

Computational linear algebra and systems of linear equations

What computational linear algebra is for

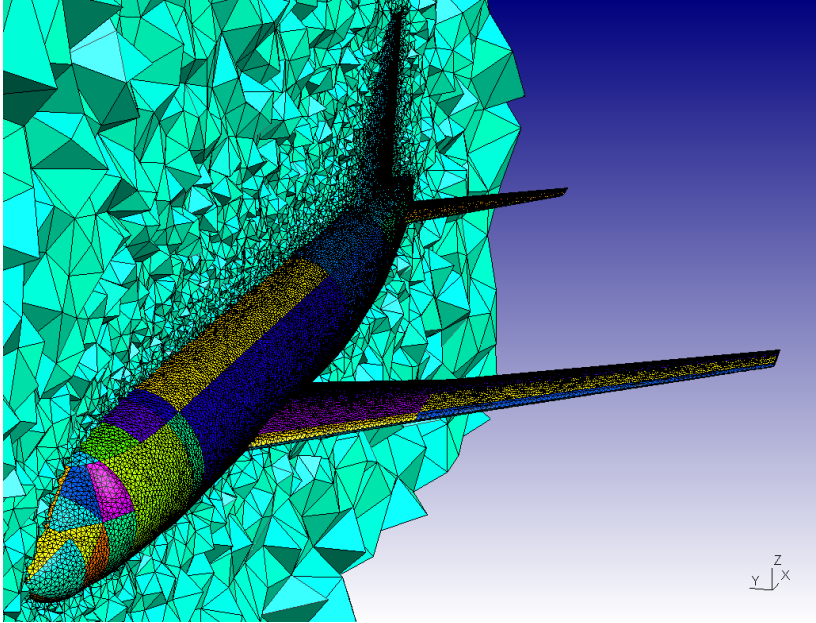
What we do in this section (and what we don't)

Linear algebra in python

Solution to the eigenvalue problem

- Finding the largest eigenvalue (basic power method)
- Finding the smallest eigenvalue (inverse power method)
- Finding the complete set of eigenvalues (QR iteration)

Almost all models of physical systems can be expressed as a set of simultaneous equations.



To solve these types of problems we need to numerically solve these (typically very large) matrix equations.

$$\begin{bmatrix} f_{x1} \\ f_{y1} \\ f_{x2} \\ f_{y2} \\ f_{x3} \\ f_{y3} \end{bmatrix} = \begin{bmatrix} k_{x1x1} & k_{x1y1} & k_{x1x2} & k_{x1y2} & k_{x1x3} & k_{x1y3} \\ k_{y1x1} & k_{y1y1} & k_{y1x2} & k_{y1y2} & k_{y1x3} & k_{y1y3} \\ k_{x2x1} & k_{x2y1} & k_{x2x2} & k_{x2y2} & k_{x2x3} & k_{x2y3} \\ k_{y2x1} & k_{y2y1} & k_{y2x2} & k_{y2y2} & k_{y2x3} & k_{y2y3} \\ k_{x3x1} & k_{x3y1} & k_{x3x2} & k_{x3y2} & k_{x3x3} & k_{x3y3} \\ k_{y3x1} & k_{y3y1} & k_{y3x2} & k_{y3y2} & k_{y3x3} & k_{y3y3} \end{bmatrix} \begin{bmatrix} u_{x1} \\ u_{y1} \\ u_{x2} \\ u_{y2} \\ u_{x3} \\ u_{y3} \end{bmatrix}$$

$$\mathbf{f} = \mathbf{K}\mathbf{u}$$

Forces acting at each point (known)

Displacement of each point (unknown)

Stiffness matrix (known)

What we will not cover here:

The “under the hood” methods of basic matrix manipulation, including:

- Numerical matrix multiplication
- Gaussian elimination
- Simple matrix inversion
- Matrix decomposition

What we will focus on instead:

Using in-built Python routines to do all the “basic stuff” above

An intro to the more advanced stuff

- the eigenvalue problem
- manipulating sparse matrices)

Matrices in python

The basic form that python uses to store matrices is the numpy array.

