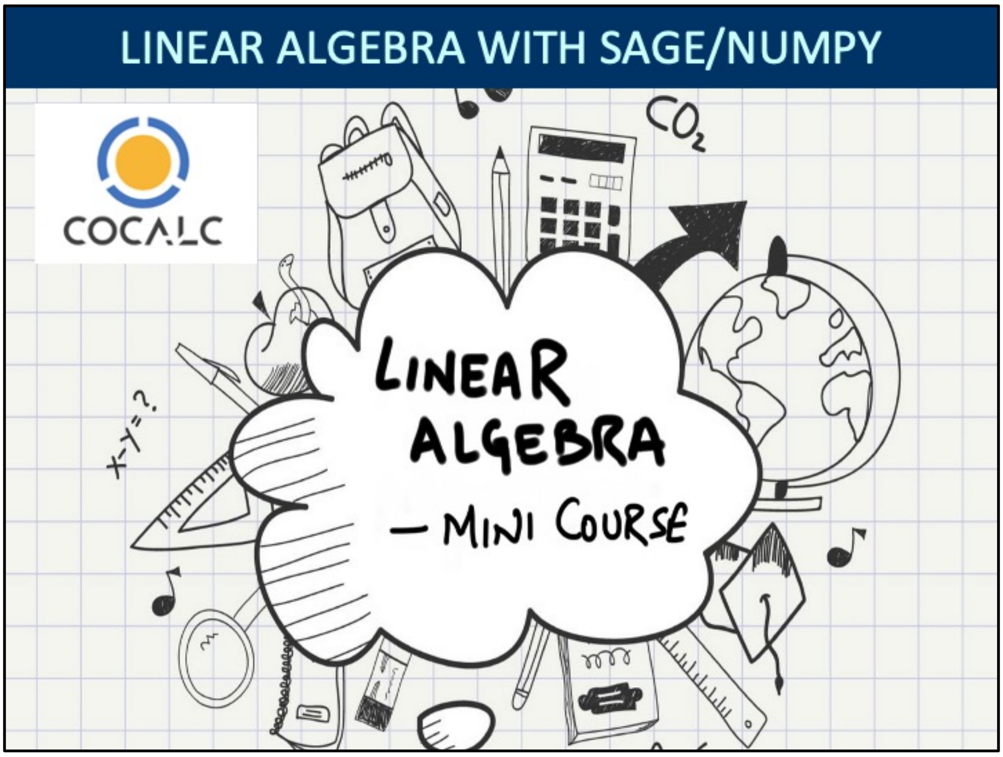
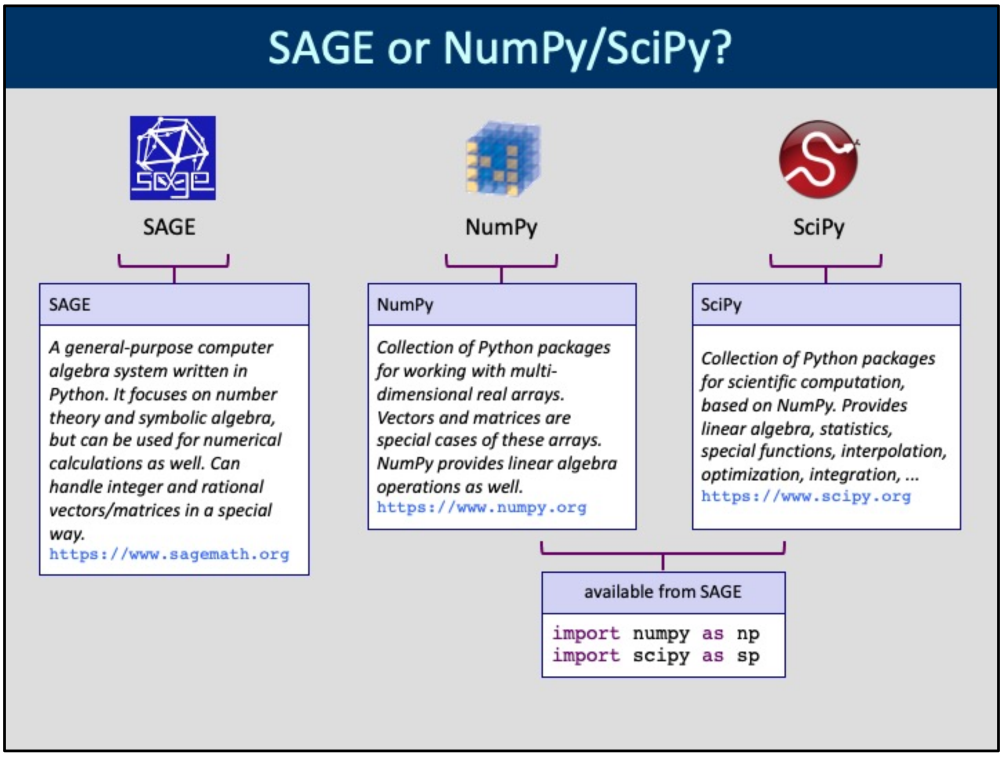
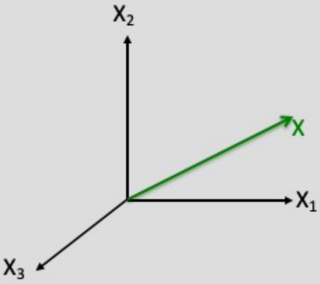


