/wiki/Determinant)

In [42]:

```
A = np.matrix([[-2,2,-3],
               [-1,1,3],
               [2,0,-1]])
print(A)
```

```
[[-2  2 -3]
 [-1  1  3]
 [ 2  0 -1]]
```

In [43]:

```
B = np.matrix([[5, -3, 2],
               [1,0,2],
               [2,-1,3]])
print(B)
```

```
[[ 5 -3  2]
 [ 1  0  2]
 [ 2 -1  3]]
```

Ejercicio

In [44]:

```
#1
Id = matrix(np.identity(10))
np.linalg.det(Id)
```

Out[44]:

```
1.0
```

In [45]:

```
#2
det(A)-det(A.T)
```

Out[45]:

```
-7.105427357601002e-15
```

In [46]:

```
#3
det(inv(A))-(det(A)**(-1))
```

Out[46]:

```
-2.7755575615628914e-17
```

In [47]:

```
#4
det(A*B)-(det(A)*det(B))
```

Out[47]:

```
-4.263256414560601e-14
```

In [48]:

```
#5
det(10*A)-(10**(len(A))*det(A))
```

Out[48]:

```
1.0913936421275139e-11
```

In [49]:

```
#6
det(A+B)-(det(A)+det(B))
```

Out[49]:

```
-17.999999999999993
```

Ejercicio: Resuelva el sistema de ecuaciones lineales mostrado anteriormente, pero usando la [**Regla de Cramer**](http://en.wikipedia.org/wiki/Cramer's_rule)

$3x + 6y - 5z = 12$

$x - 3y + 2z = -2$

$5x - y + 4z = 10$

In [51]:

```
A = np.matrix([[3, 6, -5],
               [1,-3,2],
               [5,-1,4]])
```

In [52]:

```
b = np.matrix([[12],
               [-2],
               [10]])
```

In [53]:

```
A = np.matrix(np.random.rand(100,100))
b = np.matrix(np.random.rand(100,1))
```

In [54]:

```
for i in range(0,len(b)):
    C = A.copy()
    C[:,i]=b
    print(det(C)/det(A))
```

```
-3.7402075813996563
1.4517374490953017
0.7325419882726717
-1.1128135830658559
-3.3194674725044893
1.07871957606665
0.5825489745463622
1.3263013915374382
-2.2341649376548856
-1.0962509811650718
-6.071287043863551
-1.0653026523448244
0.3267542377713717
0.2009391497755881
0.7031353652375499
1.0697039907329935
-1.6916601645322087
-2.792695739342114
-0.600972258026933
-2.2515840302826238
2.5199732490989697
-1.3149552872157706
-4.10752974046788
-0.5066318071299463
0.7612339090944155
-1.0198851674141454
-2.854448299095275
-3.116254595254324
3.4578350038138166
0.6017832155123224
2.8949633409699285
-1.566335987265641
1.8439244590795723
0.734432251564111
-1.3779882362361935
-3.6954461805982493
3.2291305858750157
-0.8757406196880436
0.8322038412469538
-1.1821825521336429
0.7393259648839736
-2.5802584270136344
-4.918699650995987
-3.983492594140729
0.22285415444076775
-0.41150872994279025
```

```
0.1877809391517492
1.522711021220588
0.9787284839633272
3.576243203299452
-1.434260273550383
-1.9479786051665555
0.000694649184958262
0.6194895819866013
0.37443078197224794
0.7548035680658981
4.05998080244426
2.9419949537018484
-2.5833551513529014
1.004698077560169
-1.188009664946074
1.8221713697492992
3.448221332871899
0.6940345030421038
3.0198693192051307
-3.5515967276420715
4.766016868197438
3.964423274790182
-3.485018262834794
2.2989212989320396
-0.1982844045156987
0.8743248671279281
-1.5357009810003888
-3.080037190727684
0.4825682286788891
5.524183821789016
0.36718120639759944
-0.3466749624369517
1.068930907657916
-1.1151872206299538
1.8467834884228373
-1.3089714517889879
-1.8018740954398262
3.0077743082059323
-1.9784685568864642
0.816025666221435
4.014651519479192
1.4016353511220438
-1.79972272312442
0.503107418135773
0.7233013409669229
1.8212860817785352
1.960029421569026
1.4157556763193355
0.6041416352325443
-1.0975137291447632
0.789346695785869
-2.2831142666651334
-1.8550800954120399
4.064799556441831
```

El módulo `scipy.linalg` permite la creación de matrices especiales, tales como matrices diagonales de bloques `block_diag` , matrices circulantes `circulant` , matrices *companion* (`companion`), matrices de Hadamard (`hadamard`), Hankel (`hankel`), Hilbert (`hilbert`), Hilbert invertida (`invhilbert`), Leslie (`leslie`), Toeplitz (`toeplitz`) y matrices triangulares (`tri` , `tril` , `triu`).

