Introduction to Data Science and Systems

Lecture 3: Computational Linear Algebra

Dr Zaiqiao Meng

University of Glasgow

Recap of the last lecture

Vectors

- **scalar multiplication** so that $a\mathbf{x}$ is defined for any scalar a. For real vectors, $a\mathbf{x} = [ax_1, ax_2, \ldots ax_n]$, elementwise scaling.

$$(\mathbb{R}, \mathbb{R}^n) \rightarrow \mathbb{R}^n$$

- **vector addition** so that $\mathbf{x} + \mathbf{y}$ vectors $\mathbf{x}, \mathbf{y}$ of equal dimension. For real vectors, $\mathbf{x} + \mathbf{y} = [x_1 + y_1, x_2 + y_2, \ldots x_d + y_d]$ the elementwise sum

$$(\mathbb{R}^n, \mathbb{R}^n) \rightarrow \mathbb{R}^n$$

- A **norm** $\| \mathbf{x} \|$ which allows the length of vectors to be measured.

$$\mathbb{R}_n \rightarrow \mathbb{R}_{\geq 0}$$

- An **inner product** $\langle \mathbf{x} \mid \mathbf{y} \rangle$, $\langle \mathbf{x}, \mathbf{y} \rangle$ or $\mathbf{x} \cdot \mathbf{y}$ which allows the angles of two vectors to be compared. The inner product of two orthogonal vectors is 0 . For real vectors $\mathbf{x} \cdot \mathbf{y} =$

$$x_1 y_1 + x_2 y_2 + x_3 y_3 \ldots x_d y_d$$

$$(\mathbb{R}^n, \mathbb{R}^n) \rightarrow \mathbb{R}$$

University of Glasgow

Operations with matrices

There are many things we can do with matrices:

- They can be **added** and **subtracted** $C = A + B$

$$(\mathbb{R}^{n\times m}, \mathbb{R}^{n\times m}) \to \mathbb{R}^{n\times m}$$

- They can be **scaled** with a scalar $C = sA$

$$(\mathbb{R}^{n\times m}, \mathbb{R}) \to \mathbb{R}^{n\times m}$$

- They can be **transposed** $B = A^T$; this exchanges rows and columns

$$\mathbb{R}^{n\times m} \to \mathbb{R}^{m\times n}$$

- They can be **applied** to vectors $\mathbf{y} = A\mathbf{x}$; this applies a matrix to a vector. (linear maps)

$$(\mathbb{R}^{n\times m}, \mathbb{R}^{m}) \to \mathbb{R}^{n}$$

- They can be **multiplied** together $C = AB$; this composes the effect of two matrices (composed maps)

University of Glasgow

Matrices and linear operators

Matrices are 2D arrays of reals that define **linear maps**;

- Vectors represent “points in space"
- Matrices represent *operations* that do things to those points in space.

The operations represented by matrices are a particular class of functions on vectors

University of Glasgow

Intended Learning Outcomes

- How discrete problems can be modelled using continuous mathematics, i.E. Using matrices
  - How graphs can be represented as matrices
  - How flows on graphs can be represented as matrix operations
- What eigenvectors and eigenvalues are
  - How the power iteration method can compute them
  - How they can be used to decompose matrices
- What the trace and determinant are, and the geometric intuition underlying them
- What positive (semi-)definiteness means and why it is important
- What the singular value decomposition (SVD) is and how it can can be used to compute functions of matrices
- What a linear system of equations is and how it can be represented by a matrix
  - What matrix inversion is and how it relates to solving linear systems of equations
  - The numerical problems with direct inversion
  - What the pseudo-inverse is, how it is derived from the SVD, and how it can be used
- How to normalise data by using matrix operations to "whiten" it
- What a low-rank approximation is and why you might use it

University of Glasgow

Graphs as matrices

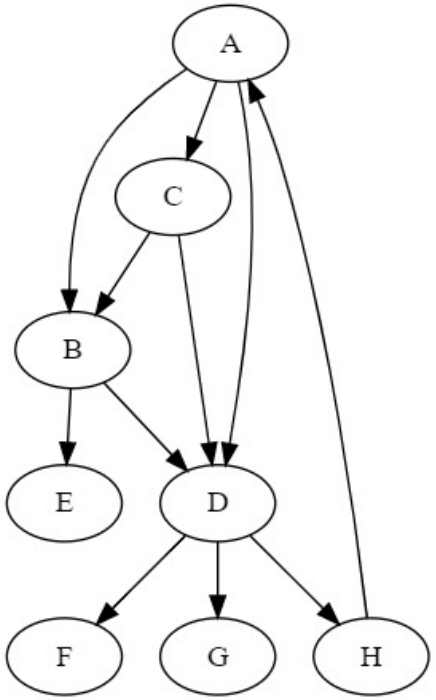
- **Example**: distributing packages
- **Task:** the packages and the connectivity between distributions centres
- **Target:** to predict which warehouses are going to receive lots of packages

How can this problem be modelled?

Graphs as matrices

- **Linear assumption:** the flow from site to site is linear -- that the packages arriving at one site is a weighted sum of the packages currently at each of the other sites

- We might model the connectivity of distribution centres as a **graph**. A **directed graph** connects **vertices** by **edges**. The definition of a graph is $G=(V,E)$, where V is a set of vertices and E is a set of edges connecting pairs of vertices.

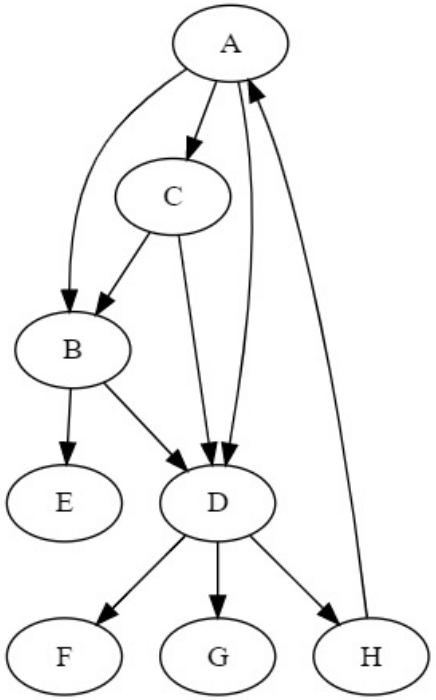


$A \rightarrow B$
$A \rightarrow C$
$A \rightarrow D$
$B \rightarrow D$
$B \rightarrow E$
$C \rightarrow B$
$C \rightarrow D$
$D \rightarrow F$
$D \rightarrow G$
$D \rightarrow H$
$H \rightarrow A$

Graphs as matrices

- We can write this as an **adjacency matrix**. We number each vertex 0,1,2,3,… We then create a square matrix A whose elements are all zero, except where there is an edge from V_i to V_j, in which case we set A_{ij}=1. The graph shown above has the adjacency matrix:

	A	B	C	D	E	F	G	H
A	0	1	1	1	0	0	0	0
B	0	0	0	1	1	0	0	0
C	0	1	0	1	0	0	0	0
D	0	0	0	0	0	1	1	1
E	0	0	0	0	0	0	0	0
F	0	0	0	0	0	0	0	0
G	0	0	0	0	0	0	0	0
H	1	0	0	0	0	0	0	0



$A \rightarrow B$
$A \rightarrow C$
$A \rightarrow D$
$B \rightarrow D$
$B \rightarrow E$
$C \rightarrow B$
$C \rightarrow D$
$D \rightarrow F$
$D \rightarrow G$
$D \rightarrow H$
$H \rightarrow A$

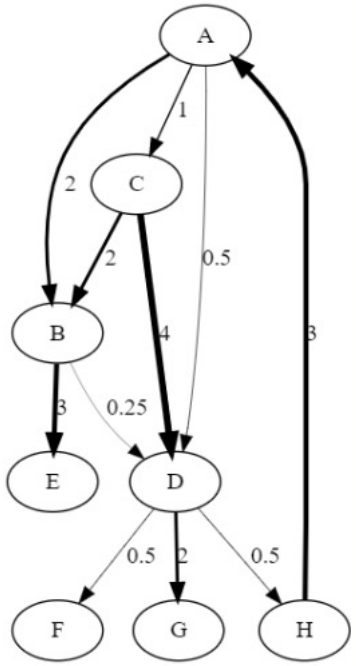
University of Glasgow

Computing graph properties

- The *out-degree* of each vertex (number of edges leaving a vertex) is the sum of each row.
- The *in-degree* of each vertex (number of edges entering a vertex) is the sum of each column.
- If the matrix is symmetric it represents an undirected graph; this is the case if it is equal to its transpose.
- A directed graph can be converted to an undirected graph by computing $A'=A+A^T$. This is equivalent to making all the arrows bi-directional.
- If there are non-zero elements on the diagonal, that means there are edges connecting vertices to themselves (self-transitions).

Edge-weighted graphs

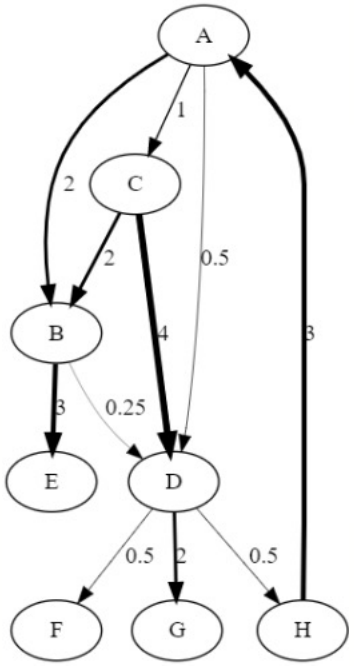
- If the some of the connections between distribution centres are stronger than others, e.g. if they are connected by bigger roads, we can model this using edge weights. Now the entry at A_{ij} represents the weight of the connection from vertex V_i to V_j.



	A	B	C	D	E	F	G	H
A	0.00	2.00	1.00	0.50	0.00	0.00	0.00	0.00
B	0.00	0.00	0.00	0.25	3.00	0.00	0.00	0.00
C	0.00	2.00	0.00	4.00	0.00	0.00	0.00	0.00
D	0.00	0.00	0.00	0.00	0.00	0.50	2.00	0.50
E	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
F	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
G	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
H	3.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00

Graphs as matrices

- **Example**: distributing packages
- **Task:** the packages and the connectivity between distributions centres
- **Target:** to predict which warehouses are going to receive lots of packages



	A	B	C	D	E	F	G	H
A	0.00	2.00	1.00	0.50	0.00	0.00	0.00	0.00
B	0.00	0.00	0.00	0.25	3.00	0.00	0.00	0.00
C	0.00	2.00	0.00	4.00	0.00	0.00	0.00	0.00
D	0.00	0.00	0.00	0.00	0.00	0.50	2.00	0.50
E	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
F	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
G	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
H	3.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00

University of Glasgow

Flow analysis: using matrices to model discrete problems

- **Adjacency matrix *A*** with edges being the weights of the **connection** between sites
- **Linear assumption:** the packages arriving at one site is a weighted sum of the packages currently at each of the other sites
- At any point in time, we can write down the proportion of packages at each depot as a vector $\boldsymbol{x_t} \in \mathbb{R}^V$, where *V* is the number of vertices in the graph (number of depots), e.g.

$$\mathbf{x_t} = [0.05 \quad 0.15 \quad 0.30 \quad 0.20 \quad 0.03 \quad 0.12 \quad 0.08 \quad 0.07]$$

  - Vectors, Matrices and linear maps
  - The flow of packages (per day) between depots is a linear map $\mathbb{R}^V \to \mathbb{R}^V$. This is represented by the adjacency matrix $A \in \mathbb{R}^{V \times V}$ (a square matrix).

University of Glasgow

Flow analysis: using matrices to model discrete problems

How many packages will be at each depot tomorrow?

Given matrix A and vector $\mathbf{x}_t$:

$$\mathbf{x}_t = [0.05 \quad 0.15 \quad 0.30 \quad 0.20 \quad 0.03 \quad 0.12 \quad 0.08 \quad 0.07]$$

and

$$A = \begin{bmatrix} 0.0 & 2.0 & 1.0 & 0.5 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.25 & 3.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 2.0 & 0.0 & 4.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.5 & 2.0 & 0.5 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 3.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \end{bmatrix}$$

This "**rotates**" the distribution of packages from today to tomorrow.

