Linear Algebra in Python

Contents

Motivation

Linear algebra is of vital importance in almost any area of science and engineering and therefore numerical linear algebra is just as important in computational science. Some of the more popular areas of application include machine learning, computer vision, finite element method, optimisation, and many others.

Computers use a discrete representation of the real numbers, rather than a continuous one, which has several consequences. Most notably, it means that there have to be gaps between adjacent numbers. We will therefore most often want to work with floating point numbers with double precision (*float* in python) which allow us to represent real numbers with very high precision. However, often in computing we deal with extremely large datasets (e.g. matrices with millions of rows and columns) so even tiny errors can grow and accumulate very fast if our algorithm is *numerically unstable*.

For example, we might be tempted to solve $A\mathbf{x} = \mathbf{b}$ by calculating an inverse of A, but

calculating inverses and determinants of matrices is generally avoided in computing due to round-off errors. Numerical linear algebra therefore aims to come up with fast and efficient algorithms to solve usual linear algebra problems without magnifying these (and other) small errors.

Python

The main library for linear algebra in Python is SciPy which makes use of NumPy arrays. NumPy also provides plenty of basic functionalities through its functions in `numpy.linalg`, but many advanced capabilities remain reserved for `scipy.linalg`.

```
import numpy as np
import numpy.linalg as la
```

Constructing some useful matrices

Identity matrix

NumPy provides us with two almost identical functions to construct identity matrices: `numpy.identity(n)` and `numpy.eye(n, [m=n], [k=0])`. The former constructs a square matrix with n rows and columns with 1s on the main diagonal. The latter constructs an $n \times m$ matrix with 1s on the kth diagonal (k=0 is the main diagonal). Examples below demonstrate this:

```
print('identity(5) = \n', np.identity(5))  # square 5x5 matrix
print('\neye(4, 5) = \n', np.eye(4, 5))  # 4x5 matrix
print('\neye(4, 5, -1) = \n', np.eye(4, 5, -1))  # 1st lower diagonal
print('\neye(4, 5, 2) = \n', np.eye(4, 5, 2))  # 2nd upper diagonal
```

```
identity(5) =
 [[1. 0. 0. 0. 0.]
 [0. 1. 0. 0. 0.]
 [0. 0. 1. 0. 0.]
 [0. 0. 0. 1. 0.]
 [0. 0. 0. 0. 1.]]

eye(4, 5) =
 [[1. 0. 0. 0. 0.]
 [0. 1. 0. 0. 0.]
 [0. 0. 1. 0. 0.]
 [0. 0. 0. 1. 0.]]

eye(4, 5, -1) =
 [[0. 0. 0. 0. 0.]
 [1. 0. 0. 0. 0.]
 [0. 1. 0. 0. 0.]
 [0. 0. 1. 0. 0.]]

eye(4, 5, 2) =
 [[0. 0. 1. 0. 0.]
 [0. 0. 0. 1. 0.]
 [0. 0. 0. 0. 1.]
 [0. 0. 0. 0. 0.]]
```

Diagonal matrix

The function `numpy.diag(array, [k=0])` either extracts a diagonal from an array or constructs a diagonal array:

1. if the input array is 2-D, returns a 1-D array with diagonal entries
2. if the input array is 1-D, returns a diagonal 2-D array with entries from the input array on the diagonal.

```
M = np.array([[1, 2],
              [3, 4]])
v = np.array([6, 8])

print('diag(M) = ', np.diag(M))
print('diag(v) = \n', np.diag(v))
```

```
diag(M) =  [1 4]
diag(v) =
 [[6 0]
 [0 8]]
```

Example: tridiagonal matrix

Let us show how we could use `numpy.diag` to construct a tridiagonal matrix, which is a banded matrix with non-zero entries on the main diagonal and the first off-diagonals above and below the main diagonal.

```
a = [1, 1, 1]
b = [2, 2, 2, 2]
c = [3, 3, 3]

A = np.diag(a, -1) + np.diag(b, 0) + np.diag(c, 1)

print(A)
```

