# numpy, linear algebra, vectorization

1. NumPy and Linear Algebra
   - arrays and matrices
   - linear algebra
2. Vectorizations
   - using `numpy.vectorize`
   - using `numpy.where`
3. Particle Movements
   - basic version of the simulation
   - vectorized implementation
   - the game of life of John Conway

MCS 507 Lecture 4
Mathematical, Statistical and Scientific Software
Jan Verschelde, 28 August 2023

## numpy, linear algebra, vectorization

## Lists and Arrays

Lists are versatile data structures in Python:

- have variable length, and
- heterogeneous, its item may not be of same type.

Arrays are

- sequences of fixed length, and
- filled with objects of the same type.

Compared to lists, arrays are

- more memory efficient, and
- allow for faster access.

Python has limited support for arrays in the module `array`, but does not support matrices or multi-dimensional arrays, and does not provide any linear algebra operations.

## NumPy

From `www.numpy.org`: *NumPy is the fundamental package for scientific computing with Python*.

Some literature about NumPy:

- Travis E. Oliphant: **Guide to Numpy.** 2nd edition, 7 Dec 2006.
  `http://web.mit.edu/dvp/Public/numpybook.pdf`
  371 pages
- Eli Bressert: **SciPy and NumPy.** O'Reilly, 2013. 57 pages
- Robert Johansson. **Numerical Python: Scientific Computing and Data Science Applications with Numpy, SciPy and Matplotlib.** 2nd ed. Edition, Apress, 2018. 700 pages

# numpy, linear algebra, vectorization

## eigenvalues and eigenvectors

In an interactive Python session:

```
>>> import numpy as np
>>> A = np.random.randint(0, 10, (2, 2))
>>> A
array([[8, 8],
       [3, 6]])
>>> [L, V] = np.linalg.eig(A)
>>> L
array([12.,  2.])
>>> V
array([[ 0.89442719, -0.8       ],
       [ 0.4472136 ,  0.6       ]])
```

## verifying the first eigenvalue and eigenvector

To check $A\mathbf{v} = \lambda\mathbf{v}$, we need `matrix` types:

```
>>> M = np.matrix(A)
>>> v1 = V[:, 0]
>>> v1
array([0.89442719, 0.4472136 ])
>>> v1t = np.matrix(v1).transpose()
>>> v1t
matrix([[0.89442719],
        [0.4472136 ]])
>>> M*v1t
matrix([[10.73312629],
        [ 5.36656315]])
>>> L[0]*v1t
matrix([[10.73312629],
        [ 5.36656315]])
```

## matrix decompositions

```
>>> W = np.matrix(V)
>>> W
matrix([[ 0.89442719, -0.8       ],
        [ 0.4472136 ,  0.6       ]])
>>> K = np.diag(L)
>>> K
array([[12.,  0.],
       [ 0.,  2.]])
>>> M*W
matrix([[10.73312629, -1.6       ],
        [ 5.36656315,  1.2       ]])
>>> W*K
matrix([[10.73312629, -1.6       ],
        [ 5.36656315,  1.2       ]])
```

## a spectral decomposition of $A$

If eigenvectors of $A$ are in the columns of $V$ and the eigenvalues on the diagonal of $\Lambda$, then $AV = V\Lambda$ or $A = V\Lambda V^{-1}$.

```
M*W == W*K  ⇒  M == W*K*W**(-1)
```

```
>>> M
matrix([[8, 8],
        [3, 6]])
>>> W*K*W**(-1)
matrix([[8., 8.],
        [3., 6.]])
```

## other decompositions

Three other decompositions of matrices:

1. LU factorization, L is lower, U is upper triangular
   `scipy.linalg.lu`
2. QR factorization, Q is orthogonal, R is upper triangular
   `numpy.linalg.qr` or `scipy.linalg.qr`

   We can use both in the same session:

   ```
   >>> from numpy.linalg import qr as npqr
   >>> from scipy.linalg import qr as spqr
   ```

   With `npqr` we then use `numpy.linalg.qr`,
   with `spqr` the `scipy.linalg.qr` is called.
3. SVD, or the singular value decomposition
   `numpy.linalg.svd` or `scipy.linalg.svd`

## a first exercise

Exercise 1:

1. Generate random 3-by-3 integer matrix $A$ with numbers in $[0, 9]$.
   Let $x$ be a vector three ones and set $b = Ax$,
   so $Ax = b$ has as solution vector a vector of ones.

   1. Compute a LU factorization of A and
      recover the solution $x$ from the factors $L$ and $U$.
   2. Compute a QR factorization of A and
      recover the solution $x$ for the factors $Q$ and $R$.
   3. Compute a SVD of A and
      recover the solution $x$ for the outcome of the SVD.

## numpy, linear algebra, vectorization

## default vectorizations

By default, functions defined with numpy
accept vectors as arguments:

```
>>> from numpy import exp, sin, linspace
>>> f = lambda x: exp(-x**2)*sin(x)
>>> a = linspace(0,1,1000)
>>> b = f(a)
>>> print b[10:12]
[ 0.01000884  0.01100945]
```

Functions that take vectors as arguments are slow when applied to scalar arguments, because the loop runs in Python.
Vectorized versions apply optimized array index arithmetic.