We'll compute the multiplication:

$$\mathbf{x}_{t+1} = \mathbf{x}_t A$$

To find how many will be in first site $\mathbf{x}_{t+1}(1)$ tomorrow, we can calculate the first element of the resulting vector $\mathbf{x}_{t+1}$:

$$\mathbf{x}_{t+1}(1) = (0.05 \times 0.0) + (0.15 \times 2.0) + (0.30 \times 1.0) + (0.20 \times 0.5) + \ldots = 0.3 + 0.3 + 0.1 = 0.7$$

University of Glasgow

Flow analysis: using matrices to model discrete problems

- This allows us to analyse an apparently *discrete* problem (connectivity of graphs) with tools from *continuous* mathematics (vectors and matrices).
  - e.g. from today to tomorrow

- The advantage of vectorised operations is that they can be accelerated using hardware such as a GPU

- Suppose we start with an *initial distribution* of packages: a vector $\boldsymbol{x_0} = \mathbf{0}$.

$$\mathbf{x_{t=1}} = \mathbf{x_{t=0}}\, A = A^T {\mathbf{x_{t=0}}}^T$$

Other Important questions

There are some harder questions we can ask:

- What about in a week‘s time? What will $x_{t=7}$ be?
- What about in *one hour's* time (i.e. a 24th of a day)? What will be $x_{t=1/24}$be?
- What about at time infinity $x_{t=\infty}$? What is the long term behaviour? Will the system reach a steady state (an **equilibrium**)? Or will it oscillate forever?
- What about if we wanted to go backwards in time? If we know $x_{t=0}$, can we predict yesterday $x_{t=-1}$?

We will solve these problems today, using some new operations that we can do with *certain kinds* of matrices:

University of Glasgow

Using matrices to model discrete problems

- packages moving between depots (how many packages at each depot at an instant in time)
- users moving between web pages (how many users on each webpage)
- cancer cells moving between tumour sites (how many cancer cells at each tumour site)
- trade between states (how many items in each state)
- shoppers walking between retailers (how many shoppers in each shop)
- traffic across a load-balancing network (how many cars at each junction)
- NPCs (non-player characters) moving between towns in a game (how many NPCs in each town)
- fluid moving between regions (how much fluid in each tank)
- blood flowing between organs (how much blood in each organ)
- beliefs moving among hypotheses (how much we believe in each hypothesis)

University of Glasgow

New matrix operations

- Matrices can be **exponentiated**: $\boldsymbol{C}=\boldsymbol{A}^n$; this "repeats" the effect of matrix (e.g. $\boldsymbol{C}=\boldsymbol{AAAA}$)
- Matrices can be **inverted**: $\boldsymbol{C}=\boldsymbol{A}^{-1}$; this undoes the effect of a matrix
- We can find **eigenvalues**: $\boldsymbol{Ax}=\boldsymbol{\lambda x}$; this identifies specific vectors x that are only scaled by a factor λ (not rotated) when transformed by matrix A.
- Matrices can be **factorised**: $\boldsymbol{A}=\boldsymbol{U\Sigma V}^T$; any matrix can expressed as the product of three other matrices with special forms.
- We can measure some properties of A numerically, including the **determinant**, **trace** and **condition number**.

Matrix powers (exponentiation)

- We can now define $A^2 = AA, A^3 = AAA, A^4 = AAAA$, etc

- These are the **powers** of a matrix, and are only defined for square matrices
  - we'd otherwise change the dimensions after the first step and be unable to reapply the same matrix

- **Package distribution**: What about in a week‘s time? What will $x_{t=7}$ be?
  - We can simply apply the matrix seven times; raising it to the power of 7.

Matrix powers (exponentiation)

- numpy.linalg.matrix_power(a, n)

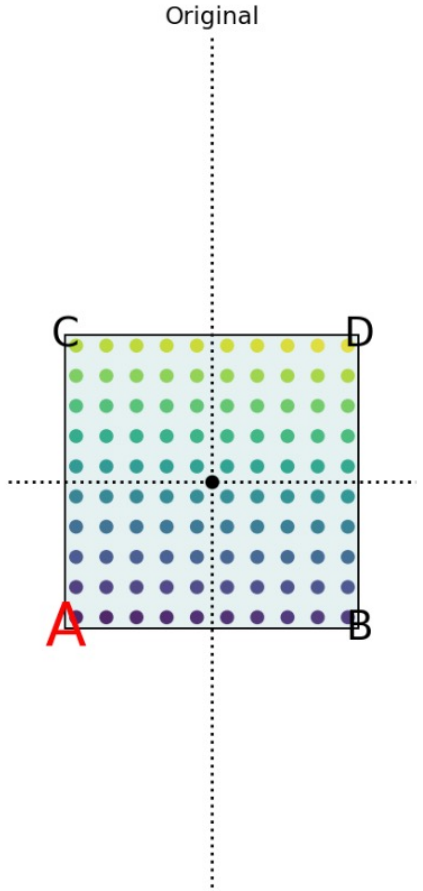
```
>>> from numpy.linalg import matrix_power
>>> i = np.array([[0, 1], [-1, 0]]) # matrix equiv. of
>>> matrix_power(i, 3) # should = -i
array([[ 0, -1],
       [ 1,  0]])
>>> matrix_power(i, 0)
array([[1, 0],
       [0, 1]])
>>> matrix_power(i, -3) # should = 1/(-i) = i, but w/ f
array([[ 0.,  1.],
       [-1.,  0.]])
```

- Applying a matrix multiple times

```
# Define a function that raises a matrix
def powm(A, n):
    B = np.eye(A.shape[0]) # start with
    for i in range(n):
        B = A @ B  # @ performs matrix m
    return B
```

```
# Demonstrate raising a matrix A to powe
A = np.array([[1.5, 0.0], [-1.1, 0.1]])
print_matrix("A", A)

for i in range(5):
    print_matrix("A^%d" % i, powm(A, i))
```

```
A
 [[ 1.5  0. ]
 [-1.1  0.1]]
A^0
 [[1. 0.]
 [0. 1.]]
A^1
 [[ 1.5  0. ]
 [-1.1  0.1]]
A^2
 [[ 2.25  0.  ]
 [-1.76  0.01]]
A^3
 [[ 3.38  0.  ]
 [-2.65  0.  ]]
A^4
 [[ 5.06  0.  ]
 [-3.98  0.  ]]
```

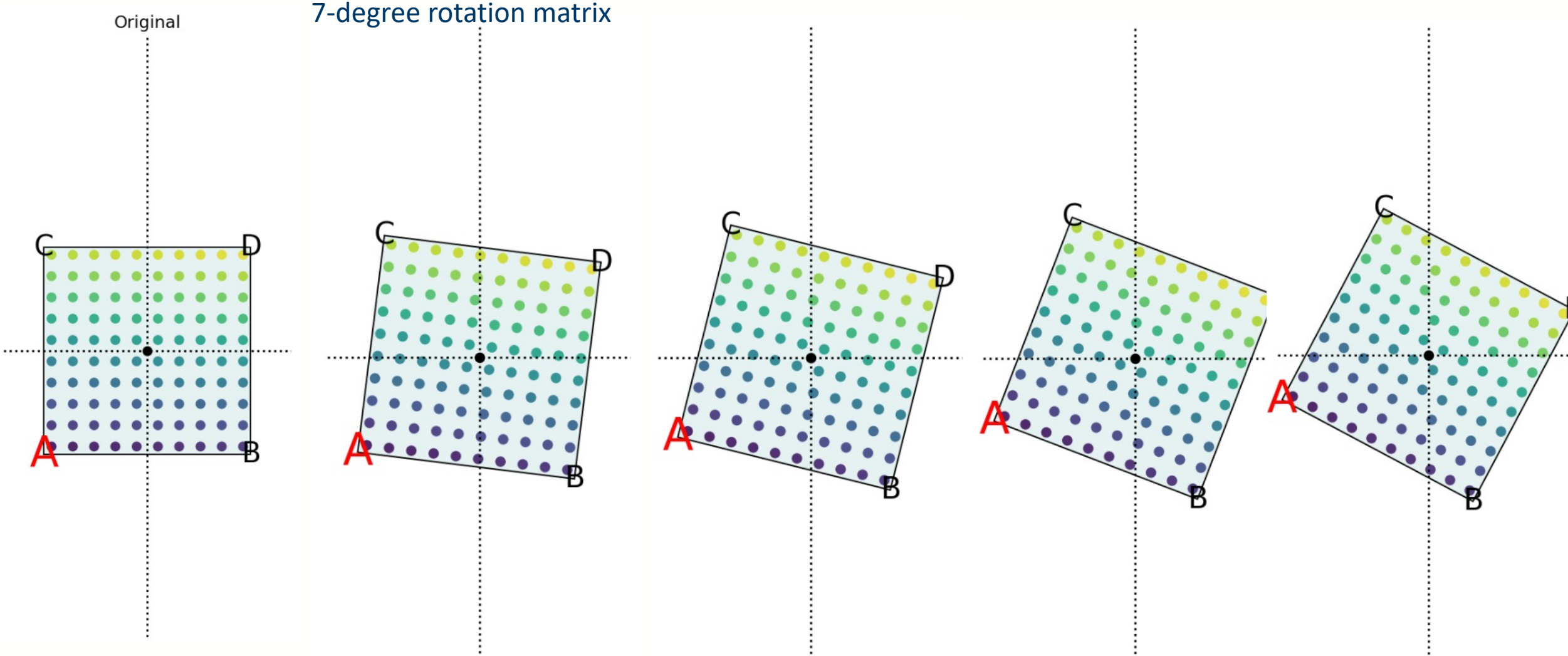


Matrix powers

- Original

A^1 A^2 A^3 A^4

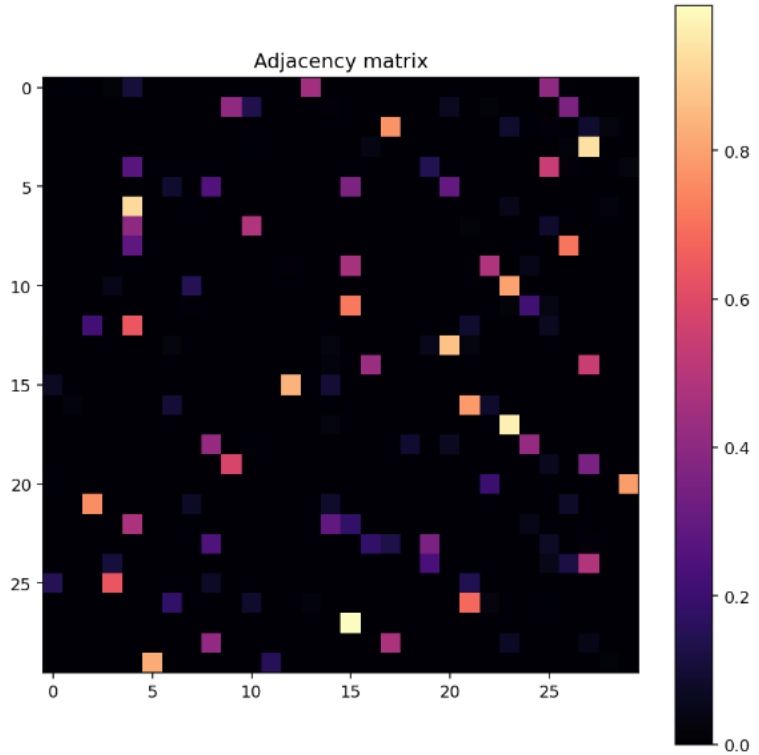
7-degree rotation matrix



Stable point: all roads lead to Rome

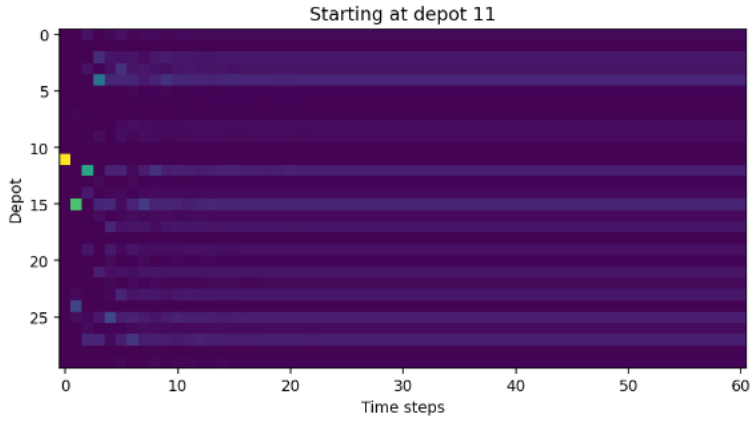
- A simple experiment: Imagine we have a conserving adjacency matrix modelling the flow of packages around a network of depots.