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

Triangular matrix

To construct upper or lower triangular matrices we use `numpy.triu(array, [k=0])` or `numpy.tril(array, [k=0])` functions (u stands for upper, l stands for lower). Returns an array whose entries below or above the kth diagonal are 0 (k=0 is the main diagonal).

```
M = np.arange(1, 13).reshape(4, 3)
print('M = \n', M)

print('\ntriu(M) = \n', np.triu(M))
print('\ntriu(M, -1) = \n', np.triu(M, -1))
print('\ntril(M, 1) = \n', np.tril(M, 1))
```

```
M =
 [[ 1  2  3]
 [ 4  5  6]
 [ 7  8  9]
 [10 11 12]]

triu(M) =
 [[1 2 3]
 [0 5 6]
 [0 0 9]
 [0 0 0]]

triu(M, -1) =
 [[ 1  2  3]
 [ 4  5  6]
 [ 0  8  9]
 [ 0  0 12]]

tril(M, 1) =
 [[ 1  2  0]
 [ 4  5  6]
 [ 7  8  9]
 [10 11 12]]
```

Sparse matrices

A sparse matrix is a matrix with mostly zero-valued entries. Performing operations on sparse matrices can therefore be unnecessarily computationally expensive, since we are unnecessarily adding 0s or multiplying by 0, for example. Therefore, when working with sparse matrices we would like to have an option to simply skip this kind of operations.

The identity matrix and diagonal matrices are examples of such matrices, although a sparse matrix does not need to have any particular structure. The non-zero entries could be completely random, and this is most often the case in practice.

SciPy allows us to build such matrices and do operations on them with the `scipy.sparse` package. There are several formats how these matrices are stored and users are encouraged to read the documentation and the Wikipedia page for an explanation of them. For example, a sparse matrix in coordinate format is stored in three arrays: one for the values of non-zero entries and two for the row and column index of those entries.

Examples: Convert a NumPy array

We can convert any array to a sparse matrix. For example, we can use the function `scipy.sparse.coo_matrix` to construct a matrix in COOrdinate format. Let us convert a tridiagonal NumPy matrix to a sparse SciPy matrix.

```
from scipy.sparse import coo_matrix


a, b, c = [1] * 9, [2] * 10, [3] * 9

A = np.diag(a, -1) + np.diag(b, 0) + np.diag(c, 1)
print(A)

spA = coo_matrix(A)
print(spA)
```

```
[[2 3 0 0 0 0 0 0 0 0]
 [1 2 3 0 0 0 0 0 0 0]
 [0 1 2 3 0 0 0 0 0 0]
 [0 0 1 2 3 0 0 0 0 0]
 [0 0 0 1 2 3 0 0 0 0]
 [0 0 0 0 1 2 3 0 0 0]
 [0 0 0 0 0 1 2 3 0 0]
 [0 0 0 0 0 0 1 2 3 0]
 [0 0 0 0 0 0 0 1 2 3]
 [0 0 0 0 0 0 0 0 1 2]]
  (0, 0)        2
  (0, 1)        3
  (1, 0)        1
  (1, 1)        2
  (1, 2)        3
  (2, 1)        1
  (2, 2)        2
  (2, 3)        3
  (3, 2)        1
  (3, 3)        2
  (3, 4)        3
  (4, 3)        1
  (4, 4)        2
  (4, 5)        3
  (5, 4)        1
  (5, 5)        2
  (5, 6)        3
  (6, 5)        1
  (6, 6)        2
  (6, 7)        3
  (7, 6)        1
  (7, 7)        2
  (7, 8)        3
  (8, 7)        1
  (8, 8)        2
  (8, 9)        3
  (9, 8)        1
  (9, 9)        2
```

In the above example, A has 100 entries, while spA ('sparse A') has only 28. In this case we would not save much computational time since a 10×10 matrix is quite small. However, sparse matrices become more useful (and even essential) the larger our matrix is.

Examples: Construct a tridiagonal matrix

Users more comfortable with SciPy may wish to directly construct sparse matrices, instead of converting a NumPy array. Let us construct the same tridiagonal matrix as in the previous