We use *nested square brackets* to define the columns and rows, with commas separating the elements:

```
7 import numpy as np
8
9 A = np.array([[2,0,1],[1,0,0],[-1,1,0]])
10
```

To define a column vectors and row vectors we need to remember to keep the nested brackets, e.g.

```
12 a = np.array([[1,2, 3]])
13
14 b = np.array([[2],[0],[-2],[1]])
```

The shape attribute returns how many columns and rows the matrix has.

Basic matrix manipulations are built-in to numpy:

Addition/subtraction:

```
12 B = A + A
13
14 C = A.T
15
```

Transpose:

Note that $A*B$ is not matrix multiplication – it just multiplies the two matrices element by element.

Matrix multiplication can be done in a few ways, all of which are equivalent:

1. using the @ notation

```
16 D1 = A @ B
```

2. Using the dot attribute

```
18 D2 = A.dot(B)
```

3. Using the inbuilt numpy matmul function

```
19  
20 D3 = np.matmul(A,B)  
21
```

For more complex matrix operations we use the linalg package from the scipy module:

```
10 import numpy as np
11 from scipy import linalg as la
12
```

Using this we can do matrix inversion:

```
In [3]: print(A)
[[ 2  0  1]
 [ 1  0  0]
 [-1  1  0]]

In [4]: B = la.inv(A)

In [5]: print(B)
[[ 0.  1. -0.]
 [ 0.  1.  1.]
 [ 1. -2.  0.]]
```

And solve linear systems directly:

```
In [8]: b = np.array([ 2],[0],[1])

In [9]: print(b)
[[2]
 [0]
 [1]]

In [10]: x = la.solve(A,b)

In [11]: print(x)
[[0.]
 [1.]
 [2.]]
```

An important quantity in linear algebra is the *matrix norm*, which expresses the “aggregate size” of the elements in a matrix.

For a matrix A with elements $A_{i,j}$, the L_p norm of A is defined by

$$\|A\|_p = \sqrt[p]{\sum_i \sum_j |A_{i,j}|^p}$$

The L_1 norm (known as the “manhattan distance”) is the sum of the Absolute values in the matrix

The L_2 norm (the “Euclidean norm”) is the square root of the sum of the squares of all the elements

The L_∞ norm is equal to the *largest value* in the matrix.

In python, norms can be computed using the scipy linalg package:

```
In [19]: print(b)
[[2]
 [0]
 [1]]

In [20]: la.norm(b,np.inf)
Out[20]: 2.0

In [21]: la.norm(b,2)
Out[21]: 2.23606797749979

In [22]: la.norm(b,1)
Out[22]: 3.0
```

Matrix decompositions

An important aspect in most computational linear algebra is being able to decompose a matrix in terms of other, simpler matrices.

The LU decomposition expresses a matrix as the product of two simpler matrices:

```
In [23]: print(A)
[[ 2  0  1]
 [ 1  0  0]
 [-1  1  0]]

In [24]: P,L,U = la.lu(A)

In [25]: print(U)
[[ 2.  0.  1.]
 [ 0.  1.  0.5]
 [ 0.  0. -0.5]]

In [26]: print(L)
[[ 1.  0.  0.]
 [-0.5 1.  0.]
 [ 0.5 0.  1.]]

In [27]: L @ U
Out[27]:
array([[ 2.,  0.,  1.],
       [-1.,  1.,  0.],
       [ 1.,  0.,  0.]])
```

We will outsource the decompositions to the scipy linalg package:

The QR decomposition decomposes a matrix into an *orthogonal matrix* Q and a right-upper-triangular matrix R:

Aside: an orthogonal matrix Q is one for which the Inverse is equal to its own transpose:

$$Q^{-1} = Q^T$$

```
In [30]: Q,R = la.qr(A)

In [31]: print(Q)
[[-8.16496581e-01 -3.65148372e-01  4.47213595e-01]
 [-4.08248290e-01 -1.82574186e-01 -8.94427191e-01]
 [ 4.08248290e-01 -9.12870929e-01  3.08983562e-17]]

In [32]: print(R)
[[-2.44948974  0.40824829 -0.81649658]
 [ 0.         -0.91287093 -0.36514837]
 [ 0.          0.         0.4472136  ]]
```

Computational expense for basic matrix manipulations

The time taken for a matrix operation (inversion, decomposition, etc) can be characterised by the *computational complexity*, which is related to the size of the matrix.