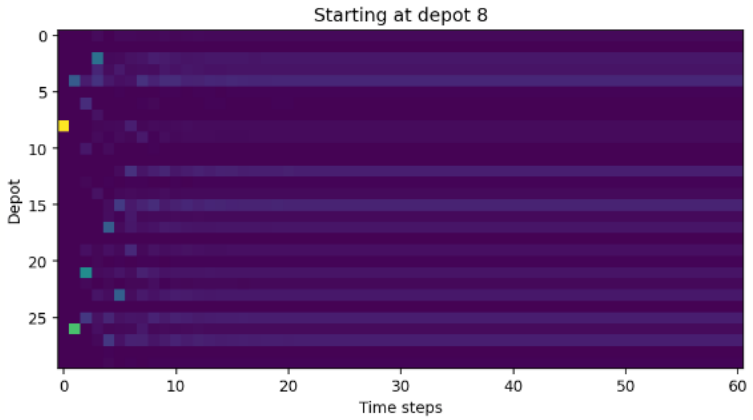
- If we start with different package distributions and then let the "natural" flow begin, what will happen?

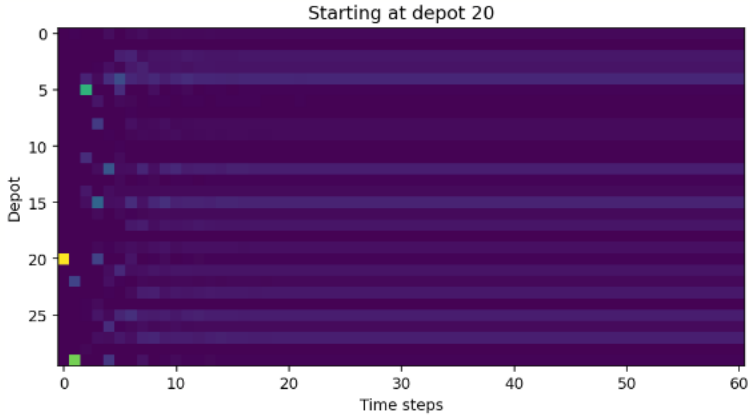


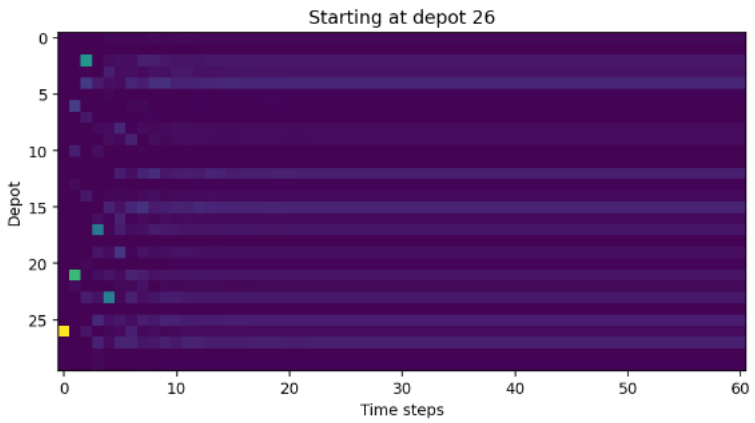
Stable point: all roads lead to Rome

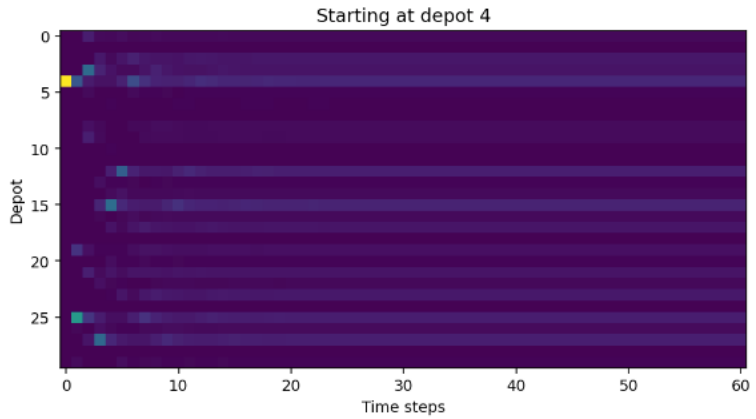
- Run the experiment a few times with 5 different starting states (11, 8, 20, 26, 4)











Surprised...?

Stable point: all roads lead to Rome

- No matter what the initial package distribution is, after several days the distribution will settle down to a **steady state** (or stable point).

- This vector is one of the *eigenvectors* of the adjacency matrix.

- Applying the matrix again does not change the distrbution of the packages, i.e. $\mathbf{x}_{t=60}\, A = \mathbf{x}_{t=59} A$

University of Glasgow

Eigenvalues and eigenvectors

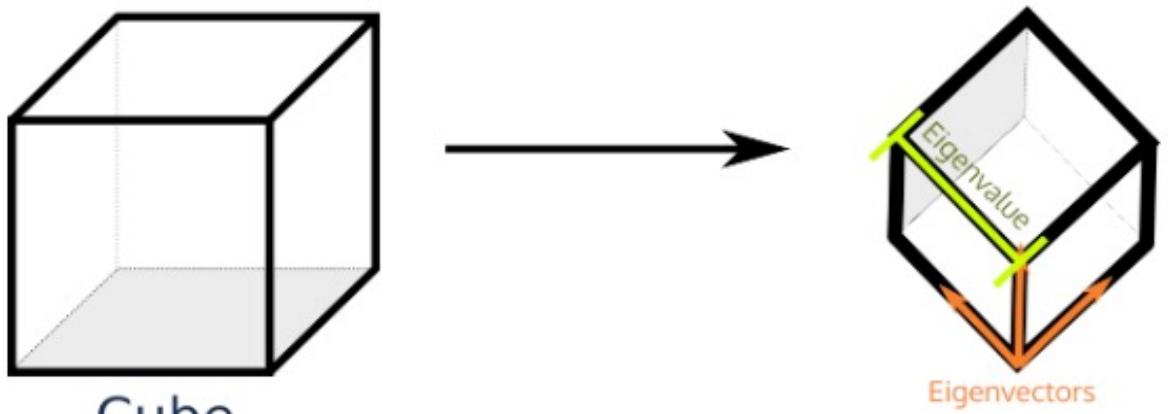
- A matrix represents a special kind of function: a **linear transform**; an operation that performs **rotation** and **scaling** on vectors.

- However, there are certain vectors which don't get rotated when multiplied by the matrix.
  - They only get scaled (stretched or compressed).
  - These vectors are called **eigenvectors**, and they can be thought of as the "fundamental" or "characteristic" vectors of the matrix.
  - The scaling factors that the matrix applies to its eigenvectors are called **eigenvalues**.

University of Glasgow

Eigenvalues and eigenvectors

- We can visualise the effect of a matrix transformation by imagining a parallelepiped (whose edges are vectors) being rotated, stretched and compressed.
- If the edges of the parallelelpiped are the **eigenvectors** of the matrix, the parallelepiped will **only** be stretched or compressed, **not** rotated.
- If the eigenvalue is greater than 1, the eigenvector is stretched. If it's between 0 and 1, it's shrunk.
- If the edges of this parallelepiped have unit length, then after the transformation their lengths will be equal to the eigenvalues.



University of Glasgow

How to find the leading eigenvector: the power iteration method

- "All roads lead to Rome", illustrates how to find one of the eigenvectors of a matrix.
  - We simply apply the matrix repeatedly to a random initial vector and wait until the result converges to a steady state.

$$\begin{aligned}\mathbf{x_n} &= AAAA \ldots AA\mathbf{x_0} \\ &= A^n\mathbf{x_0}\end{aligned}$$

  - this **will generally either explode in value or collapse to zero**. However, we can fix the problem by **normalising** the resulting vector after each application of the matrix:

$$\mathbf{x_n} = \frac{A\mathbf{x_{n-1}}}{\|A\mathbf{x_{n-1}}\|_\infty}$$

  - This process is called **power iteration**.

University of Glasgow

How to find the leading eigenvector: the power iteration method

- Regardless of which vector x_0 we start with (the vector above is chosen randomly), the power iteration method always approaches a fixed vector (though possibly with sign flips).
  - This is true for almost every square matrix.
  - The vector that results from power iteration is known as the **leading eigenvector**.
- It satisfies the definition of an eigenvector because the matrix ***A*** performs only scaling on this vector (no rotation).
  - The scaling effect is eliminated by the normalisation step in the power iteration, but any other effects pass through
  - We can write the scaling effect of the ***A*** on an eigenvector ***x*** as follows:

$$\boldsymbol{Ax=\lambda x}$$

where $\boldsymbol{\lambda}$ is the eigenvalue.

University of Glasgow

Computing eigenvectors and eigenvalues with Numpy

- The power iteration method enables us to calculate the leading eigenvector and eigenvalue, but if we want to know **all** the (linearlly independent) eigenvectors and eigenvalues of a matrix, we can use ***np.linalg.eig***:

- For very large matrices, if you just want to compute the leading eigenvector, power iteration is much faster than using ***np.linalg.eig.***

```
# Compute eigenvalues and eigenvectors with Numpy
print("A=\n%s\n" % A)
evals, evecs = np.linalg.eig(A)  # evecs is a matrix whose
#columns are the unit eigenvectors
print("Eigenvalues:\n", evals)
print()
print("Unit Eigenvectors (columns):\n", evecs)
```

```
A=
[[-1.2011307   1.68673513  2.5605476 ]
 [ 1.68673513  1.70214986  1.81661604]
 [ 2.5605476   1.81661604  0.09588057]]

Eigenvalues:
 [ 4.37541523 -3.20636073 -0.57215476]

Unit Eigenvectors (columns):
 [[-0.46584035 -0.80396258  0.36964433]
 [-0.67928143  0.05720192 -0.73164518]
 [-0.56707099  0.59192238  0.57276382]]
```

University of Glasgow

Eigenvectors and eigenvalues

- Consider a vector function $\boldsymbol{f(x)}$. There may exist vectors such that $\boldsymbol{f(x)=\lambda x}$. The function maps these vectors to scaled versions of themselves. No rotation or skewing is applied, just pure scaling.
- Any square matrix $\boldsymbol{A}$ represents a function $\boldsymbol{f(x)}$ and may have vectors like this, such that

$$\boldsymbol{Ax_i=\lambda_i x_i}$$

- Each vector $\boldsymbol{x_i}$ satisfying this equation is known as an **eigenvector** and each corresponding factor $\boldsymbol{\lambda_i}$ is known as an **eigenvalue**.
  - For any matrix, the **eigenvalues** are uniquely determined, but the eigenvectors are not.
  - The eigenvectors are **orthogonal**, i.e. the dot product of any pair of eigenvectors is zero.

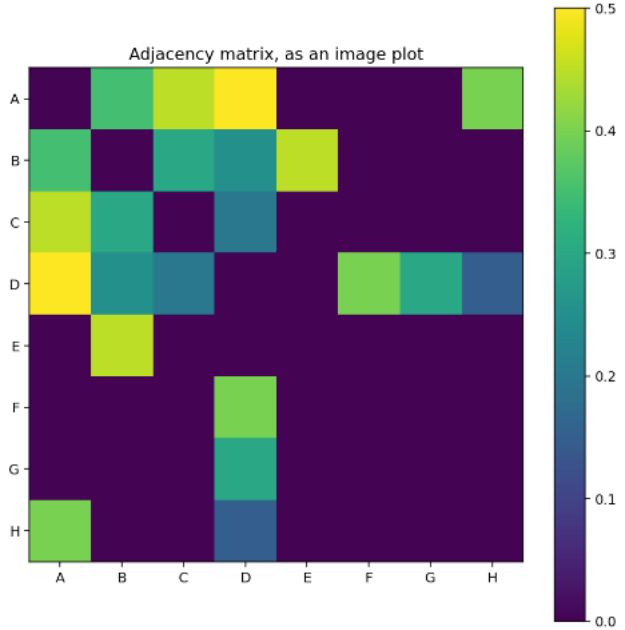
Eigenproblems

- **Eigenproblems** are problems that can be tackled using eigenvalues and eigenvectors.

- **The eigendecomposition:** revisiting the package distribution problem
  - What about at time infinity $x_{t=\infty}$? What is the long term behaviour? Will the system reach a steady state (an **equilibrium**)? Or will it oscillate forever?