Eigenvalores y eigenvectores

El cálculo de *eigenvectores* y *eigenvalores* es uno de los más complicados (y útiles) a realizarse en matrices cuadradas. **SciPy** posee varias rutinas para calcularlas:

- `eigvals`
- `eigvalsh`
- `eigvals_banded`

Y los respectivos métodos para *eigenvectores*: `eig` , `eigh` y `eigh_banded` .

Ejercicio: Calcule los _eigenvectores_ e _eigenvalores_ de las siguientes matrices usando los diferentes métodos. -

$$A = \begin{bmatrix} 4 & 6 & 4 \\ -2 & -3 & -4 \\ 0 & 0 & 2 \end{bmatrix}$$

-

$$B = \begin{bmatrix} 1 & 2 & 0 \\ 0 & 1 & 2 \\ 0 & 0 & 1 \end{bmatrix}$$

NOTA Si es posible, utilice los métodos de creación de matrices especiales.

Algebra lineal simbólica

Es posible manipular algebraicamente a matrices de expresiones simbólicas, usando la clase de `Matrix` de **SimPy** .

In [55]:

```
from ipywidgets import interact
from IPython.display import display
```

Cuando se trabaja con **Sympy** **no** se puede usar `%pylab inline` ya que `%pylab%` importa variables que entraran en conflicto con **Sympy**. Es mejor usar, `%matplotlib inline` e importar `numpy` y `matplotlib`.

In [56]:

```
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt
```

In [57]:

```
from sympy import *
```

In [58]:

```
init_printing(use_latex='mathjax')
```

In [59]:

```
x = Symbol('x')
y = Symbol('y')
```

In [60]:

```
A = Matrix([[1,x], [y,1]])
A
```

Out[60]:

$$\begin{bmatrix} 1 & x \\ y & 1 \end{bmatrix}$$

In [61]:

```
A[0,0]
```

Out[61]:

$$1$$

In [62]:

```
A[:,1]
```

Out[62]:

$$\begin{bmatrix} x \\ 1 \end{bmatrix}$$

In [63]:

```
A**2
```

Out[63]:

$$\begin{bmatrix} xy+1 & 2x \\ 2y & xy+1 \end{bmatrix}$$

In [64]:

```
A.inv()
```

Out[64]:

$$\begin{bmatrix} \frac{1}{-xy+1} & -\frac{x}{-xy+1} \\ -\frac{y}{-xy+1} & \frac{1}{-xy+1} \end{bmatrix}$$

In [65]:

```
I = A.inv()*A
I
```

Out[65]:

$$\begin{bmatrix} -\frac{xy}{-xy+1} + \frac{1}{-xy+1} & 0 \\ 0 & -\frac{xy}{-xy+1} + \frac{1}{-xy+1} \end{bmatrix}$$

In [66]:

```
I = simplify(I)
I
```

Out[66]:

$$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

Para matrices pequeñas, puedes calcular los *eigenvalores* simbólicamente.

In [67]:

```
A.eigenvals()
```

Out[67]:

$$\left\{1 - \sqrt{xy} : 1, \ \sqrt{xy} + 1 : 1\right\}$$

In [68]:

```
A.subs({x:0, y:1})
```

Out[68]:

$$\begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix}$$

Ejercicio: Cree matrices de 3×3 de *Hilbert*, *Leslie* y *Circulantes* y muéstrelas de manera simbólica.

Ejemplos

Procesamiento de imágenes

Vamos a representar las imágenes como matrices $\mathbf{R}^{n \times m \times k}$. Usaremos primero el método decomposición de matrices conocido como [*Single Value Decomposition* (http://en.wikipedia.org/wiki/Singular_value_decomposition)](http://en.wikipedia.org/wiki/Singular_value_decomposition) (**SVD**) para reducir el tamaño de la imagen.