Here we give the order of time for each of the major operations, for a *dense, unstructured matrix*:

Addition/subtraction of two $n \times m$ matrices

Multiplication of a $p \times m$ by a $m \times n$ matrix

Inversion of an $n \times n$ matrix:

LU decomposition of an $n \times n$ matrix:

QR decomposition of an $n \times n$ matrix:

The eigenvalue problem

Definition:

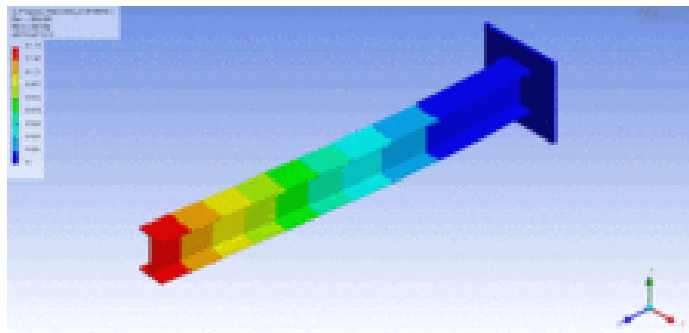
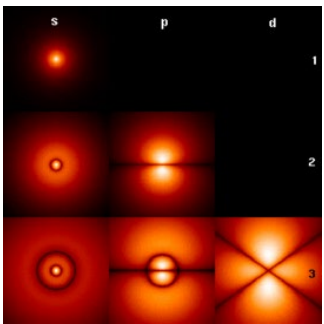
Let \mathbf{A} be a (square) $n \times n$ matrix. A nonzero vector \mathbf{v} is an *eigenvector* of \mathbf{A} if, for some scalar λ , \mathbf{v} satisfies

$$\mathbf{A}\mathbf{v} = \lambda\mathbf{v}$$

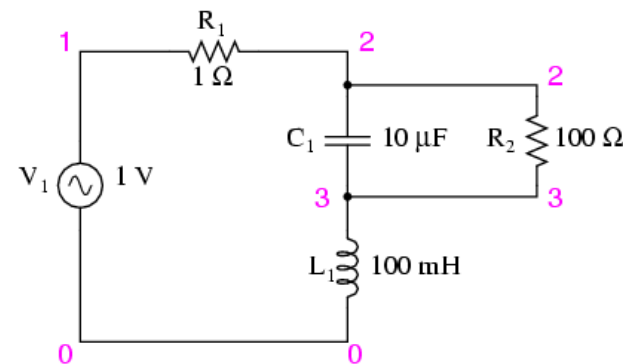
The scalar number λ (which may be zero) is called an *eigenvalue* of \mathbf{A} , associated with \mathbf{v} .

The set of all eigenvalues of \mathbf{A} is called the *spectrum* of \mathbf{A} .

Eigenvalues and Eigenvectors are extremely widely used in all branches of engineering and the physical sciences