Revisiting the package distribution problem

- We will work with an undirected (symmetric) graph to ensure that the eigenvalues are all real (why?)
  - Making the adjacency matrix symmetric also means we don't have to worry about transposing it before using ***np.linalg.eig***

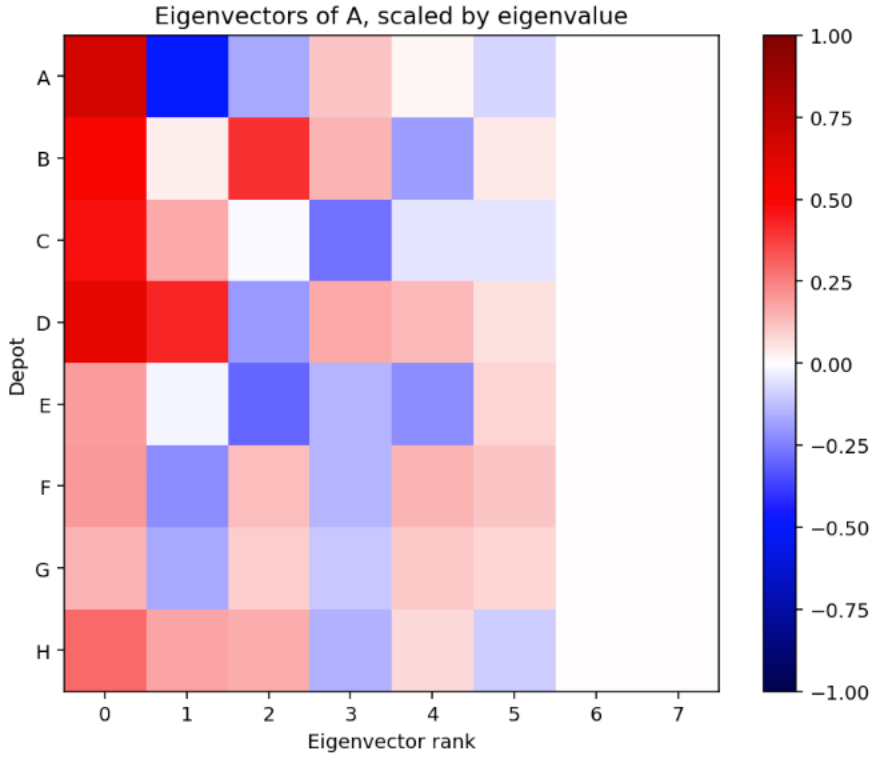


Revisiting the package distribution problem

- compute eigendecomposition of A

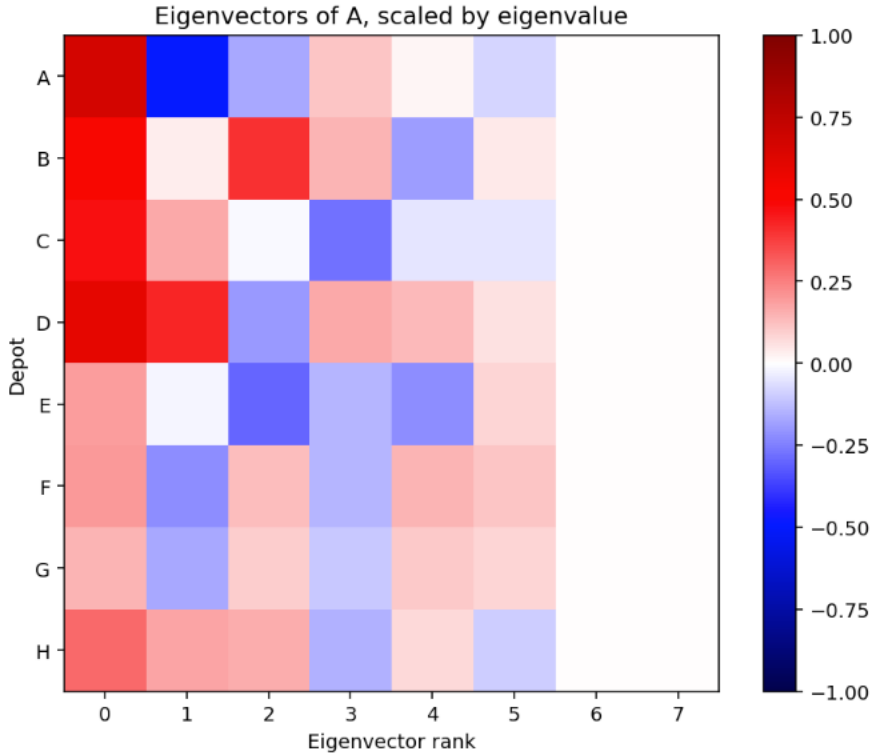
```
# compute eigendecomposition of A
evals, evecs = np.linalg.eig(A)
print(evals)

# plot eigenvectors as an image, in order of decreasing eigenvalue
fig = plt.figure(figsize=(8, 6))
ax = fig.add_subplot(1,1,1)
order = np.argsort(-np.abs(evals))
img = ax.imshow(((evecs[:, order] * evals[order])), cmap='seismic', vmin=-1, vmax=1)
ax.set_yticklabels("ABCDEFGH")
ax.set_yticks(np.arange(8))
ax.set_xlabel("Eigenvector rank")
ax.set_ylabel("Depot")
fig.colorbar(img)#, orientation='horizontal')
ax.set_title("Eigenvectors of A, scaled by eigenvalue");
```

```
[ 1.20818368e+00 -7.51883610e-01 -6.07983833e-01 -4.59054268e-01
  2.24475250e-01  3.86262784e-01 -1.35946555e-17  1.83665339e-17]
```



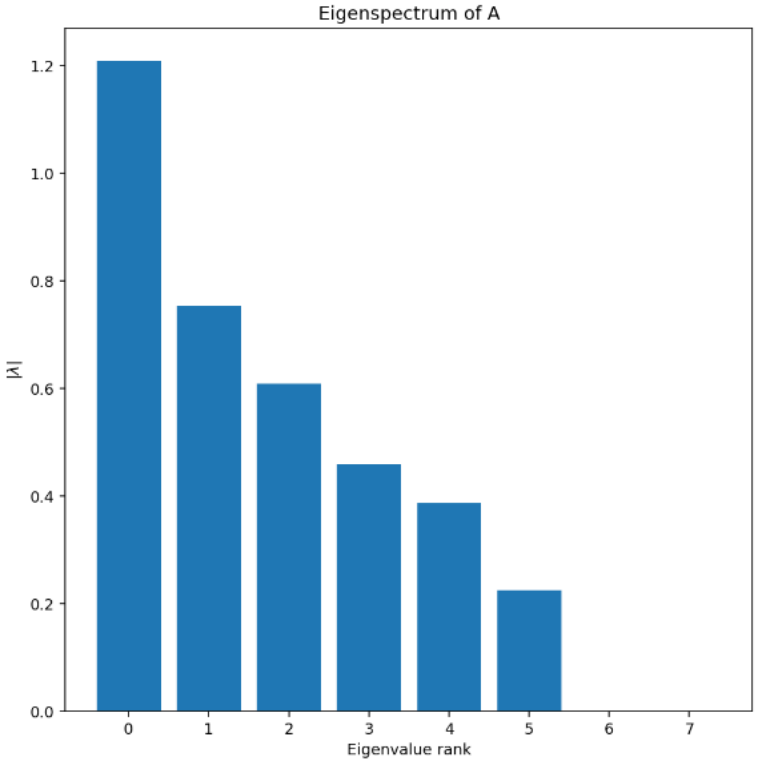
Revisiting the package distribution problem

- The leading eigenvector tells us what the steady state of the package distribution will be if we keep the system running for long enough.
- Small eigenvalues correspond to route patterns that are rarely used, and large eigenvalues to the dominant path of packages moving through the system.
- For this distribution network, most of the packages will pile up at A, B, C, D and H.



The eigenspectrum

- The eigenspectrum is just the sequence of absolute eigenvalues, ordered by magnitude $|\lambda_1|>|\lambda_2|>\cdots>|\lambda_n|$.
- This *ranks* the eigenvectors in order of "importance".
  - This can be useful in finding "simplified" versions of linear transforms.



University of Glasgow

Numerical instability of eigendecomposition algorithms

- *np.linalg.eig* can suffer from numerical instabilities due to rounding errors resulting from limitations on floating point precision.
- This means that sometimes the smallest eigenvectors are not completely orthogonal.

- If your matrix satisfies certain special conditions, you might be able to use a more stable algorithm. For example, if it is real and symmetric (or Hermitian, in the case of a complex matrix), you can use *np.linalg.eigh*.
  - a **Hermitian matrix** is a complex square matrix that is equal to its own conjugate transpose—that is, the element in the i-th row and j-th column is equal to the complex conjugate of the element in the j-th row and i-th column, for all indices i and j

University of Glasgow

np.linalg.eig vs np.linalg.eigh

- np.linalg.eig
- np.linalg.eigh

numpy.linalg.eig

`linalg.eig(a)` [source]

Compute the eigenvalues and right eigenvectors of a square array.

Parameters: **a** : *(..., M, M) array*
Matrices for which the eigenvalues and right eigenvectors will be computed

Returns: **A namedtuple with the following attributes:**

eigenvalues : *(..., M) array*
The eigenvalues, each repeated according to its multiplicity. The eigenvalues are not necessarily ordered. The resulting array will be of complex type, unless the imaginary part is zero in which case it will be cast to a real type. When *a* is real the resulting eigenvalues will be real (0 imaginary part) or occur in conjugate pairs

eigenvectors : *(..., M, M) array*
The normalized (unit "length") eigenvectors, such that the column `eigenvectors[:,i]` is the eigenvector corresponding to the eigenvalue `eigenvalues[i]`.

Raises: **LinAlgError**
If the eigenvalue computation does not converge.

numpy.linalg.eigh

`linalg.eigh(a, UPLO='L')` # [source]

Return the eigenvalues and eigenvectors of a complex Hermitian (conjugate symmetric) or a real symmetric matrix.

Returns two objects, a 1-D array containing the eigenvalues of *a*, and a 2-D square array or matrix (depending on the input type) of the corresponding eigenvectors (in columns).

Parameters: **a** : *(..., M, M) array*
Hermitian or real symmetric matrices whose eigenvalues and eigenvectors are to be computed.

UPLO : *{'L', 'U'}, optional*
Specifies whether the calculation is done with the lower triangular part of *a* ('L', default) or the upper triangular part ('U'). Irrespective of this value only the real parts of the diagonal will be considered in the computation to preserve the notion of a Hermitian matrix. It therefore follows that the imaginary part of the diagonal will always be treated as zero.

Returns: **A namedtuple with the following attributes:**

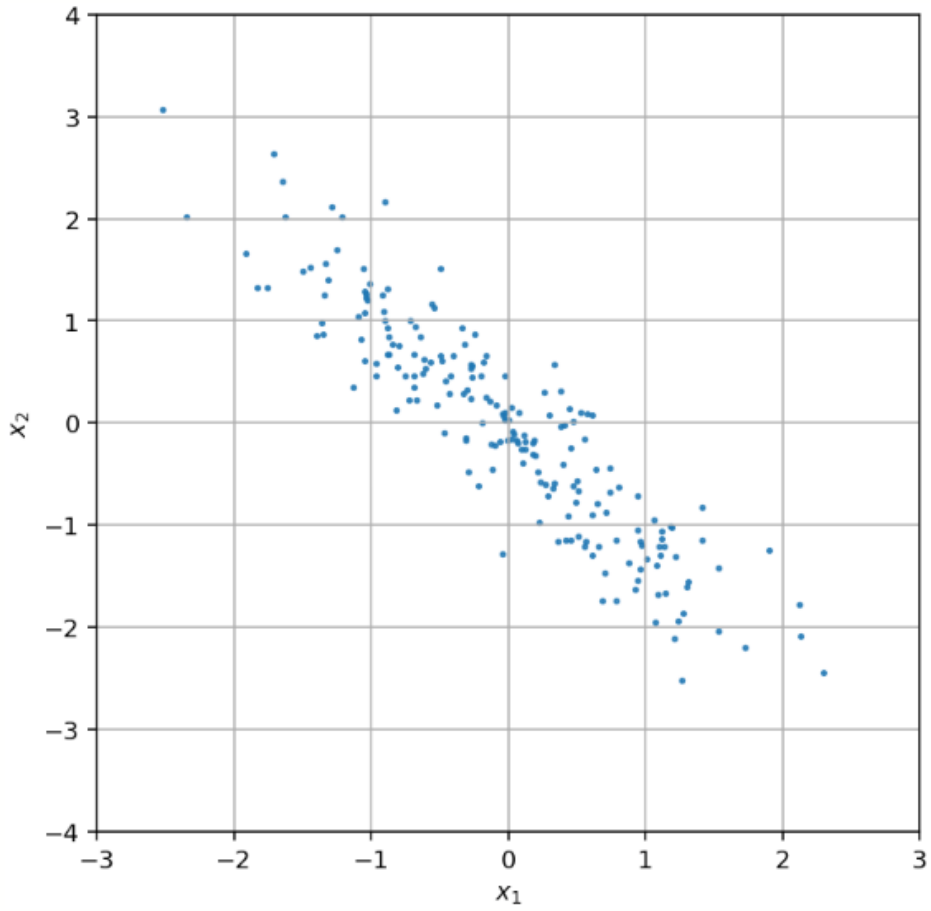
eigenvalues : *(..., M) ndarray*
The eigenvalues in ascending order, each repeated according to its multiplicity.

eigenvectors : *{(..., M, M) ndarray, (..., M, M) matrix}*
The column `eigenvectors[:, i]` is the normalized eigenvector corresponding to the eigenvalue `eigenvalues[i]`. Will return a matrix object if *a* is a matrix object.