## with `math.exp` and `math.sin`

```
>>> import math
>>> g = lambda x: math.exp(-x**2)*math.sin(x)
>>> a = linspace(0, 1, 1000)
>>> b = g(a)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 1, in <lambda>
TypeError: only length-1 arrays can be converted to Python
>>> for i in range(1000): b[i] = g(a[i])
...
>>>
```

The `for i in range(1000)` will be much slower compared to the vectorized version.

## numpy.vectorize

```
>>> from numpy import vectorize
>>> vg = vectorize(g)
>>> c = vg(a)
>>> type(vg)
<class 'numpy.lib.function_base.vectorize'>
>>> print vg(a[10:12])
[ 0.01000884  0.01100945]
```

## an exercise

Exercise 2:

2. Evaluate $x^2 + 2x - 1 = (x+2)x - 1$ at `a`,
   where `a = numpy.linspace(0,1,1000)`
   with inplace arithmetic defined by

   ```
   y = a; y += 2; y *= a; y -= 1.
   ```

   Wrap the inplace arithmetic sequence for $x^2 + 2x - 1$ in a function and compare the execution time to the vectorized version of $(x+2)x - 1$.

# numpy, linear algebra, vectorization

1. NumPy and Linear Algebra
   - arrays and matrices
   - linear algebra
2. Vectorizations
   - using `numpy.vectorize`
   - using `numpy.where`
3. Particle Movements
   - basic version of the simulation
   - vectorized implementation
   - the game of life of John Conway

## function with branchings

Recall the inline if-else statement:

```
>>> import math
>>> logpos = lambda x: \
(0.0 if x <= 0 else math.log(x))
>>> logpos(-1)
0.0
>>> logpos(10)
2.302585092994046
```

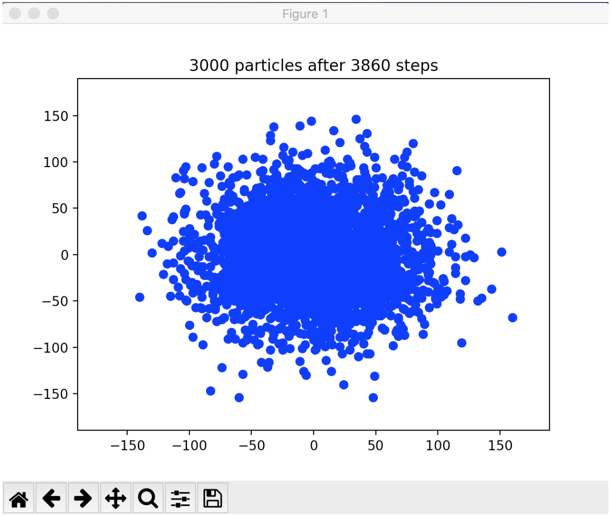
We would like to vectorize the if-else.

## using where

```
>>> vlogpos = lambda x: \
np.where(x > 0, np.log(x), 0.0)
>>> a = np.linspace(-1,1,100)
>>> b = vlogpos(a)
>>> print b[49:51]
[ 0.          -4.59511985]
```

# numpy, linear algebra, vectorization

## particle movements



## designing the simulation

All particles originate at $(0, 0)$.

For every particle, for every time step do:

1. generate a random integer $d \in \{1, 2, 3, 4\}$,
2. move according to the value of $d$:
    1. if $d = 1$: move particle north
    2. if $d = 2$: move particle south
    3. if $d = 3$: move particle east
    4. if $d = 4$: move particle west

## the script `walk.py`

```
import numpy as np
import random
import matplotlib
import matplotlib.pyplot as plt

def particles(npa, nst, pls):
    """
    Shows random particle movement with
    npa : number of particles,
    nst : number of time steps,
    pls : how many steps for next plot.
    """
```

## every particle starts at $(0, 0)$

```
xpa = np.zeros(npa)
ypa = np.zeros(npa)
xymax = 3*np.sqrt(nst)
xymin = -xymax
plt.ion()
fig = plt.figure()
ax = fig.add_subplot(111)
ax.set_xlim(xymin, xymax)
ax.set_ylim(xymin, xymax)
dots, = ax.plot(x, y,'bo')
strtitle = 'animating %d particles' % npa
ax.set_title(strtitle)
fig.canvas.draw()
plt.pause(0.00001)
```

## moving at random in the main loop

For as many steps as the value of `nst`,
and for every particle, we roll a four sided die
to determine whether to move north, south, east, or west.

```
for step in range(nst):
    for i in range(npa):
        die = random.randint(1, 4)
        if die == 1:
            ypa[i] += 1 # north
        elif die == 2:
            ypa[i] -= 1 # south
        elif die == 3:
            xpa[i] += 1 # east
        elif die == 4:
            xpa[i] -= 1 # west
```

## plotting and the main function

```
        if(step+1) % pls == 0:
            dots.set_xdata(x); dots.set_ydata(y)
            strtitle = '%d particles after %d steps' \
                % (npa, step+1)
            ax.set_title(strtitle)
            fig.canvas.draw()
            plt.pause(0.00001)

def main():
    """
    Fixes the seed for the random numbers
    and starts the particle simulation.
    """
    random.seed(10)
    particles(3000, 4000, 20)
```

$\rightarrow$ will run a simulation of 3000 particles
over 4000 stages plotted every 20 time steps.