example.

```
from scipy.sparse import diags


diagonals = [[1] * 9, [2] * 10, [3] * 9]

A = diags(diagonals, [-1, 0, 1], format='coo')
print(A.toarray())  # print the entire array
print(A)
```

```
[[2. 3. 0. 0. 0. 0. 0. 0. 0. 0.]
 [1. 2. 3. 0. 0. 0. 0. 0. 0. 0.]
 [0. 1. 2. 3. 0. 0. 0. 0. 0. 0.]
 [0. 0. 1. 2. 3. 0. 0. 0. 0. 0.]
 [0. 0. 0. 1. 2. 3. 0. 0. 0. 0.]
 [0. 0. 0. 0. 1. 2. 3. 0. 0. 0.]
 [0. 0. 0. 0. 0. 1. 2. 3. 0. 0.]
 [0. 0. 0. 0. 0. 0. 1. 2. 3. 0.]
 [0. 0. 0. 0. 0. 0. 0. 1. 2. 3.]
 [0. 0. 0. 0. 0. 0. 0. 0. 1. 2.]]
  (1, 0)        1.0
  (2, 1)        1.0
  (3, 2)        1.0
  (4, 3)        1.0
  (5, 4)        1.0
  (6, 5)        1.0
  (7, 6)        1.0
  (8, 7)        1.0
  (9, 8)        1.0
  (0, 0)        2.0
  (1, 1)        2.0
  (2, 2)        2.0
  (3, 3)        2.0
  (4, 4)        2.0
  (5, 5)        2.0
  (6, 6)        2.0
  (7, 7)        2.0
  (8, 8)        2.0
  (9, 9)        2.0
  (0, 1)        3.0
  (1, 2)        3.0
  (2, 3)        3.0
  (3, 4)        3.0
  (4, 5)        3.0
  (5, 6)        3.0
  (6, 7)        3.0
  (7, 8)        3.0
  (8, 9)        3.0
```

Matrix operations

Arithmetic operations

All arithmetic operators (`+`, `-`, `*`, `/`, `%`, `**`, `//`) on arrays apply elementwise. If both operands are matrices they need to have the same dimensions (to be conformable for the operation).

```
M = np.array([[1, 2],
              [3, 4]])

print('M =\n', M)
print('M + M =\n', M + M)  # add
print('M - M =\n', M - M)  # subtract
print('4 * M =\n', 4 * M)  # multiply by a scalar
print('M * M =\n', M * M)  # multiply matrices elementwise
print('M / M =\n', M / M)  # divide elementwise
print('M ** 3 =\n', M ** 3)  # exponentiation elementwise (M**3 = M*M*M)
print('M % 2 =\n', M % 2)  # modulus
print('M // 2 =\n', M // 2)  # integer (floor) division
```

```
M =
 [[1 2]
 [3 4]]
M + M =
 [[2 4]
 [6 8]]
M - M =
 [[0 0]
 [0 0]]
4 * M =
 [[ 4  8]
 [12 16]]
M * M =
 [[ 1  4]
 [ 9 16]]
M / M =
 [[1. 1.]
 [1. 1.]]
M ** 3 =
 [[ 1  8]
 [27 64]]
M % 2 =
 [[1 0]
 [1 0]]
M // 2 =
 [[0 1]
 [1 2]]
```

Matrix multiplication

NumPy uses the function `numpy.matmul(array1, array2)` to multiply two matrices. Users of Python 3.5 or newer can use the operator `@` instead.

To raise a matrix to some power we need to use the function `numpy.linalg.matrix_power(array, exp)`.

Note: When one of the operands is a 1-D array (we might initialise a vector this way if we are not careful - see below on reshaping arrays), it will automatically be assumed that it is a row vector if it is left-multiplying or a column vector if it is right-multiplying. The result will be a 1-D array as well.

```
print('M @ M =\n', M @ M)
print('M @ M @ M =\n', M @ M @ M)
print('M^3 = \n', la.matrix_power(M, 3))

x = np.array([1, 2])  # 1-D array
print('x = ', x)

print('x @ M = \n', x @ M)
print('M @ x = \n', M @ x)
```

```
M @ M =
 [[ 7 10]
 [15 22]]
M @ M @ M =
 [[ 37  54]
 [ 81 118]]
M^3 =
 [[ 37  54]
 [ 81 118]]
x =  [1 2]
x @ M =
 [ 7 10]
M @ x =
 [ 5 11]
```

Inner product

The function `numpy.dot(array1, array2)` is an alternative matrix product function but we recommend using it only for calculating an inner product of two vectors. Curious readers are encouraged to read the documentation for other uses, although `@` is preferred.

```
x = np.array([1, 2])
y = np.array([3, 4])

print(np.dot(x, y))
```

```
11
```

Transpose

A matrix can be transposed using `numpy.transpose(array)` or with the array attribute `array.T`.

```
print('M.T = \n', M.T)
print('numpy.transpose(M) = \n', np.transpose(M))
```

```
M.T =
 [[1 3]
 [2 4]]
numpy.transpose(M) =
 [[1 3]
 [2 4]]
```