Principal Component Analysis (PCA) - analysing data with linear matrices

- Let's generate a dataset in $\mathbb{R}^2$:
  - 200 examples
  - First 10 observations

```
[[-1.36401964  0.97382507]
 [-0.67846501  0.94091341]
 [-0.26893673  0.56671007]
 [ 0.32968114 -0.63947773]
 [ 0.63723353 -0.45517774]
 [-0.26585839  0.44739262]
 [ 0.33394489 -0.59711092]
 [ 0.9658487  -1.16179343]
 [ 0.9424511  -0.71066843]
 [ 0.7834083  -1.74599991]]
```



University of Glasgow

Principal Component Analysis (PCA) - analysing data with linear matrices

- **Mean vector：** The mean vector is the **geometric centroid** of a set of vectors and can be thought of as capturing "centre of mass" of those vectors.

$$\mathrm{mean}(\mathbf{x_1}, \mathbf{x_2}, \ldots, \mathbf{x_n}) = \frac{1}{N} \sum_i \mathbf{x_i}$$

- **Covariance matrix：** measures the spread of a dataset.

$$\Sigma_{ij} = \frac{1}{N-1} \sum_{k=1}^{N} (X_{ki} - \mu_i)(X_{kj} - \mu_j)$$

University of Glasgow

Principal Component Analysis (PCA)

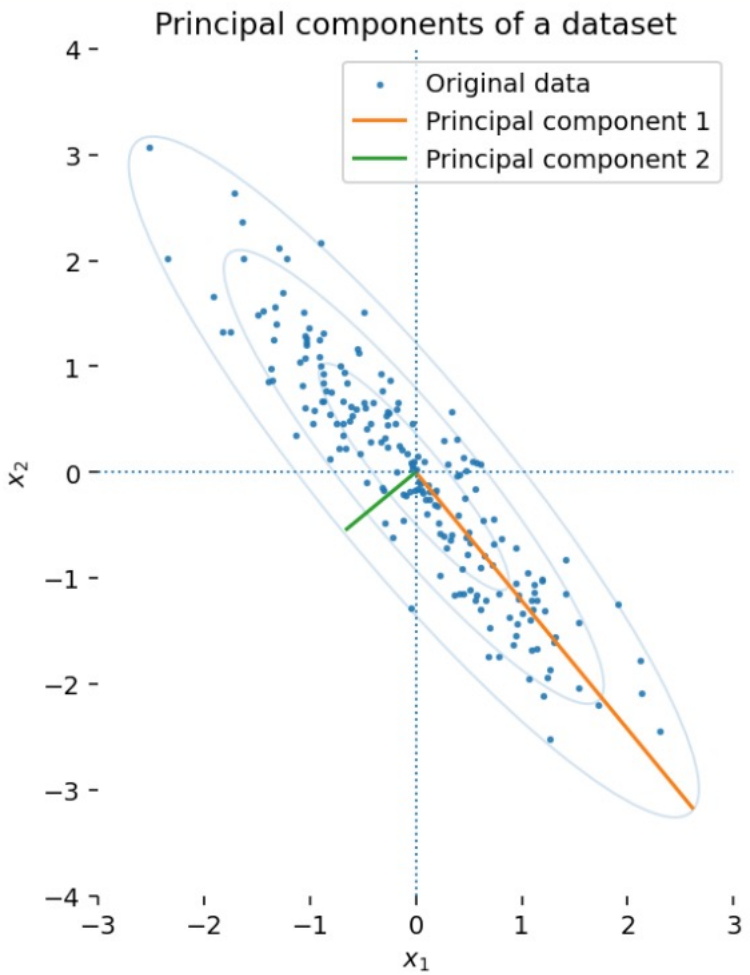
Process:

1. Standardise the data (mean = 0, standard deviation = 1). (Optional, but recommended)
2. Compute the covariance matrix.
3. Perform eigendecomposition on the covariance matrix.
4. Sort eigenvalues in descending order and choose the top k eigenvectors, where k is the number of dimensions you want to keep.
5. Transform the original data using these k eigenvectors to get the principal components.

See lecture notes for an example.

Principal Component Analysis (PCA)

- The eigenvectors of the covariance matrix are called the **principal components**
  - They tell us the directions in which the data varies most
- This is **an incredibly useful thing to be able to do**, particularly with high-dimensional data sets where the variables may be correlated in complicated ways.



Eigendecomposition: Reconstruction of the covariance matrix from its eigenvectors and eigenvalues

- We are now interested in reconstruct the covariance matrix using the eigenvectors and values.

$$\Sigma = Q\Lambda Q^T$$

where Q is a matrix of unit eigenvectors $\mathbf{x_i}$ (same as the output np.linalg.eig) and Λ is a diagonal matrix of eigenvalues (λ_i on the diagonal, zero elsewhere).

University of Glasgow

Eigendecomposition: Reconstruction of the covariance matrix from its eigenvectors and eigenvalues

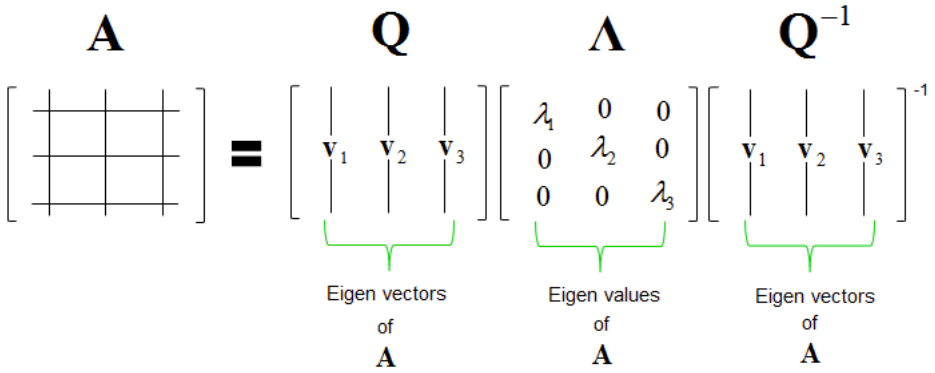
- An experiment on $\boldsymbol{Q\Lambda Q}^T$

```
print("Eigenvectors =\n%s\n" % evecs)
print("Eigenvalues =\n%s\n" % evals)
xx = np.cov(x, rowvar=False)
print(xx)
evals, evecs = np.linalg.eig(np.cov(x, rowvar=False))
reconstructed_A = evecs @ np.diag(evals) @ evecs.T
print_matrix("Q", evecs)
print_matrix("\Lambda", np.diag(evals))

# Display reconstructed covariance matrix for comparison with original covariance matrix
# (see previous cell)
print_matrix("\Sigma = Q\Lambda Q^T", reconstructed_A)
```

```
Eigenvectors =
[[-0.77184948  0.6358053 ]
 [-0.6358053  -0.77184948]]

Eigenvalues =
[0.07933228 1.87499022]

[[ 0.80522408 -0.88121193]
 [-0.88121193  1.14909842]]
Q
 [[-0.77  0.64]
 [-0.64 -0.77]]
\Lambda
 [[0.08 0.  ]
 [0.   1.87]]
\Sigma = Q\Lambda Q^T
 [[ 0.81 -0.88]
 [-0.88  1.15]]
```

Eigendecomposition

- **Eigendecomposition** is the factorisation of a matrix into a canonical form, whereby the matrix is represented in terms of its eigenvalues and eigenvectors.



- Decomposition does **NOT** result in a *compression* of the matrix; **instead**, it breaks it down into constituent parts to make certain operations on the matrix easier to perform.

University of Glasgow

Approximating a matrix

- Imagine we started with a very high dimensional data set, so $\boldsymbol{A}$ is a very large matrix.
  - It's so large, we don't want to store it in memory.

- Instead, we just want to store the first few principal components Q and use these to reconstruct an *approximation* to $\boldsymbol{A}$. Providing we keep the largest principal components, we will probably retain most of the information.

- Matrix approximation can be used to simplify transformations or to compress matrices for data transmission.

University of Glasgow

Eigendecomposition VS Eigenspectrum

- **Eigendecomposition**: factorisation of a matrix into a canonical form
- **Eigenspectrum**: the sequence of absolute eigenvalues, *ranks* the eigenvectors in order of "importance”
  - One large eigenvalue and many small ones - just one vector might approximate this matrix.
  - All eigenvalues similar magnitude? We will not be able to approximate this transform easily.

University of Glasgow

Eigendecomposition VS PCA

- **Eigendecomposition** is the factorisation of a matrix into its eigenvalues and eigenvectors. Specifically, for a square matrix.
  - decompose a matrix into its constituent eigenvalues and eigenvectors.
  - *a general matrix decomposition method*

- **PCA** is a statistical procedure that uses eigendecomposition (or singular value decomposition) to convert correlated features (**covariance matrix**) into a set of linearly uncorrelated features called principal components.
  - *a data analysis technique*
  - Dimensionality reduction
  - Noise reduction and data visualisation.
  - discover which dimensions (features) capture the most variance in the data.

University of Glasgow

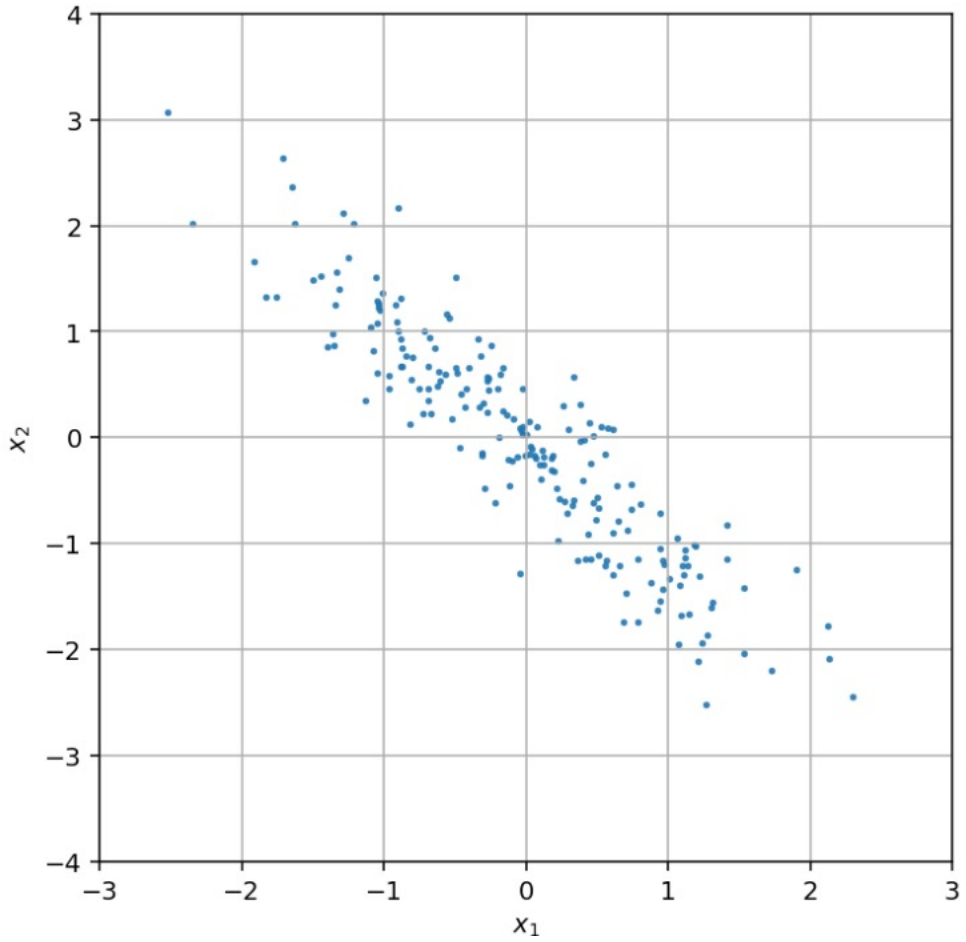
Dimensionality reduction

- We can also reduce the dimensionality of our original dataset by projecting it onto the few principal components of the covariance matrix that we've kept

- We can do this by multiplying the dataset matrix by each component and saving the projected data into a new, lower-dimensional matrix.