Series LC with resistance in parallel with C

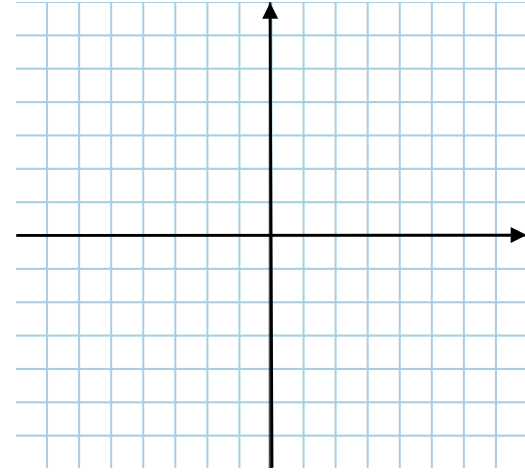
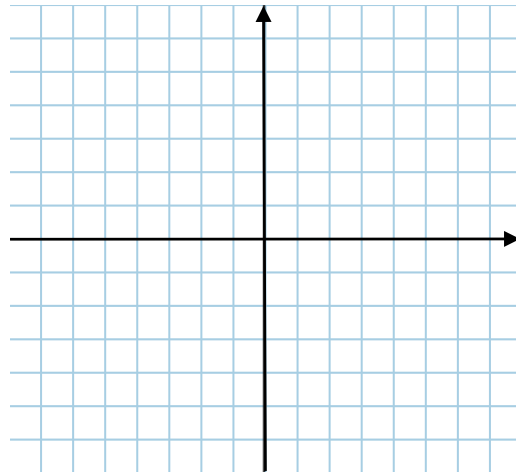


Eigenvalue \longleftrightarrow Frequency

Eigenvector \longleftrightarrow Mode of vibration

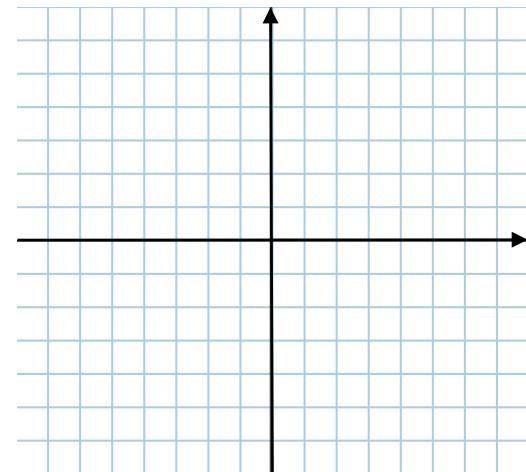
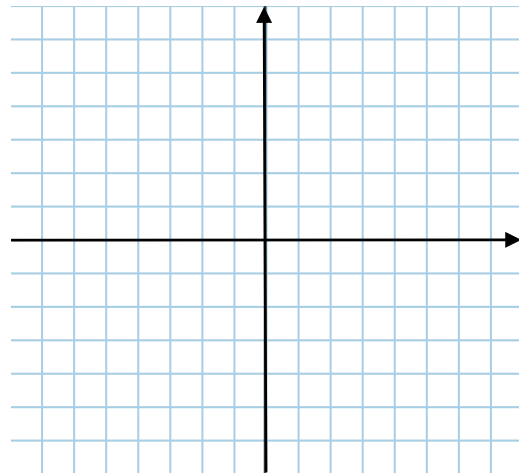
A matrix can be thought of as a *linear transformation of a vector*

$$u = Av$$



Sometimes a linear transformation changes the magnitude of a vector without affecting its “direction”. Such a vector is called an *eigenvector* of the matrix.

$$Av = \lambda v$$



For large matrices there will be a lot of eigenvalues, and their computation can be difficult.

Often we are interested only in the largest eigenvalue, or the smallest eigenvalue.

$$\mathbf{A}\mathbf{v} = \lambda\mathbf{v}$$

In this situation *Iterative Power methods* can be used.

The Basic Power Method

This algorithm computes the largest eigenvalue and eigenvector of a matrix.