Reshaping arrays

Note that 1-D NumPy arrays remain unchanged when transposed. To avoid this kind of issues, column (or row) vectors should be initialised as 2-D $n \times 1$ (or $1 \times n$) arrays. Below demonstrates this, as well as how to reshape 1-D arrays to 2-D. A very useful method of doing this is by giving -1 as one of the new shape parameters. We cannot give -1 for more than one shape parameter, as this is the unknown value which NumPy will figure out for us. For example, if we start from a 1-D array of 4 entries, reshaping it to (-1, 1, 2) will generate an array of shape $k \times 1 \times 2$ where $k \in \mathbb{N}$ has to satisfy $k \cdot 1 \cdot 2 = 4$ so that the total number of entries (A.size in NumPy) is preserved. If this k does not exist, the reshaping is not possible.

```
import numpy as np

x = np.array([1., 2., 3.])
print('x =', x, ', number of dimensions: ', x.ndim, ', shape:', x.shape)
print('x.T =', x.T)

x = np.array([[1., 2., 3.]])
print('x =', x, ', number of dimensions: ', x.ndim, ', shape:', x.shape)
print('x.T = \n', x.T)

x = np.array([1., 2., 3.]).reshape(-1, 1)  # reshape
print('[1., 2., 3.].reshape(-1, 1) = \n', x, ', number of dimensions: ', x.ndim, ',
```

```
x = [1. 2. 3.] , number of dimensions:  1 , shape: (3,)
x.T = [1. 2. 3.]
x = [[1. 2. 3.]] , number of dimensions:  2 , shape: (1, 3)
x.T =
 [[1.]
 [2.]
 [3.]]
[1., 2., 3.].reshape(-1, 1) =
 [[1.]
 [2.]
 [3.]] , number of dimensions:  2 , shape: (3, 1)
```

Complex conjugate

We use `numpy.conjugate(array)` or its alias `numpy.conj(array)` to find the complex conjugate of an array containing complex entries. Alternatively we can use the `.conj()` attribute of NumPy arrays.

To get the conjugate transpose (or Hermitian transpose) we would therefore write `array.T.conj()`.

```
C = np.array([[1 + 1j, 2 + 2j],
              [3 + 3j, 4 + 4j]])

print('C = \n', C)
print('conj(C) = \n', np.conj(C))
print('C.conj() = \n', C.conj())

print('\nC.T.conj() = \n', C.T.conj())
```

```
C =
 [[1.+1.j 2.+2.j]
 [3.+3.j 4.+4.j]]
conj(C) =
 [[1.-1.j 2.-2.j]
 [3.-3.j 4.-4.j]]
C.conj() =
 [[1.-1.j 2.-2.j]
 [3.-3.j 4.-4.j]]

C.T.conj() =
 [[1.-1.j 3.-3.j]
 [2.-2.j 4.-4.j]]
```

Norms

The function `numpy.linalg.norm(array, [order])` returns a matrix or vector norm of specified order. If the order is not specified, it returns the Frobenius matrix norm (or 2-norm for vectors).

```
x = np.array([1, 2, 3])
M = np.array([[1, 2],
              [3, 4]])

print('x = ', x)
print('M = \n', M)

print('\nnorm(x, 1) = ',la.norm(x, 1))
print('norm(x) = ', la.norm(x))
print('norm(x, np.inf)', la.norm(x, np.inf))

print('\nnorm(M)', la.norm(M))
print('norm(M, np.inf)', la.norm(M, np.inf))
```

```
x =  [1 2 3]
M =
 [[1 2]
 [3 4]]

norm(x, 1) =  6.0
norm(x) =  3.7416573867739413
norm(x, np.inf) 3.0

norm(M) 5.477225575051661
norm(M, np.inf) 7.0
```

Matrix inverse

We can find the inverse of a square matrix with the function `numpy.linalg.inv(array)`. As briefly mentioned in the motivation section, an inverse of large matrix is very likely to be imprecise due to round-off errors. However, we might be able to get away with it with certain smaller matrices.

```
print('inverse(M) = \n', la.inv(M))
```

```
inverse(M) =
 [[-2.   1. ]
 [ 1.5 -0.5]]
```

Solving systems of linear equations

We will most often want to solve a linear matrix equation $A\mathbf{x} = \mathbf{b}$.

As briefly mentioned in the motivation section, we should not solve this by finding an inverse A^{-1} and then multiplying $A^{-1}\mathbf{b}$ to find $\mathbf{x}$.