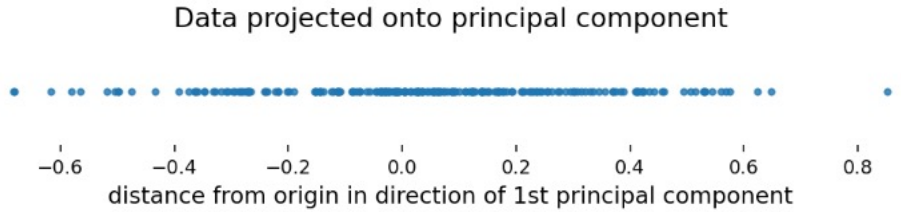
Dimensionality reduction

- Original data points



- Project to its eigenvector

[0.6358053 -0.77184948]



University of Glasgow

Uses of eigendecomposition

- **Matrix decomposition** (e.g. PCA) is an *essential* tool in data analysis.
  - Recommenders, image compression, visualisation etc.
- **Spectral Clustering:** Used for graph-based clustering.
  - The graph Laplacian matrix's eigenvectors are used to cluster data points
- **PageRank Algorithm:** Used by Google's search engine to rank web pages.
  - Relies on finding the eigenvector corresponding to the largest eigenvalue of the modified adjacency matrix of the web graph.
- **Latent Semantic Analysis (LSA):** A text mining technique for topic modeling and dimensionality reduction.
  - Uses singular value decomposition (a generalization of eigendecomposition) of the term-document matrix to capture latent topics.

Matrix properties - and (some) relations to the the eigendecomposition

- **Trace:** The trace of a square matrix can be computed from the sum of its diagonal values:

$$\mathrm{Tr}(A) = a_{1,1} + a_{2,2} + \cdots + a_{n,n}$$

- The **perimeter** of the parallelotope of a unit cube transformed by the matrix
- It is also equal to the sum of the eigenvalues of ***A***

$$\mathrm{Tr}(A) = \sum_{i=1}^{n} \lambda_i$$

- **Determinant:** is equal to the product of the eigenvalues of the matrix.

$$\det(A) = \prod_{i=1}^{n} \lambda_i$$

- It measures how much the space expands or contracts after the linear transform. (the **volume** of the parallelotope)
- If any eigenvalue is 0, det(A)=0.
  - the transformation **cannot be reversed**
  - information has been lost.

Definite and semi-definite matrices

A matrix is called

- **positive definite** if all of its eigenvalues are greater than zero: $\lambda_i > 0$.
- **positive semi-definite** if all of its eigenvalues are greater than or equal to zero: $\lambda_i \geq 0$.
- **negative definite** if all of the eigenvalues are less than zero: $\lambda_i < 0$,
- **negative semi-definite** if all the eigenvalues are less than or equal to zero: $\lambda_i \leq 0$.

- A covariance matrix is **symmetric**, and its entries are based on the variances and covariances of the variables in a dataset.
  - By its very definition, the covariance matrix has properties that make it positive **semi-definite**.

University of Glasgow

Summary of eigenproblems

- Eigenvectors exist only for square matrices.
- A matrix A transforms a general vector by rotating and scaling it. However, the eigenvectors of A are special because they can only be scaled, not rotated by the transform.
- The eigenvalues of A are the scaling factors λ_i that correspond to each unit eigenvector $\mathbf{x_i}$.
- Eigenvectors and eigenvalues can be computed algorithmically (e.g. by the power iteration algorithm for finding the leading eigenvector).
- Eigendecomposition is the process of breaking a matrix down into its constituent eigenvalues and eigenvectors. These serve as a compact summary of the matrix.
- The eigenspectrum is just the list of (absolute) eigenvalues of a matrix, in rank order, largest first.
- If we have a complete set of eigenvectors and eigenvalues, we can reconstruct the matrix.
- We can approximate a large matrix A with a few leading eigenvectors; this is a simplified or truncated approximation to the original matrix.
- If we repeatedly apply a matrix A to some vector $\mathbf{x}$, the vector will be stretched more and more along the largest eigenvectors.

University of Glasgow

Things we can tell from eigenvectors/values

- If a matrix has one or more zero eigenvalues, the transform it performs is one that collapses one or more dimensions in vector space. This type of operation is irreversible, and this tells us that A is singular (un-invertible) - more on that in a moment.
- Eigenvectors corresponding to larger (absolute) eigenvalues are more "important"; they are representing directions in which data will get stretched most.
- If the eigenspectrum is nearly flat (eigenvalues all have similar values), then A represents a transform that stretches vectors almost equally in all directions (like transforming a sphere to a sphere).
- If the eigenspectrum has a few large eigenvalues and lots of small ones, then vectors will get stretched along a few directions, but shrink away to nothing along others (like transforming a sphere to a long, skinny ellipse).

Matrix Inversion

Four basic algebraic operations on matrices:

- scalar multiplication c$\mathbf{A}$;
- matrix addition $\mathbf{A+B}$;
- matrix multiplication $\mathbf{BA}$
- matrix transposition $\mathbf{A}^T$

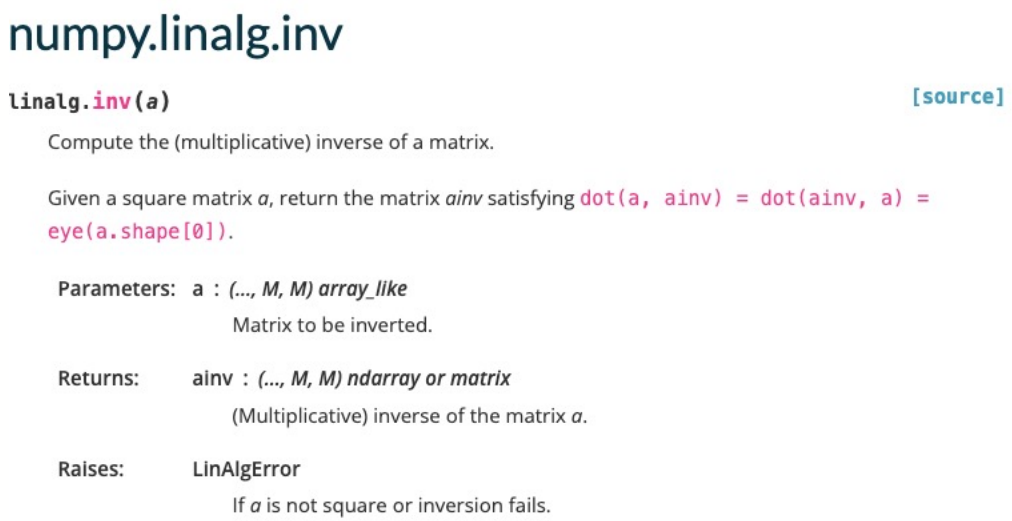
A further important operation: **inversion $\mathbf{A}^{-1}$**, defined such that $\mathbf{A}^{-1}\mathbf{A}=\mathbf{I}$:

- $\mathbf{A}^{-1}(\mathbf{Ax})=\mathbf{x}$,
- $(\mathbf{A}^{-1})^{-1}=\mathbf{A}$
- $(\mathbf{AB})^{-1}=\mathbf{B}^{-1}\mathbf{A}^{-1}$

Inversion is only defined for certain kinds of matrices: ***Square Matrices, Non-singular Matrices***

Computing the inverse of a matrix

- There are many ways to compute the inverse of a matrix. There is a standard recursive algorithm which you may have seen in Maths courses, but this is only useful for very small matrices.
- Instead, we often use the workhorse of matrix decompositions: the **singular value decomposition**, which we will discuss later.
- In the meantime, we can use the NumPy method *np.linalg.inv*, as shown below:



numpy.linalg.inv

`linalg.inv(a)` [source]

Compute the (multiplicative) inverse of a matrix.

Given a square matrix *a*, return the matrix *ainv* satisfying `dot(a, ainv) = dot(ainv, a) = eye(a.shape[0])`.

Parameters: a : *(..., M, M) array_like*
Matrix to be inverted.

Returns: ainv : *(..., M, M) ndarray or matrix*
(Multiplicative) inverse of the matrix *a*.

Raises: LinAlgError
If *a* is not square or inversion fails.

University of Glasgow

Inversion as "undo"

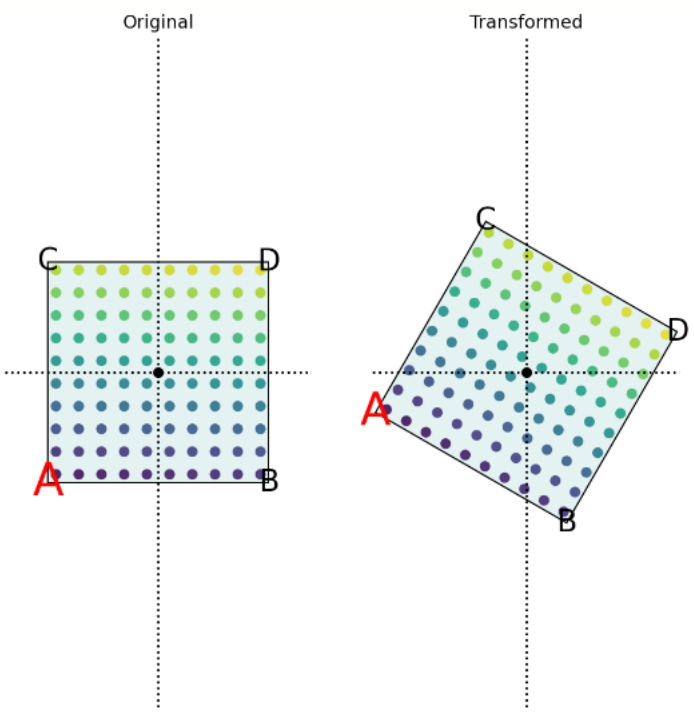
- Inversion of a matrix creates a new linear operator which reverses the original operation. Let's see it in action:

```
# Verify that left-multiplying by the inverse "undoes" the transformation applied by A to a ra
# Create a random vector
x = np.random.normal(0, 1, (3, 1))
print_matrix("x", x)
# Tranform it
print_matrix("Ax", A @ x)
# Left-multiply by the inverse to recover the original vector
print_matrix("A^{-1}(Ax)", np.linalg.inv(A) @ (A @ x))
```

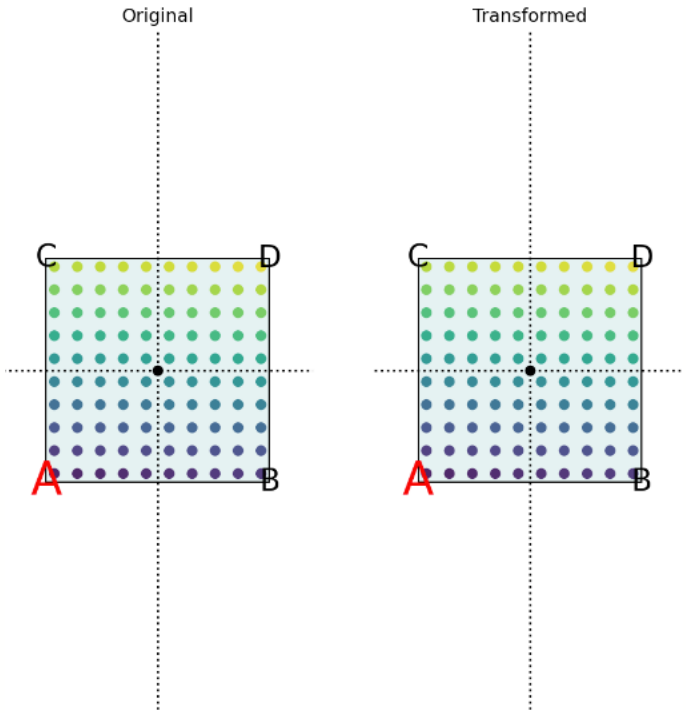
```
x
 [[-0.18]
 [ 0.95]
 [-0.92]]
Ax
 [[-0.28]
 [ 0.73]
 [-1.11]]
A^{-1}(Ax)
 [[-0.18]
 [ 0.95]
 [-0.92]]
```

Inversion as "undo"

- 30 degree rotation matrix



- 30 degree rotation matrix, left-multiplied by its inverse



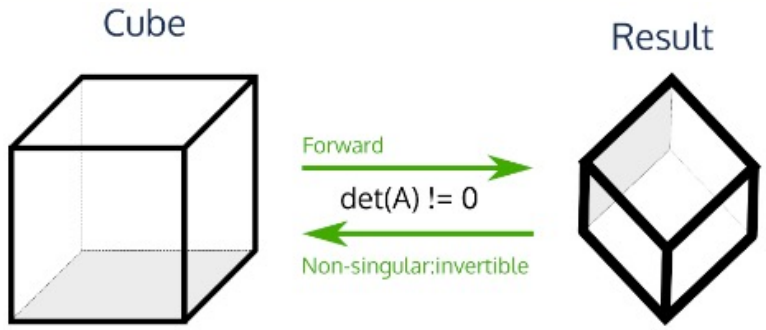
University of Glasgow

BUT only square matrices can be inverted !