We will learn about the linear algebra concepts necessary for the Biocybernetics / Systems Biology course. The slides are accompanied by simple homework assignments.



CoCalc offers two ways of doing linear algebra. You can either use the SAGE classes and functions (which are somewhat special), or you can rely on the NumPy/SciPy package collection. In the following we will try both.

Vectors



$$\mathbf{x} = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} \qquad a\,\mathbf{x} + b\,\mathbf{y} = \begin{pmatrix} a\,x_1 + b\,y_1 \\ a\,x_2 + b\,y_2 \\ \vdots \\ a\,x_n + b\,y_n \end{pmatrix}$$

SAGE style

```
xs = vector([1.0,3.0,2.0])
ys = vector([-2.0, 1.5, 3.4])
```

Vector type

Note that we use lists of reals to tell SAGE that we need vectors with real elements.

NumPy style

```
xn = np.array([1.0,3.0,2.0])
yn = np.array([-2.0, 1.5, 3.4])
```

NumPy array

One-dimensional NumPy arrays have vector semantics by default.

An n-dimensional vector is an ordered list of n numbers (usually real or complex). It can be interpreted as the coordinates of a point in an n-dimensional space.

Scalar product and norm

$$\mathbf{x} \cdot \mathbf{y} = \sum_{i=1}^{n} x_i y_i = \|\mathbf{x}\| \, \|\mathbf{y}\| \, \cos\alpha$$

Scalar product

Can be used to calculate the projection of one vector onto another or the angle between the vectors.

$$\|\mathbf{x}\| = \sqrt{\mathbf{x} \cdot \mathbf{x}} = \sqrt{\sum_{i=1}^{n} x_i^2}$$

Vector norm

The length of the vector: the square root of the scalar product of the vector with itself.