NumPy provides us with a function `numpy.linalg.solve(A, b)` which will solve the equation for $\mathbf{x}$ but it does it in a different way (see LU decomposition at the end of the notebook). It is required that A is not singular. For singular matrices we need to use the least-squares method.

```
A = np.array([[3., 5., -1.],
              [1., 4., 1.],
              [9., 0., 2.]])
b = np.array([10, 7, 1])

x = np.linalg.solve(A, b)
print(x)
```

```
[ 0.2  1.8 -0.4]
```

Example: Runge-Kutta, Wave equation

Consider the 1-D wave equation

$$u_{tt} + u_{xx} = 0, \qquad u(0) = u(L) = 0,$$

in the domain [0, L]. To solve it, let us write a standard explicit second order Runge-Kutta time-stepping algorithm. First, we need to discretise the equation with a finite difference approximation,

$$\frac{\partial^2}{\partial t^2} u_i - \frac{u_{i-1} - 2u_i + u_{i+1}}{\Delta x^2} = 0, \qquad u_0 = u_{N+1} = 0,$$

where $\Delta x = L/(N+1)$ and $u_i = u(x_i) = u(i\Delta x)$, $i = 1, \ldots, N$. Writing $\mathbf{v} = (u_1, u_2, \ldots, u_N)^T$, we can express this equation as:

$$\frac{\partial^2}{\partial t^2} \mathbf{v} + A\mathbf{v} = 0,$$

where A is an $N \times N$ tridiagonal coefficient matrix:

$$A = \frac{1}{\Delta x^2} \begin{bmatrix} 2 & -1 & 0 & \cdots & 0 & 0 \\ -1 & 2 & -1 & \cdots & 0 & 0 \\ 0 & -1 & 2 & \cdots & 0 & 0 \\ & \vdots & & & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & 2 & -1 \\ 0 & 0 & 0 & \cdots & -1 & 2 \end{bmatrix}.$$

Now we begin to see how linear algebra can be used to solve our PDE. We want to solve an ODE, so we introduce $\mathbf{w} = dv/dt$

u(x, 0) = {\rm e}^{-x^2}, u_t(x, 0) = 0

```
from scipy.sparse import diags

n = 10
vals = np.array([-np.ones(n-1), 2*np.ones(n), -np.ones(n-1)]) #/ dx**2
A = diags(vals, [-1, 0, 1])
print(A.toarray())
```

```
[[ 2. -1.  0.  0.  0.  0.  0.  0.  0.  0.]
 [-1.  2. -1.  0.  0.  0.  0.  0.  0.  0.]
 [ 0. -1.  2. -1.  0.  0.  0.  0.  0.  0.]
 [ 0.  0. -1.  2. -1.  0.  0.  0.  0.  0.]
 [ 0.  0.  0. -1.  2. -1.  0.  0.  0.  0.]
 [ 0.  0.  0.  0. -1.  2. -1.  0.  0.  0.]
 [ 0.  0.  0.  0.  0. -1.  2. -1.  0.  0.]
 [ 0.  0.  0.  0.  0.  0. -1.  2. -1.  0.]
 [ 0.  0.  0.  0.  0.  0.  0. -1.  2. -1.]
 [ 0.  0.  0.  0.  0.  0.  0.  0. -1.  2.]]
```

Further reading

Trefethen, L.N. and Bau III, D., 1997. Numerical linear algebra (Vol. 50). Siam.

QR and LU decomposition

The most common method for finding a numerical solutions of an equation $A\mathbf{x} = \mathbf{b}$ make use of QR and LU decompositions.

QR decomposition (or factorisation) decomposes a matrix A into a product $A = QR$, where Q is an orthogonal (or unitary if complex) matrix and R is upper-triangular. An orthogonal matrix Q is a matrix whose transpose is equal to its inverse, i.e. $Q^T = Q^{-1}$. Having found Q and R, solving the equation is then performed via a simple back-substitution because the LHS is upper-triangular:

$$\begin{aligned} A\mathbf{x} &= \mathbf{b} \\ QR\mathbf{x} &= \mathbf{b} \\ R\mathbf{x} &= Q^T\mathbf{b}. \end{aligned}$$

Hence we got away with calculating an inverse of A, but rather we found an inverse of Q which does not require any further calculations.

LU decomposition factorises A such that $A = LU$ where L is a lower-triangular matrix and U is an upper triangular matrix.

$$\begin{aligned} A\mathbf{x} &= \mathbf{b} \\ LU\mathbf{x} &= \mathbf{b} \end{aligned}$$

First we solve $L\mathbf{y} = \mathbf{b}$ for $\mathbf{y}$, where $\mathbf{y} = U\mathbf{x}$ and then we solve $U\mathbf{x} = \mathbf{y}$ for $\mathbf{x}$. It is very simple to solve these two equations since solving triangular systems involves only back- or forward- substitution.