- Inversion is only defined for square matrices, representing a linear transform $\mathbb{R}^n \rightarrow \mathbb{R}^n$.
- This is equivalent to saying that the determinant of the matrix must be non-zero: det(A)≠0. why?
  - It must be a bijective (both injective and surjective) transformation. In simpler terms, each input vector must map to a unique output vector,
  - If det(A)=0, then the transformation A collapses at least one dimension, which means it's not bijective.

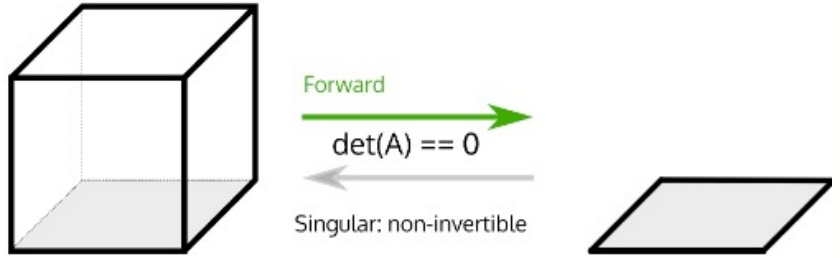
Singular and non-singular matrices

- A matrix with det(A)=0 is called **singular** and has no inverse.
- A matrix which is invertible is called **non-singular**.



Geometric intuition

- One of the dimensions of the parallelepiped has been squashed to nothing at all.
- It is impossible to reverse the transformation, because information was lost in the forward transform.



University of Glasgow

Matrix inversion algorithms

- **Matrix operations** involve *lots* of repeated floating point operations (roundoff to accumulate)

- Inversion is particularly hard to compute in a **stable** form directly, and many matrices that *theoretically* could be inverted cannot be inverted using floating point representation.

Time complexity

- Matrix inversion, for a general $n \times n$ matrix, takes $O(n^3)$ time. It is *provable* that no general matrix inversion algorithm can ever be faster than $O(n^3)$

University of Glasgow

Matrix inversion algorithms

There are many special kinds of matrices for which much faster inversion is possible

- **orthogonal** matrix (rows and columns are all orthogonal unit vectors): $O(1)$, $A^{-1}=A^T$
- **diagonal** matrix (all non-diagonal elements are zero): $O(n)$, $A^{-1}=1/A$ (i.e. the reciprocal of the diagonal elements of A).
- **positive-definite** matrix: $O(n^2)$ via the ***Cholesky decomposition***. We won't discuss this further.
- **triangular** matrix (all elements either above or below the main diagonal are zero): $O(n^2)$, trivially invertible by **elimination algorithms**.

University of Glasgow

Matrix inversion algorithms

- The inverse of a **sparse matrix** is in general **not sparse**; it will (most likely) be dense.
  - the process of inversion inherently involves combinations of all rows and columns
  - interplay between the non-zero elements, during inversion, can lead to non-zero values in many positions of the inverse matrix

- A sparse matrix could easily be 1,000,000 x 1,000,000, but with maybe only a few million non-zero entries, and might be stored in a few dozen megabytes.
  - The inverse form would have 1,000,000,000,000 entries and require a **terabyte** or more to store!

University of Glasgow

Revisting the package distribution problems

Problem: “predicting” the past with inversion

- We can use **inversion** to solve the problem of predicting the distribution at $\mathbf{x}_{t=-1}$ given $\mathbf{x}_{t=0}$.

$$\mathbf{x}_{t=-1} = A^{-1}\mathbf{x}_{t=0}$$

- We can compute any negative power of the matrix to "undo" any number of steps:

$$A^{-k} = \underbrace{A^{-1}A^{-1}A^{-1}\dots A^{-1}}_{\text{k repetitions}}$$

Linear systems

- Imagine a system where there is an input, ***x***, and an output, ***y***.

$$f(x) = Ax = y$$

$$A = \begin{bmatrix} 0.5 & 1.0 & 2.0 \\ 1.0 & 0.5 & 0.0 \\ 0.6 & 1.1 & -0.3 \end{bmatrix}$$

$$\begin{aligned} y_1 &= 0.5x_1 + 1.0x_2 + 2.0x_3 \\ y_2 &= 1.0x_1 + 0.5x_2 + 0.0x_3 \\ y_3 &= 0.6x_1 + 1.1x_2 - 0.3x_3 \end{aligned}$$

This is a **linear system** or **linear system of equations**

Task: Given ***y,*** "predicting" the past **x**

Solving linear systems

- The solution of linear systems is apparently simple for cases where ***A*** is ***square***.
- If ***Ax=y***, then left-multiplying both sides by $\boldsymbol{A}^{-1}$ we get

$$A^{-1}A\mathbf{x} = A^{-1}\mathbf{y}$$
$$I\mathbf{x} = A^{-1}\mathbf{y}$$
$$\mathbf{x} = A^{-1}\mathbf{y}$$

- This only works for square matrices, as $\boldsymbol{A}^{-1}$ is not defined for non-square matrices. This means that $\mathbf{x}$ and $\mathbf{y}$ must have the same number of dimensions.

University of Glasgow

Approximate solutions for linear systems

- In practice, linear systems are almost never solved with a direct inversion.
  - Inversion is $O(N^3)$
  - ***Nonsquare*** *in nature*

- Instead, linear systems are typically solved iteratively, either using specialised algorithms based on knowledge of the structure of the system, or using **optimisation**, which will be the topic of the next Unit (lecture 4).
  - to find solutions that minimise $\|A\mathbf{x}-\mathbf{y}\|_2$ by adjusting the value of $\mathbf{x}$ repeatedly.

University of Glasgow

Singular value decomposition

- Eigendecompositions only apply to **diagonalizable** matrices; which are a subset of **square matrices**.
- There are many problems which have **non-square** matrices which we would like to be able to decompose.
- The **singular value decomposition** (SVD) is a general approach to decomposing any matrix $\boldsymbol{A}$. It is the powerhouse of computational linear algebra.
- The SVD produces a decomposition which splits ***ANY*** matrix up into three matrices:

$$\boldsymbol{A=U\Sigma V^T}$$

Singular value decomposition

- The SVD produces a decomposition which splits ***ANY*** matrix up into three matrices:

$$A=U\Sigma V^T$$

where

- A is any m×n matrix,
- U is a **square unitary** m×m matrix, whose columns contain the **left singular vectors**,
- V is an **square unitary** n×n matrix, whose columns contain the **right singular vectors**,
- Σ is a diagonal m×n matrix, whose diagonal contains the **singular values**.

If A is real, then U and V will be **orthogonal** matrices ($U^T=U^{-1}$), whose rows all have unit norm and whose columns also all have unit norm.

University of Glasgow

eigendecomposition vs SVD

- **Eigendecomposition**: Can only be applied to square matrices.
- **SVD**: Can be applied to any matrix (square or rectangular).
- If A is a symmetric positive semi-definite matrix (like a covariance matrix), then the eigendecomposition and SVD give related results.

The SVD is the same as:

- taking the eigenvectors of $\boldsymbol{A^TA}$ to get $\boldsymbol{U}$
  - NB: $\boldsymbol{A}$ @ $\boldsymbol{A.T}$ is a symetric matrix
- taking the square root of the *absolute* value of the eigenvalues $\boldsymbol{\lambda_i}$ of $\boldsymbol{A^TA}$ to get $\boldsymbol{\Sigma_i} = \sqrt{|\boldsymbol{\lambda_i}|}$taking the eigenvectors of $\boldsymbol{AA^T}$ to get $\boldsymbol{V^T}$

University of Glasgow

SVD decomposes any matrix into three matrices with special forms

- The SVD produces a decomposition which splits ***ANY*** matrix up into three matrices:

$$A=U\Sigma V^T$$

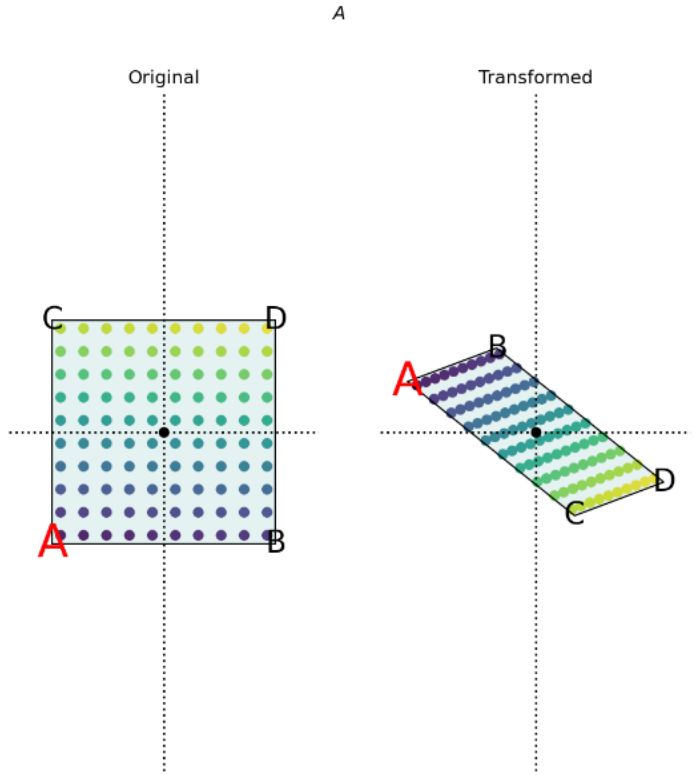
Special forms of matrices, like orthogonal matrices and diagonal matrices, are much easier to work with than general matrices. This is the power of the SVD.

- **U** is orthogonal, so is a pure rotation matrix,
- **Σ** is diagonal, so is a pure scaling matrix,
- **V** is orthogonal, so is a pure rotation matrix.