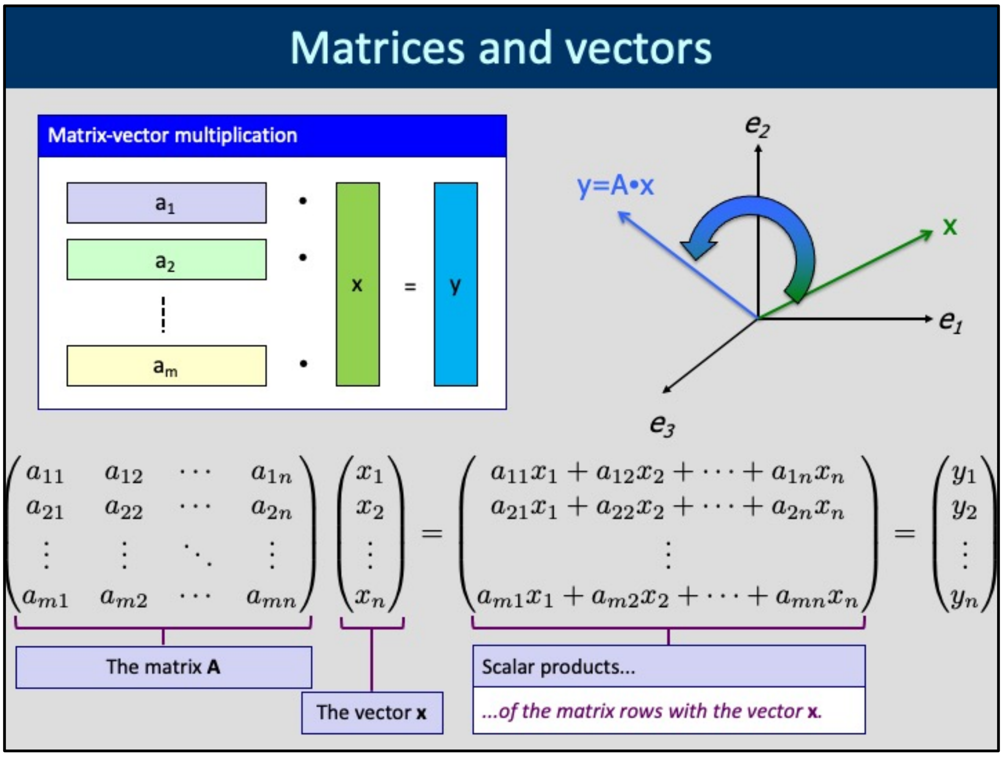
SAGE style

```
xs.dot_product(ys)
|   9.3
ys.dot_product(xs)
|   9.3
norm(xs)
|   3.741657
```

NumPy style

```
np.dot(xn,yn)
|   9.3
np.dot(yn,xn)
|   9.3
np.linalg.norm(yn)
|   4.2201895
```

If two vectors are orthogonal to each other, then their scalar product is zero, because cos 90° = 0. From this follows that the null vector is orthogonal to all other vectors.



Matrices are rectangular tables of numbers. The rows and/or columns can be regarded as vectors; you can think about a matrix as a row vector of column vectors or as a column vector of row vectors. If the numbers of the rows and columns are equal then we have a square matrix.
The product of a matrix with a vector is defined as the scalar product of the row vectors of the matrix with the vector. By multiplying a vector with a matrix we get a new vector which is a rotated and scaled version of the original vector.

Solving linear equation systems

$$2x_1 + 3x_2 = 6$$
$$4x_1 + 9x_2 = 15$$

$$\underbrace{\begin{pmatrix} 2 & 3 \\ 4 & 9 \end{pmatrix}}_{\mathbf{A}} \cdot \underbrace{\begin{pmatrix} x_1 \\ x_2 \end{pmatrix}}_{\mathbf{x}} = \underbrace{\begin{pmatrix} 6 \\ 15 \end{pmatrix}}_{\mathbf{b}}$$

$$\mathbf{A} \cdot \mathbf{x} = \mathbf{b}$$
$$\mathbf{x} = \mathbf{A}^{-1} \cdot \mathbf{b}$$

The formal solution

By multiplying the right-hand-side vector with the inverse of the coefficient matrix, we get the solution. This is not the practical way, however...