La **SVD** de una matriz (real o compleja) $\mathbf{M}$ de $m \times n$ es una factorización de la forma $\mathbf{M} = U \cdot S \cdot V^*$, en la cual U es matriz $m \times m$ unitaria, S es una matriz $m \times n$ rectangular diagonal con elementos no-negativos, y V^* es la conjugada traspuesta de una matriz unitaria de $n \times n$.

A los elementos de la diagonal S_{ii} of S se les denomina valores singulares de $\mathbf{M}$. A las m columnas de U y a las n de V se les llama vectores singulares izquierdos o derechos, respectivamente.

Cuando $\mathbf{M}$ es cuadrada ($m \times m$) y real con determinante positivo, U, V^*, y S son matrices reales de $m \times m$, entonces S puede ser interpretada como una matriz de escalamiento, y U, V^* como matrices de rotación.

In [69]:

```
import scipy.misc
img = scipy.misc.face()
plt.imshow(img)
```

Out[69]:

```
<matplotlib.image.AxesImage at 0x7f61aa06be50>
```

In [70]:

```
img
```

Out[70]:

```
array([[[121, 112, 131],
        [138, 129, 148],
        [153, 144, 165],
        ...,
        [119, 126,  74],
        [131, 136,  82],
        [139, 144,  90]],

       [[ 89,  82, 100],
        [110, 103, 121],
        [130, 122, 143],
        ...,
        [118, 125,  71],
        [134, 141,  87],
        [146, 153,  99]],

       [[ 73,  66,  84],
        [ 94,  87, 105],
        [115, 108, 126],
        ...,
        [117, 126,  71],
        [133, 142,  87],
        [144, 153,  98]],

       ...,

       [[ 87, 106,  76],
        [ 94, 110,  81],
        [107, 124,  92],
        ...,
        [120, 158,  97],
        [119, 157,  96],
        [119, 158,  95]],

       [[ 85, 101,  72],
        [ 95, 111,  82],
        [112, 127,  96],
        ...,
        [121, 157,  96],
        [120, 156,  94],
        [120, 156,  94]],

       [[ 85, 101,  74],
        [ 97, 113,  84],
        [111, 126,  97],
        ...,
        [120, 156,  95],
        [119, 155,  93],
        [118, 154,  92]]], dtype=uint8)
```

In [72]:

```
shape(img)
#Es la resolucion de la imagen
```

Out[72]:

$(768, \ 1024, \ 3)$

In [73]:

```
U, S, Vs = scipy.linalg.svd(img[:,:,2])
print(U.shape)
print(S.shape)
print(Vs.shape)
```

```
(768, 768)
(768,)
(1024, 1024)
```

La matriz S está representada como una matriz *sparse*. Como queremos hacer una compresión de la imagen, sólo nos quedaremos con 32 de los *valores singulares*. Creamos una nueva matriz cuyos elementos están dados por la siguiente fórmula:

$$\Sigma_{j=1}^{k} \quad s_j(u_j \cdot v_j)$$

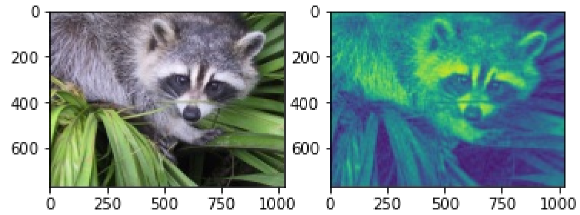
donde, s son los valores singulares, u y v son los vectores singulares.

In [74]:

```
A = numpy.dot( U[:, 0:32], numpy.dot(numpy.diag(S[0:32]), Vs[0:32,:]))
```

In [75]:

```
plt.subplot(121, aspect='equal'); plt.imshow(img);
#plt.gray()

plt.subplot(122, aspect='equal'); plt.imshow(A);
```



Autómatas Celulares

Un [autómata celular (http://en.wikipedia.org/wiki/Cellular_automaton)](http://en.wikipedia.org/wiki/Cellular_automaton) (**CA**) es un modelo del mundo con física simple. Se les conoce como *autómatas celulares* ya que el espacio está dividido en pedazos discretos, llamados *celdas* (de ahí "celular") y a que computa (i.e. es un "autómata").