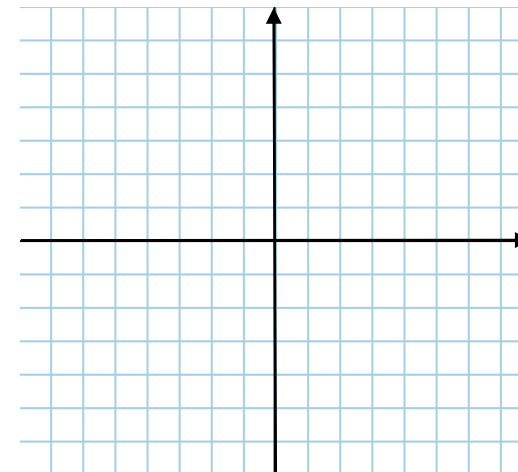
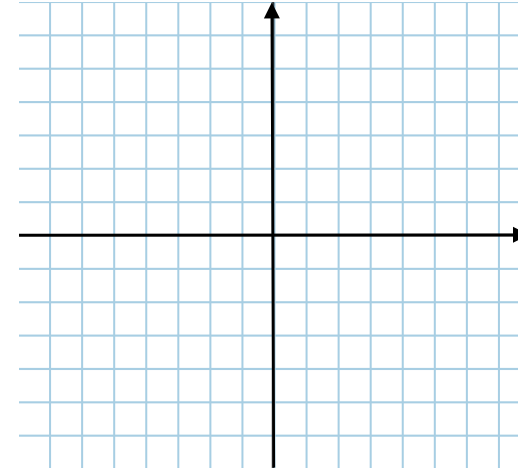
The idea: each transformation by a matrix A *stretches the vector in the direction of the largest eigenvector** v_{\max} of A .

If we keep on applying A to *any* vector, then eventually this vector will point in the direction of v .

Basic Power Iteration

1. Start with a vector v_k (preferably randomized)
2. Apply $u = Av_k$
3. Compute $v_{k+1} = \frac{u}{\mu_k}$, where $\mu_k = \|u\|_{\infty}$
4. Repeat from step 2 until converged.

The quantity μ_k converges to the largest eigenvalue, with eigenvector v_k .



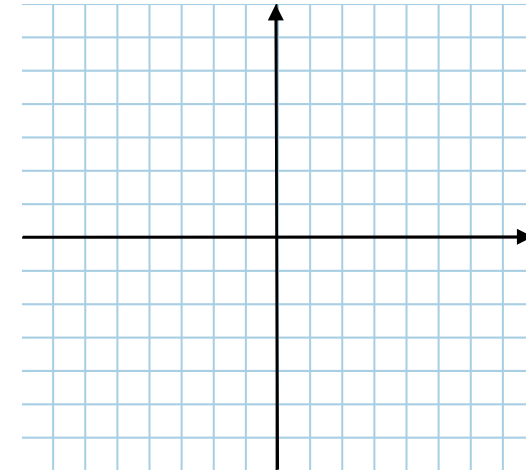
*vector with the largest eigenvalue

The inverse power method

The idea: the eigenvalues of A^{-1} are the reciprocals of the eigenvalues of A .
We can therefore find the *smallest* eigenvalue of A :

Inverse Power Iteration

1. Start with a vector v_k (preferably randomized)
2. Apply $u = A^{-1}v_k$
3. Compute $v_{k+1} = \frac{u}{\mu_k}$, where $\mu_k = \|u\|_\infty$
4. Repeat from step 2 until converged.



The quantity $1/\mu_k$ converges to the smallest eigenvalue of A , with eigenvector v_k .

The QR algorithm

This gives the complete set of eigenvalues of A. The only problem is that it is a bit slow to converge, and is computationally expensive if the matrix is large.

QR Iteration

1. Form the QR decomposition

$$Q_k R_k = A_k$$

2. Create a new matrix

$$A_{k+1} = R_k Q_k$$

3. Repeat from Step 1 until converged.

The eigenvalues are given by the diagonal components of A_k .

Other important approaches:

- Shifted Power method
 - modification of the power method to find the complete spectrum
- The Rayleigh Quotient method
 - A fast iteration method for finding the largest eigenvalue
- The Shifted QR method
 - The convergence of the QR method depends on the ratios between eigenvalues being large. Introducing a “shift” to the eigenvalues improves the convergence
- Arnoldi iteration
 - Combines power iteration with Gram-Schmidt orthogonalisation to compute the full spectrum (very fast for Sparse matrices)