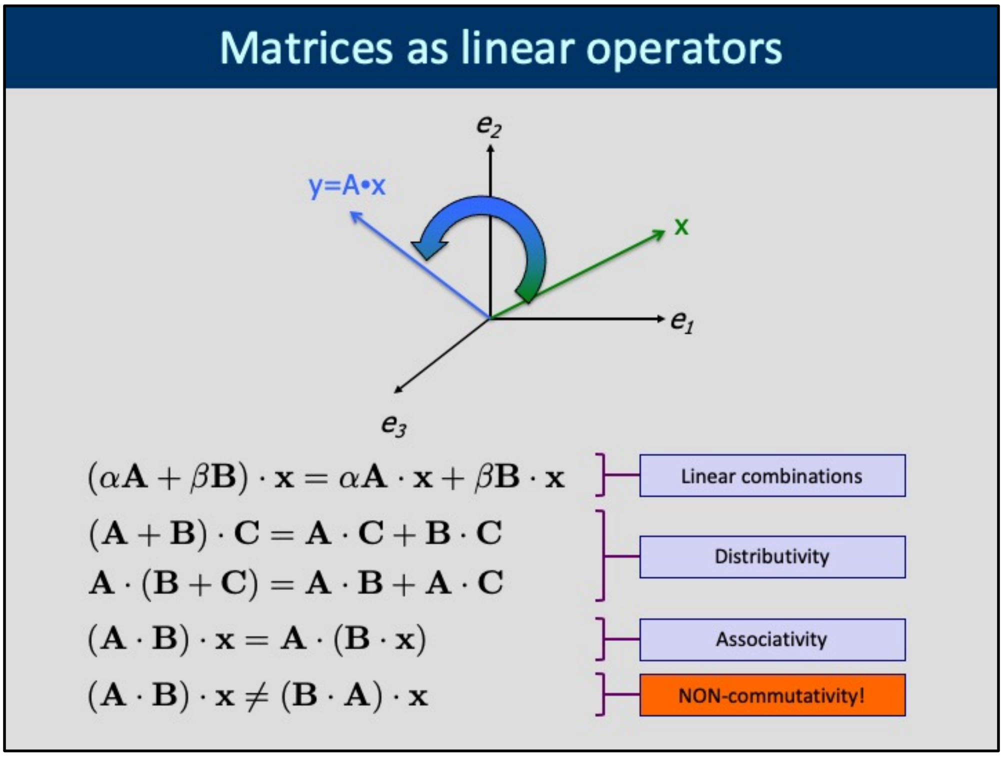
SAGE style

```
A = matrix( [[2,3],[4,9]])
b = vector( [6,15])
# naive way, don't do it!
x = A.inverse() * b
|   (3/2, 1)
# proper way
x = A \ b; x
|   (3/2, 1)
```

NumPy style

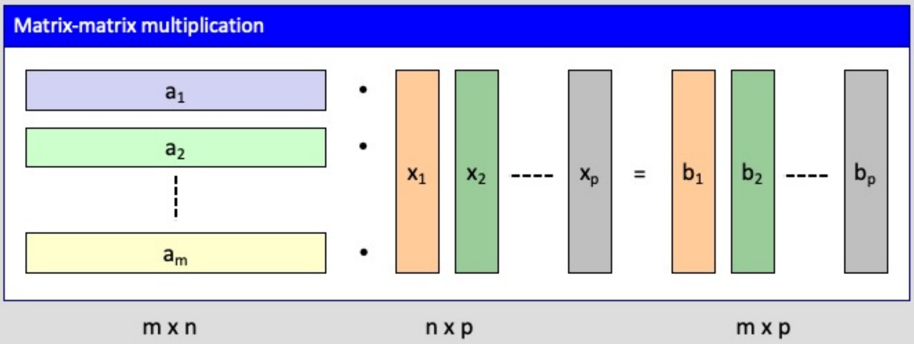
```
A = np.array([[2,3],[4,9]])
b = np.array([6,15])
# naive way, don't do it!
Ainv = np.linalg.inv(A)
x = Ainv.dot(b)
|   array([ 1.5,   1. ])
# proper way
x = np.linalg.solve(A,b); x
|   array([ 1.5,   1. ])
```

