1. Linear Algebra in Computation

A huge fraction of numerical computation involves linear algebra at some level or another. This includes simple application of matrix multiplication, but also applications of matrix inversion, decompositions, and other important operations. Vectors and higher-dimensional objects in linear algebra are held as arrays in NumPy (and other languages).

You are aware of the usual matrix notation:

$$\begin{aligned} x_i &\rightarrow \vec{x} \\ Q_{ij} &\rightarrow \mathbf{Q} \end{aligned} \tag{1}$$

so for example:

$$y_i = \sum_j Q_{ij} x_j \tag{2}$$

may also be written:

$$\vec{y} = \mathbf{Q} \cdot \vec{x} \tag{3}$$

If you have an array in Python or another language, you can perform these operations explicitly, but it is better to use the explicit matrix operations in NumPy and associated packages. It will be far faster. If you delve deep into high-order objects (e.g. T_{ijkl}) you can definitely run into cases where the standard routines won't do the operation you want, but most cases will work fine.

I show some examples in the workbook.

2. Linear systems of equations

Many physical and statistical problems boil down to solving a linear system of the form:

$$\mathbf{A} \cdot \vec{x} = \vec{b} \tag{4}$$

where $\mathbf{A}$ and $\vec{b}$ are known, and we want to know $\vec{x}$.

If A is an $M \times N$ matrix (and therefore $\vec{x}$ is an N-vector) then under what conditions is there a unique, exact solution $\vec{x}$ to this equation?

How can we check if matrices are singular?

You have probably done solved linear systems by hand using a Gauss-Jordan type of technique. This is not what is done numerically. Instead, the simplest matrix inversion technique of practical use is *LU decomposition*.

LU decomposition yields a lower triangular matrix $\mathbf{L}$ and and upper triangular matrix $\mathbf{U}$, which satisfy:

$$\mathbf{L} \cdot \mathbf{U} = \mathbf{A} \tag{5}$$

This then means we can solve:

$$\mathbf{L} \cdot \vec{y} = \mathbf{b} \tag{6}$$

and then:

$$\mathbf{U} \cdot \vec{x} = \mathbf{y} \tag{7}$$

and the resulting $\vec{x}$ satisfies:

$$\mathbf{A} \cdot \vec{x} = \mathbf{L} \cdot \mathbf{U} \cdot \vec{x} = \mathbf{L} \cdot \vec{y} = \vec{b} \tag{8}$$

Hooray, but how does that help us. Well, with a triangular matrix, the equations 6 and 7 can easily be solved by backsubstitution. That is, you can start at:

$$y_0 = \frac{b_0}{L_{00}} \tag{9}$$

and then find:

$$y_i = \frac{b_i - \sum_{j=0}^{i-1} L_{ij} y_j}{L_{11}} \tag{10}$$

How many operations does the solution take once you have L and U?

It turns out that the LU decomposition can be done efficiently, with the same scaling with N. Now, in most implementations, in fact what is done is that the rows of **A** are shuffled in the procedure so as to keep the numerics stable.

Also, the decomposition is not unique. There are $N^2 + N$ values to set in **L** and **U** but only N^2 values in **A**. A typical choice is to let the diagonal of **L** be just 1s.

This gives me a way of solving the equations. How do I determine the inverse of A?

If I have an LU decomposition of A, how can I calculate the determinant of A?

I show in the notebook an example of using the `linalg` routine `solve`, which utilizes this technique to solve a set of linear equations.

3. Sparse matrices

There are many applications where you have in principle very large matrices, but they are in fact very sparse.

band-diagonal