Los **CA** están gobernados por *reglas* (la física) que determina como evoluciona el sistema en el tiempo. El tiempo también está dividido en pasos (*steps*) dicretos, y la regla especifica como cambia el estado actual del "mundo" en el tiempo $t + 1$ basado en el tiempo actual t.

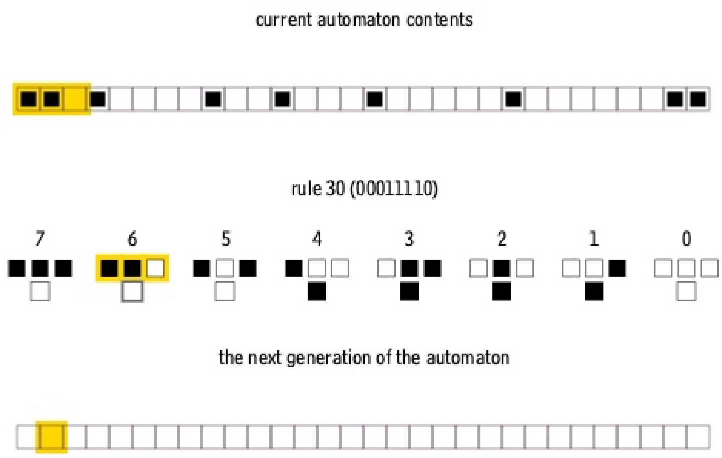
Los **CA** fueron estudiados extensivamente a principios de los 80s por **Stephen Wolfram** (Sí, el de `Mathematica`). En particular el estudio **CA**s unidimensionales, llamadas *autómatas celulares elementales*.

Un *autómata celular elemental* es un **CA** 1D en la cual cada celda tiene dos posibles estados, y en la cual la regla tiene como entradas el estado actual de la celda y el estado de sus vecinos inmediatos (que son dos en 1D). Existen, entonces $2^3 = 8$ posibles patrones de tres celdas (*vecindad*) y $2^8 = 256$ reglas posibles.

Por ejemplo:

Estado								
Actual	111	110	101	100	011	010	001	000
Siguiente	0	0	1	1	0	0	1	0

Wolfram sugirió nombrar las reglas usando el renglón inferior como binario. En el caso recién mostrado, es la **Regla 50**.



El siguiente código está basado en el trabajo de Allen B. Downey. Representa un autómata celular.

In [76]:

```python
import numpy as np

# Basado en el código de Allen B. Downey

#Genra un objeto

class CA(object):
    """Representa un autómata celular 1D.

    Los parámetros del constructor son:

    rule:  Un entero del 0-255.
    n:      Número de renglones (timesteps).
    ratio:  Razón de los renglones a las columnas
    """

    def __init__(self, rule, n=100, ratio=2):
        """
        Atributos:
        table:  Diccionario que mapea el estado, al siguiente.
        n, m:   Renglones, columnas.
        array:  Arreglo que contiene los datos.
        next:   Índice del siguiente estado.
        """
        self.table = self.make_table(rule)
        self.n = n
        self.m = ratio*n + 1
        self.array = np.zeros((n, self.m), dtype=np.int8)
        self.next = 0

    def make_table(self, rule):
        """Regresa la tabla con las reglas del CA
        (Implementada como un diccionario).
        """
        table = {}
        for i, bit in enumerate(binary(rule, 8)):
            t = binary(7-i, 3)
            table[t] = bit
        return table

    def start_single(self):
        """La semilla es una sola y aparece a la mitad del arreglo 1D."""
        self.array[0, int(self.m/2)] = 1
        self.next += 1

    def start_random(self):
        """Valores aleatorios en el tiempo t_0"""
        self.array[0] = np.random.random([1,self.m]).round()
        self.next += 1

    def loop(self, steps=1):
        """Ejecuta el número especificado de pasos."""
        [self.step() for i in range(steps)]

    def step(self):
        """Avanza un paso t -> t+1."""
        i = self.next
        self.next += 1

        a = self.array
        t = self.table
        for j in range(1,self.m-1):
            a[i,j] = t[tuple(a[i-1, j-1:j+2])]

    def get_array(self, start=0, end=None):
        """Obtiene una rebanada de las columnas del CA.
        """
        if start==0 and end==None:
            return self.array
        else:
            return self.array[:, start:end]
```

In [77]:

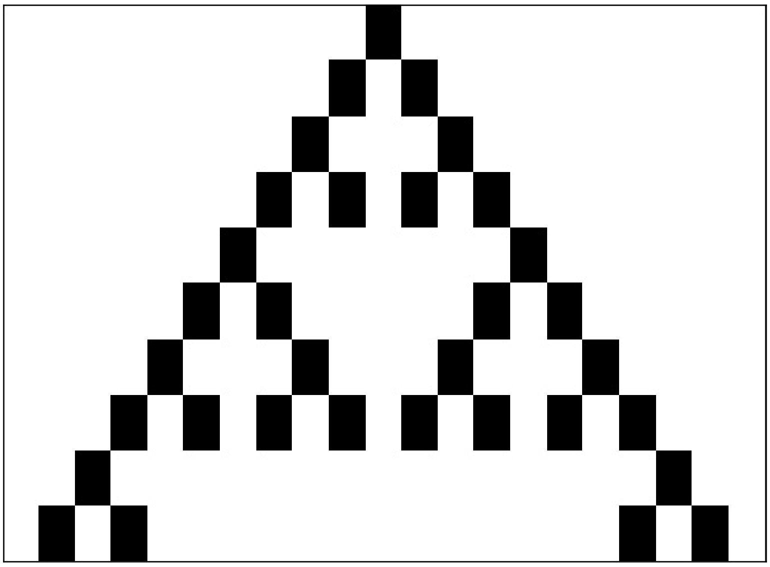
```
# Basado en el código de Allen B. Downey

def binary(n, digits):
    """Regresa una tupla de enteros representando (n) en binario."""
    t = []
    for i in range(digits):
        n, r = divmod(n, 2)
        t.append(r)

    return tuple(reversed(t))
```

In [78]:

```
import numpy

# Basado en el código de Allen B. Downey

class CADrawer(object):
    """Dibuja el CA usando matplotlib"""

    def __init__(self):
        # we only need to import pyplot if a PyplotDrawer
        # gets instantiated
        global pyplot
        import matplotlib.pyplot as pyplot

    def draw(self, ca, start=0, end=None):
        pyplot.figure(figsize=(8, 6), dpi=80)
        pyplot.gray()
        a = ca.get_array(start, end)
        rows, cols = a.shape

        # flipud puts the first row at the top;
        # negating it makes the non-zero cells black.
        pyplot.pcolor(-numpy.flipud(a))
        pyplot.axis([0, cols, 0, rows])

        # empty lists draw no ticks
        pyplot.xticks([])
        pyplot.yticks([])

    def show(self):
        """display the pseudocolor representation of the CA"""
        pyplot.show()

    def save(self, filename='ca.png'):
        """save the pseudocolor representation of the CA in (filename)."""
        pyplot.savefig(filename)
```

In [79]:

```
rule = 154
n = 10
ca = CA(rule, n)
ca.start_single()
ca.loop(n-1)
drawer = CADrawer()
drawer.draw(ca)
drawer.show()
```



Ejercicio: Escribe un método que genere las 255 reglas y las muestre en una gráfica (con _subplots_, obviamente).

Ejercicio: La página de la [wikipedia](http://en.wikipedia.org/wiki/Cellular_automaton#Classification) menciona 4 clasificaciones ¿Puedes identificarlos en tu gráfica?

Ejercicio: Escribe una animación interactiva en la cual, reciba la regla, el intervalo del tiempo y con eso la vaya dibujando de manera animada.

Ejemplo de solución

In [80]:

```
import numpy

def getEvolvedCA(rule, n = 30):
    ca = CA(rule, n)
    ca.start_single()
    ca.loop(n-1)
    return ca.get_array()

class MosaicCADrawer(object):

    def __init__(self, time_steps):
        self.rows = 26
        self.cols = 10
        self.time_steps = time_steps


    def draw(self):
        fig, ax = plt.subplots(self.rows,self.cols, figsize=(100,80), sharey=True)
        for row in range(self.rows):
            for col in range(self.cols):
                rule = row*self.cols + col
                if rule <= 255:
                    ca_universe = getEvolvedCA(rule, self.time_steps)

                    ax[row,col].imshow(-ca_universe)
                    ax[row,col].set_xlabel(rule)

                ax[row,col].set_xticks([])
                ax[row,col].set_yticks([])
```

In [153]:

```
ca_drawer = MosaicCADrawer(10)
ca_drawer.draw()
```