Of course you can solve a linear equation system only if there are as many equations as unknowns (i.e. $m = n$). Additional conditions must be satisfied, e.g. the equations should not be linear combinations of each other (we say they shall be linearly independent), nor should they be "contradictory". All these requirements can be formalised using mathematical techniques that go beyond the scope of this lecture. There are special algorithms that are used by numerical packages such as SAGE or NumPy/SciPy to solve linear equation systems. They NEVER calculate the matrix inverse, because it can be done only in $O(N^3)$ time.



Because the product of a matrix and a vector results in a new vector that is a rotated and scaled version of the original vector, we can say that a matrix transforms a vector into another one by rotating and dilating/contracting it. Matrices can thus be regarded as operators, mapping vectors onto other vectors. Some of the operator properties are listed on the slide.

Matrix-matrix multiplication [1]



$$\begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix} \begin{pmatrix} x_{11} & x_{12} & \cdots & x_{1p} \\ x_{21} & x_{22} & \cdots & x_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ x_{n1} & x_{n2} & \cdots & x_{np} \end{pmatrix} = \begin{pmatrix} b_{11} & b_{12} & \cdots & b_{1p} \\ b_{21} & b_{22} & \cdots & b_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ b_{m1} & b_{m2} & \cdots & b_{mp} \end{pmatrix}$$

The matrix-matrix multiplication is a generalization of the matrix-vector multiplication. We multiply each column vector of the second matrix (X) by the first matrix (A) and use the result vectors as the columns of the result matrix (B). The number of columns of the first matrix (n) should be equal to the number of rows of the second matrix.

Matrix-matrix multiplication [2]

$$\begin{pmatrix} 2 & 3 \\ 4 & 9 \end{pmatrix} \cdot \begin{pmatrix} 4 & 5 \\ 6 & 7 \end{pmatrix} = \begin{pmatrix} 26 & 31 \\ 70 & 83 \end{pmatrix} \qquad \begin{pmatrix} 4 & 5 \\ 6 & 7 \end{pmatrix} \cdot \begin{pmatrix} 2 & 3 \\ 4 & 9 \end{pmatrix} = \begin{pmatrix} 28 & 57 \\ 40 & 81 \end{pmatrix}$$

$$\mathbf{A} \cdot \mathbf{B} = \left[\sum_{k=1}^{p} a_{ik} a_{kj} \right] \neq \mathbf{B} \cdot \mathbf{A}$$