## numpy, linear algebra, vectorization

## vectorization

Vectorization: replace the Python `for` loops
by NumPy operations on arrays.

To speed up the simulation:

1. generate all random directions at once,
2. use `where` to update coordinates.

The built-in function `where` has the syntax

```
numpy.where(condition, [x, y])
```

and returns elements either from `x` or `y` (optional)
depending on `condition`.

## generating all moves

In the code listing below, the initial plot is omitted.

```
def particles(npa, nst, pls):
    """
    Shows random particle movement with
    npa : number of particles,
    nst : number of time steps,
    pls : how many steps for next plot.
    """
    xpa = np.zeros(npa)
    ypa = np.zeros(npa)
    xymax = 3*np.sqrt(nst)
    xymin = -xymax
    moves = np.random.randint(1, 5, nst*npa)
    moves.shape = (nst, npa)
```

## script `walkvector.py` continued

```
for step in range(nst):
    this_move = moves[step, :]
    ypa += np.where(this_move == 1, 1, 0)
    ypa -= np.where(this_move == 2, 1, 0)
    xpa += np.where(this_move == 3, 1, 0)
    xpa -= np.where(this_move == 4, 1, 0)
    if(step+1) % pls == 0:
        dots.set_xdata(xpa)
        dots.set_ydata(ypa)
        strtitle = '%d particles after %d steps' \
            % (npa, step+1)
        ax.set_title(strtitle)
        fig.canvas.draw()
        plt.pause(0.00001)
```

## running on a MacBook Pro

```
$ time python3 walk.py

real    0m30.829s
user    0m30.037s
sys     0m0.577s

$ time python3 walkvector.py

real    0m11.154s
user    0m10.319s
sys     0m0.621s
$
```

The vectorized version is almost three times faster.

## numpy, linear algebra, vectorization

## simulating cellular growth

The game of life is a discovery of John Conway.

Consider a rectangular grid of cells with rules:

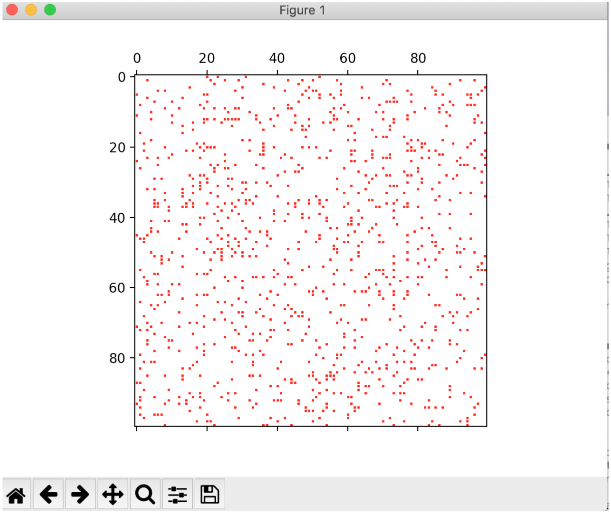
1. An empty cell is born when it has 3 neighbors.
2. A living cell can either die or survive, as follows:
    1. die by loneliness, if the cell has one or no neighbors;
    2. die by overpopulation, if the cell has $\geq$ 4 neighbors;
    3. survive, if the cell has two or three neighbors.

## design of the code

Three ingredients:

1. The rectangular grid is represented by a NumPy matrix $A$
   - of integers: $A_{i,j} \in \{0, 1\}$,
   - $A_{i,j} = 0$: cell $(i, j)$ is dead,
   - $A_{i,j} = 1$: cell $(i, j)$ is alive.
2. We update the matrix applying the rules, running over all pairs of indices $(i, j)$.
3. The grid can be plotted with the `spy` method.

# visualizing a matrix with a spy plot



## the script to make a spy plot

```
import numpy as np
from scipy import sparse
import matplotlib.pyplot as plt

R = 0.1 # ratio of nonzeroes
N = 100 # dimension of the matrix
A = np.random.rand(N, N)
A = np.matrix(A < R, int)
print(A)
S = sparse.coo_matrix(A)
print('number of nonzeros :', S.nnz)
fig = plt.figure()
ax = fig.add_subplot(111)
ax.spy(A, markersize=1, color='red')
plt.show()
```

## code for the game of life

Exercise 3:

2. Write a Python script to visualize the game of life, by a simple application of the rules of the game.

Exercise 4:

3. To vectorize the script for the game of life, write the rules of the game with matrix operations. Observe that the count of live neighbors can happen by adding a matrix with one column shifted. Compare the running time of the vectorized game with your original formulation of the previous exercise.