Rotate, scale, rotate

- *The SVD splits any matrix transformation into a rotate-scale-rotate operation.*

```
A
 [[ 0.4   0.75]
 [ 0.15 -0.6 ]]
```




```
: u, sigma, vt = np.linalg.svd(A)

  show_matrix_effect(vt, "V^T")
  show_matrix_effect(np.diag(sigma) @ vt, "\Sigma V^T")
  show_matrix_effect(u @ np.diag(sigma) @ vt, "U \Sigma V^T")

  #Sanity
  print("Determinant of vt:" ,np.linalg.det(vt))
  print("Determinant of u:" ,np.linalg.det(u))
  print("Determinant of Sigma:" ,np.linalg.det(np.diag(sigma)))
```

```
V^T
 [[-0.26 -0.97]
 [ 0.97 -0.26]]
\Sigma V^T
 [[-0.25 -0.96]
 [ 0.34 -0.09]]
U \Sigma V^T
 [[ 0.4   0.75]
 [ 0.15 -0.6 ]]
Determinant of vt: 1.0
Determinant of u: -1.0
Determinant of Sigma: 0.3525
```

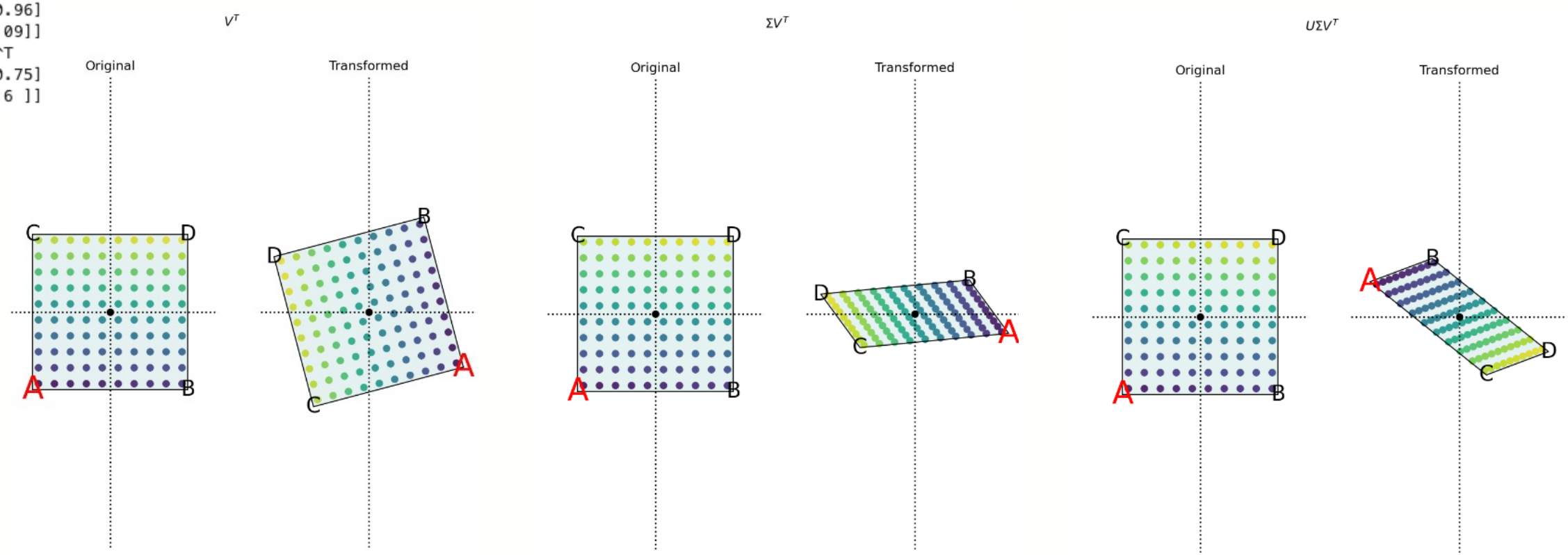
Rotate, scale, rotate

```
V^T
 [[-0.26 -0.97]
 [ 0.97 -0.26]]
\Sigma V^T
 [[-0.25 -0.96]
 [ 0.34 -0.09]]
U \Sigma V^T
 [[ 0.4   0.75]
 [ 0.15 -0.6 ]]
```

Rotate

Scale

Rotate



Rotate, scale, rotate

$A=U\Sigma V^T$: U is orthogonal, so is a pure rotation matrix, Σ is diagonal, so is a pure scaling matrix, V is orthogonal, so is a pure rotation matrix.

Why does an orthogonal matrix only do pure rotation?

- **Orthogonal Matrix：** $U^{-1}= U^T$, $V^{-1}= V^T$
- **Preservation of Length**
  - $\|x\|_2=x^Tx$, $U^TU = U^{-1}U = I$
  - $\|Ux\|_2=(Ux)^T(Ux)=x^TU^TUx=x^Tx = \|x\|_2$
  - **This equation means the length of x is preserved after the transformation.**
- **Preservation of Angles**:
  - $x\cdot y=x^Ty$
  - $(Ux)\cdot(Uy)=(Ux)^T(Uy)=x^TU^TUy=x^Ty$
  - Since the dot product remains unchanged, the angle between the vectors is preserved.

University of Glasgow

Using the SVD

- **Fractional powers:** We can use the SVD to compute interesting matrix functions like the square root of a matrix $\mathbf{A}^{1/2}$.
- **Invert a matrix:** $\mathbf{A}^{-1}$, which will "undo" the operation.

- The rule is simple: to do any of these operations, ignore U and V (which are just rotations), and apply the function to the singular values elementwise:

$$A^n = V\Sigma^n U^T$$

University of Glasgow

Inversion - relation to SVD

Invert a matrix: A^{-1}, which will "undo" the operation.

$$A^{-1} = V\Sigma^{-1}U^T$$

- This can be computed in O(n) time because $\boldsymbol{\Sigma}^{-1}$ can be computed simply by taking the reciprocal of each of the diagonal elements of $\boldsymbol{\Sigma}$.

- *N.B. No free lunch! As a consequence, we now know that computing the SVD must take* $O(n^3)$ *time for square matrices, since inversion cannot be achieved faster than* $O(n^3)$.

Pseudo-inverse (non-symmetric)

- We can also pseudo-invert a matrix: $A+$, which will approximately "undo" the operation, even when A isn't square.

- The pseudo-inverse of A is just

$$A^{+} = V\Sigma^{-1}U^{T}$$

which is the same as the standard inverse computed via SVD, but taking care that Σ is the right shape - appropriate zero padding is required!

Fortunately, this is taken care of by the Numpy method ***np.linalg.pinv***.

University of Glasgow

A few more matrix properties

Rank of a matrix

- The **rank** of a matrix is equal to the number of non-zero singular values.
- If the number of non-zero singular values is equal to the size of the matrix, then the matrix is **full rank**.
- A full rank matrix has a non-zero determinant and will be invertible.
- The rank tells us how many dimensions the parallelotope that the transform represents will have.
- If a matrix does not have full rank, it is **singular** (non-invertible) and has **deficient rank**.
- If the number of non-zero singular values is much less than the size of the matrix, the matrix is **low rank**.

University of Glasgow

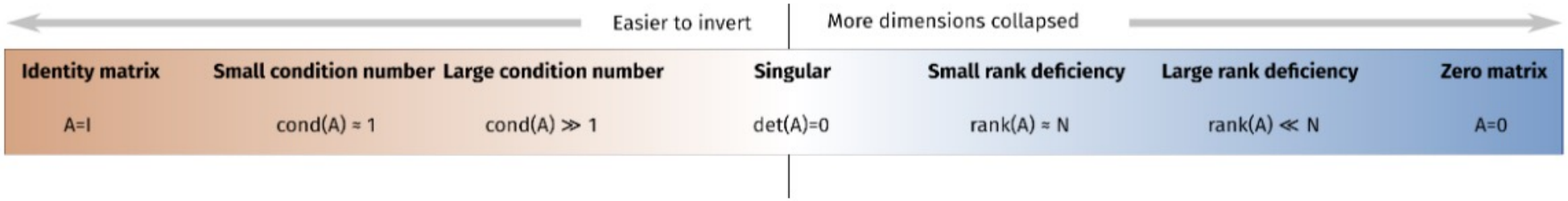
A few more matrix properties

Condition number of a matrix

- The **condition number** number of a matrix is the ratio of the largest singular value to the smallest.
- This is only defined for full rank matrices.
- The condition number measures how sensitive inversion of the matrix is to small changes.
- A matrix with a small condition number is called **well-conditioned** and is unlikely to cause numerical issues.
- A matrix with a large condition number is **ill-conditioned**, and numerical issues are likely to be significant.
- An ill-conditioned matrix is almost singular, so inverting it will lead to invalid results dur to floating point roundoff errors.

Relation to singularity

- A **singular** matrix A is un-invertible and has det(A)=0. Singularity is a binary property, and is either true or false.
- We can think of **rank** as measuring "how singular" the matrix is, i.e. how many dimensions are lost in the transform.
- We can think of the **condition number** as measuring how close a non-singular matrix is to being singular. A matrix which is nearly singular may become effectively singular due to floating point roundoff errors.



University of Glasgow

Applying decompositions

- Whitening a data set (self-study in the lab)
  - **Whitening** removes all linear correlations within a dataset.
  - It is a normalizing step used to standardise data before analysis.
- Whitening does the following:
  - centers the data around its mean, so it has **zero mean**.
  - "squashes" the data so that its distribution is **spherical** and has **unit covariance**.
    - '**spherical**' means that it is roughly spread out evenly in all directions in the space it's plotted in.
    - '**unit covariance**' means the diagonal elements are all 1s. This means each feature (or dimension) has a variance of 1.
      - We can normalise to 1 by: 1.0 / np.sqrt(sigma)

Whitening a data set

1. **Center the Data:** Before applying SVD, you should first center the data by subtracting the mean of each feature.

$$X_{\text{centered}} = X - \text{mean}(X)$$

2. **Apply SVD:** For the centered data X_{centered}, perform SVD:

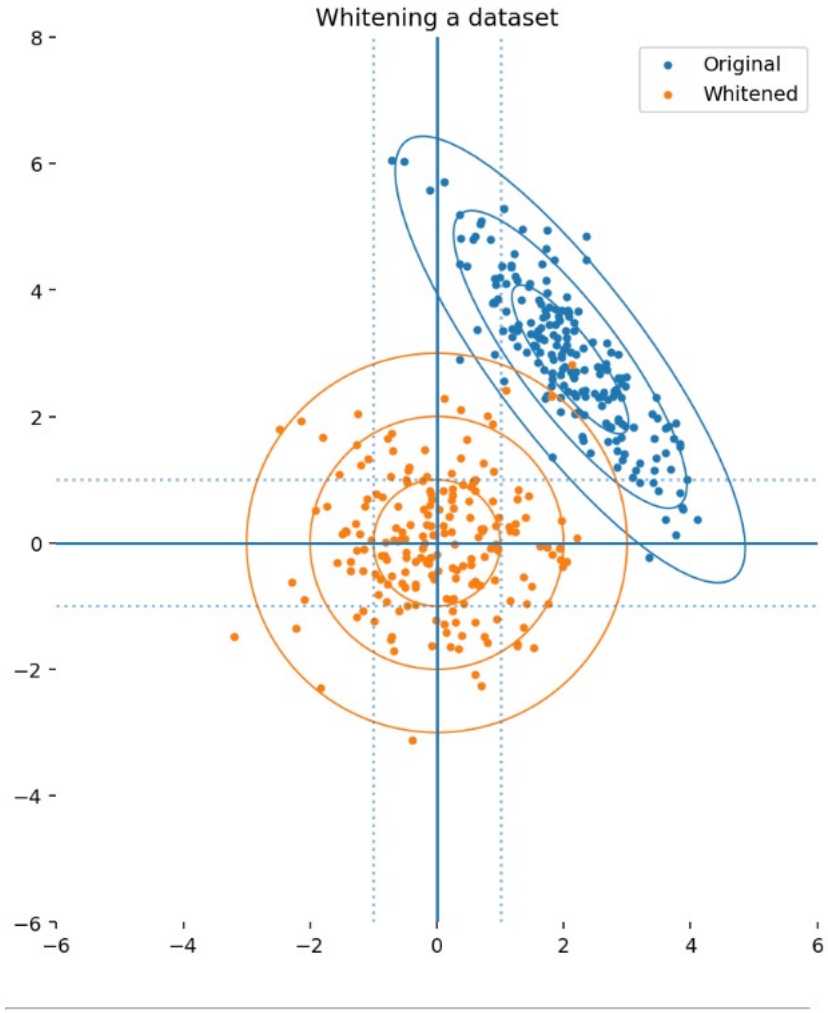
$$X_{\text{centered}} = U\Sigma V^T$$

Where: U contains the left singular vectors.Σ is a diagonal matrix containing the singular values. V^T contains the right singular vectors.

3. **Whitening Transform:** The whitening transform can be computed using U and Σ. Specifically, the whitened data X_{whitened} is given by:

$$X_{\text{whitened}} = U\Sigma^{-1/2}U^T X_{\text{centered}}$$

Here, $\Sigma^{-1/2}$ is a diagonal matrix where each element is the **inverse square root** of the corresponding **singular value** in Σ.



University of Glasgow

PCA vs Whitening a data set

- Whitening can be seen as an extension of the PCA process.
  - Both SVD (more numerically stable) and eigendecomposition can be used in both the tasks
  - But their objectives are different
- Objectives
  - **PCA**: Reduce the dimensionality of data while retaining as much variance as possible.
    - Only keep *k* most significant components.
    - Speed up your unsupervised feature learning algorithm
  - **Whitening**: Transform the data such that it has an identity covariance matrix.
    - No need to find *k* most significant components.
    - An important pre-processing step for many algorithms


University of Glasgow
Thank you
Contact:
Zaiqiao.Meng@Glasgow.ac.uk