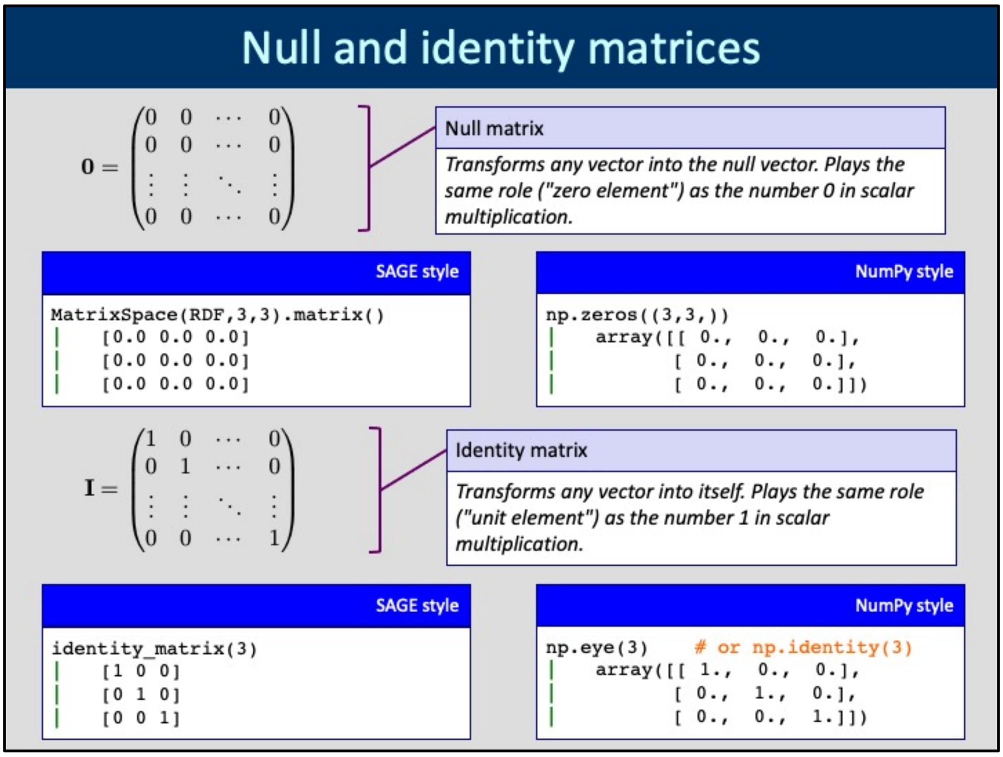
SAGE style

```
A = matrix( [[2,3],[4,9]])
B = matrix( [[4,5],[6,7]])
print(A*B)
|    [26 31]
|    [70 83]
print(B*A)
|    [28 57]
|    [40 81]
```

NumPy style

```
A = np.array([[2,3],[4,9]])
B = np.array([[4,5],[6,7]])
print(np.matmul(A,B))
|    array([[26, 31],
|         [70, 83]])
print(np.matmul(B,A))
|    array([[28, 57],
|         [40, 81]])
```

Unlike the multiplication of scalars, in general matrix multiplication is NOT commutative, i.e. the order of the matrices does matter.
Note that if you use NumPy-style matrices, then the multiplication operator * will perform element-wise multiplication! This is quite dangerous because SAGE-style matrices can be matrix-multiplied using the * operator.



The identity matrix should be square (i.e. NxN), the zero or null matrix need not be square, can be NxM.

The diagonal matrix

$$\mathbf{\Lambda} = \begin{pmatrix} \lambda_1 & 0 & \cdots & 0 \\ 0 & \lambda_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \lambda_n \end{pmatrix}$$

Diagonal matrix

Multiplying a matrix from the left with a diagonal matrix is equivalent of multiplying the rows with the corresponding main diagonal elements.

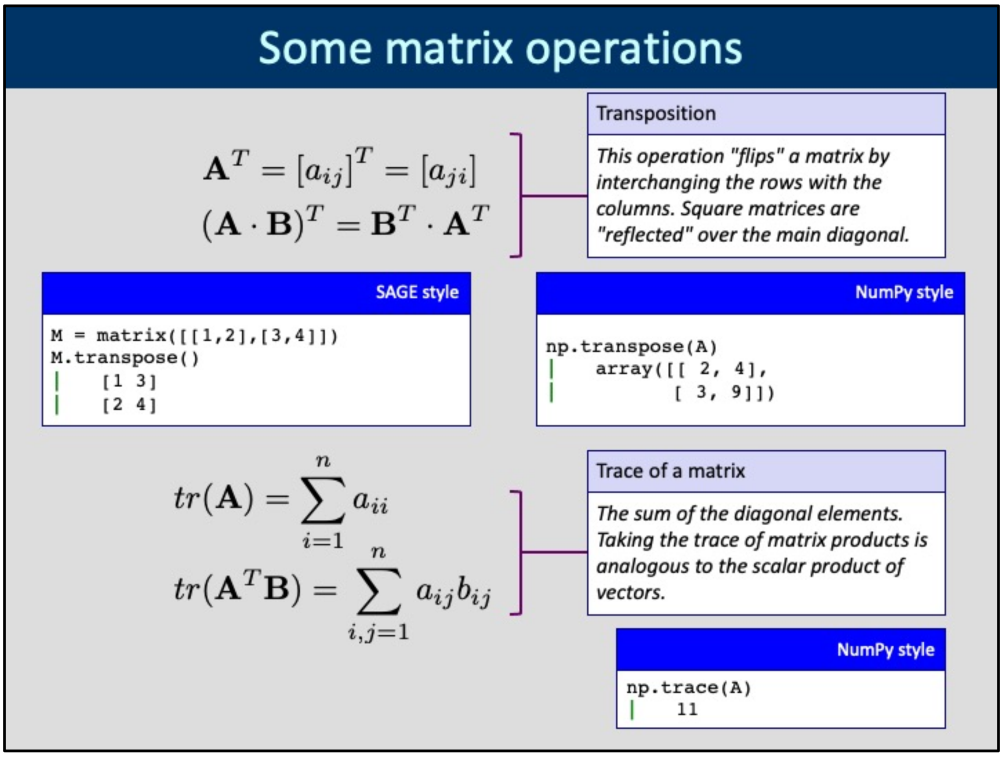
SAGE style

```
diagonal_matrix([1.0,2.0,3.0])
|    [1.0 0.0 0.0]
|    [0.0 2.0 0.0]
|    [0.0 0.0 3.0]
```

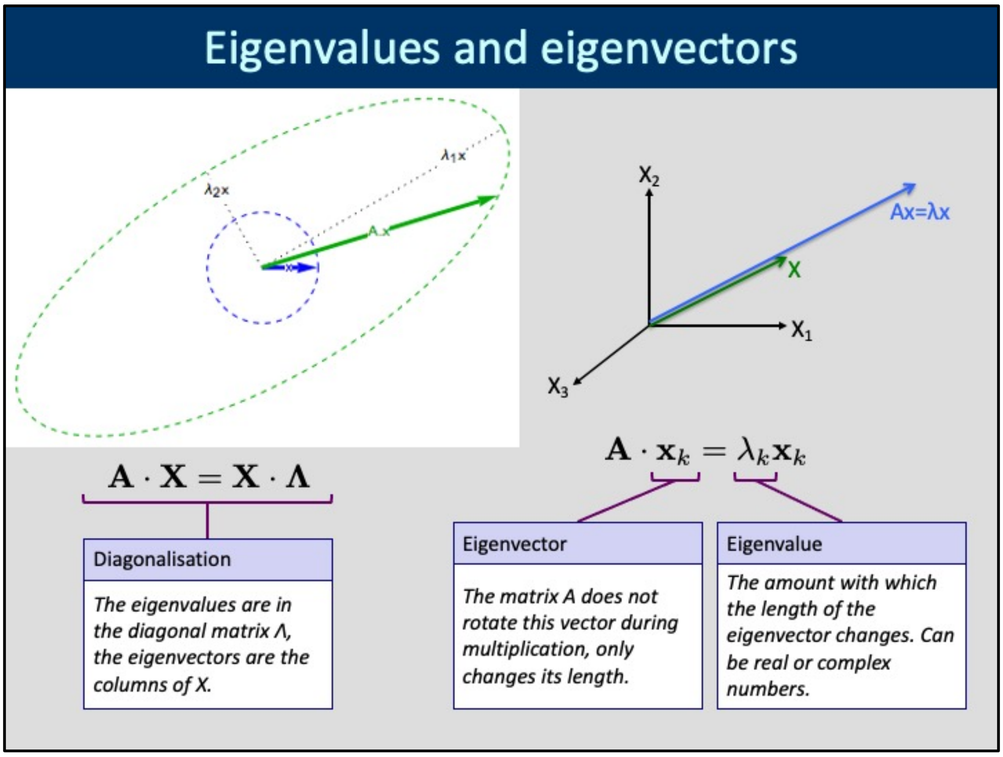
NumPy style

```
np.diag([1,2,3])
|    array([[ 1.,  0.,  0.],
|           [ 0.,  2.,  0.],
|           [ 0.,  0.,  3.]])
```

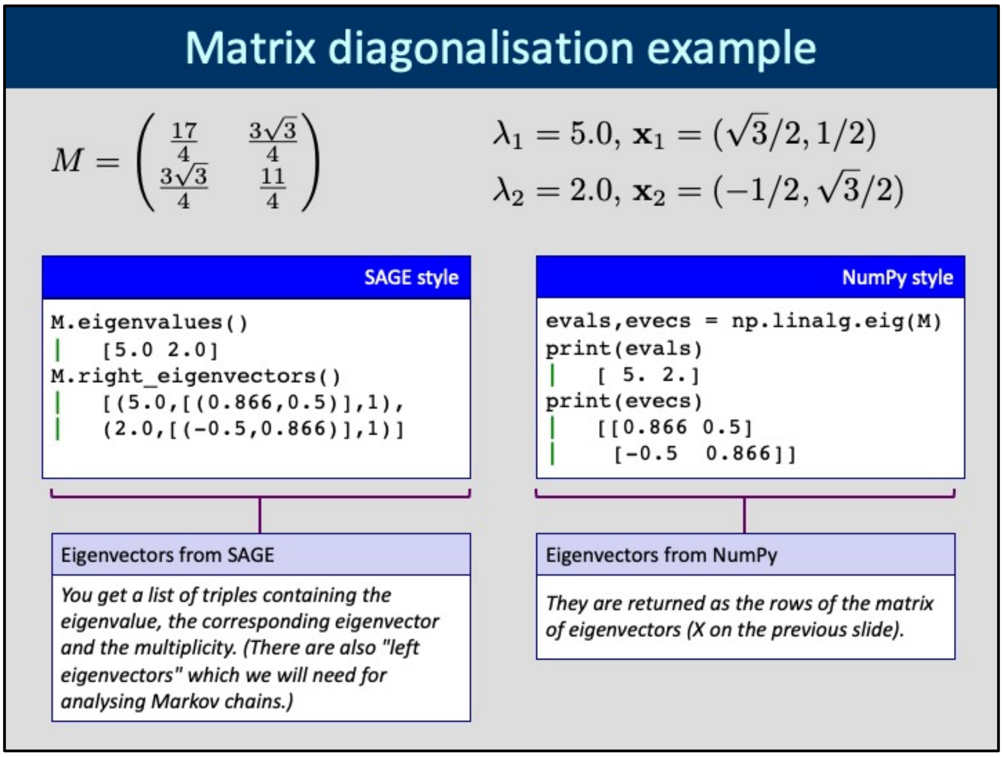
Diagonal matrices make linear algebra operations "simpler". We will make use of them later when we discuss eigenvalues and eigenvectors.



transpose(A) in SAGE and np.transpose(A) in NumPy style both transpose matrices. The np.trace(A) function works only in NumPy, because trace() in SAGE means tracing the execution...



In general matrices rotate and scale (dilate or contract) the vectors during matrix-vector multiplication. For a rectangular nxn matrix, are there any vectors which are not rotated by the matrix, only the length is changed? In most cases we can find such vectors: these are the "eigenvectors" of the matrix (the German word "eigen" means "own", it has nothing to do with the German Nobel laureate Manfred Eigen). The factor by which the matrix dilates or contracts its eigenvector is the "eigenvalue", indicated by λ in the equation above. In general an nxn matrix has n eigenvalues and eigenvectors, they can be bundled together into the eigenvector matrix X and the diagonal eigenvalue matrix Λ. Sometimes more than one eigenvalues have the same value, they are called "degenerate".



This is just a concrete example, illustrating also the fact that symmetric matrices with real elements have real eigenvalues (in general eigenvalues can be complex numbers).