Introduction to Data Science and Systems (M)

Lecture Week 3: Computational linear algebra - *linear systems, inversion and matrix decompositions*

University of Glasgow - material prepared by John H. Williamson (adapted to IDSS by BSJ, MZ and NP), v20232024a

Summary

By the end of this unit you should know:

- 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 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

In [1]:
```
import numpy as np
import matplotlib as mpl
from jhwutils.matrices import print_matrix, show_matrix_effect
import matplotlib.pyplot as plt
%matplotlib inline
plt.rc('figure', figsize=(8.0, 8.0), dpi=140)
```

Graphs as matrices

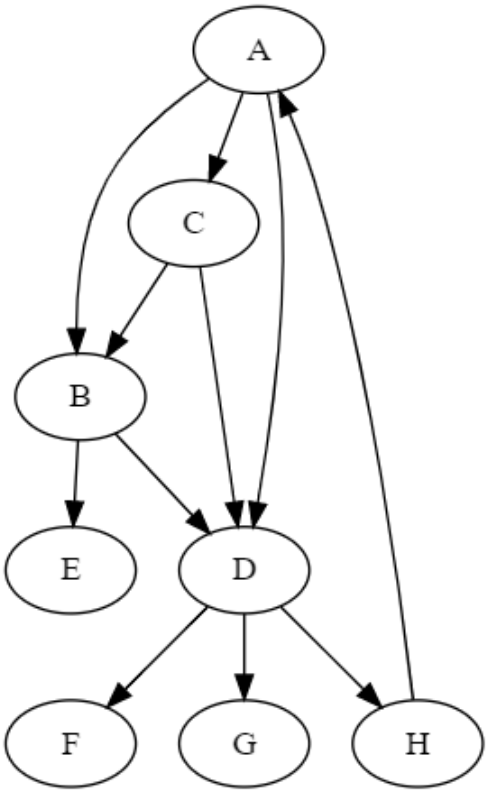
Example: distributing packages

[Image by nSeika shared CC BY].

You run a large logistics company. You have to route packages between distributions centres efficiently, so they will be ready for local delivery. To do this, you need to be able to predict which warehouses are going to receive lots of packages (maybe they are connected to other sites by several direct motorways) and which will receive few packages (maybe they are remote).

How can this problem be modelled? If we can make the assumption that 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 -- then we can model the problem with linear algebra.

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.



The graph above has 8 vertices (A, B, C, D, E, F, G, H) and 11 edges:

$$
\begin{aligned}
A &\to B \\
A &\to C \\
A &\to D \\
B &\to D \\
B &\to E \\
C &\to B \\
C &\to D \\
D &\to F \\
D &\to G \\
D &\to H \\
H &\to A
\end{aligned}
$$

We can write this as an **adjacency matrix**. We number each vertex $0, 1, 2, 3, \ldots$. 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
```

(the letters aren't part of the matrix and are just shown for clarity).

Computing graph properties

There are some graph properties which we can compute easily from this binary matrix:

- 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).

In [2]:
```
# Our adjacency matrix:
adj = np.array([[0, 1, 1, 1, 0, 0, 0, 0],
                [0, 0, 0, 1, 1, 0, 0, 0],
                [0, 1, 0, 1, 0, 0, 0, 0],
                [0, 0, 0, 0, 0, 1, 1, 1],
                [0, 0, 0, 0, 0, 0, 0, 0],
                [0, 0, 0, 0, 0, 0, 0, 0],
                [0, 0, 0, 0, 0, 0, 0, 0],
                [1, 0, 0, 0, 0, 0, 0, 0]])

# compute in-degrees and out-degrees
in_degrees = np.sum(adj, axis=0)
out_degrees = np.sum(adj, axis=1)
print('In degrees: ', list(zip('ABCDEFGH', in_degrees)))
print('Out degrees:', list(zip('ABCDEFGH', out_degrees)))
```

```
In degrees:  [('A', 1), ('B', 2), ('C', 1), ('D', 3), ('E', 1), ('F', 1), ('G', 1), ('H', 1)]
Out degrees: [('A', 3), ('B', 2), ('C', 2), ('D', 3), ('E', 0), ('F', 0), ('G', 0), ('H', 1)]
```

In [3]:
```
# is the graph undirected?
print(np.allclose(adj, adj.T))
```

```
False
```

In [4]:
```
# if we want to *force* our adjacency matrix to be symmetric,
# i.e. convert the graph from directed to undirected,
# we can add it to its transpose
adj_sym = adj + adj.T
print(adj_sym)
```

```
[[0 1 1 1 0 0 0 1]
 [1 0 1 1 1 0 0 0]
 [1 1 0 1 0 0 0 0]
 [1 1 1 0 0 1 1 1]
 [0 1 0 0 0 0 0 0]
 [0 0 0 1 0 0 0 0]
 [0 0 0 1 0 0 0 0]
 [1 0 0 1 0 0 0 0]]
```

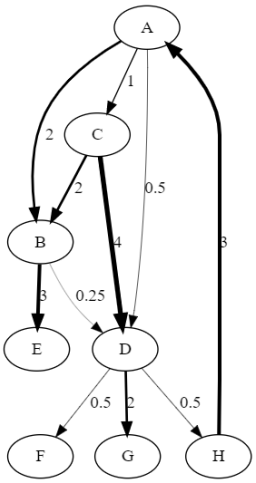
In [5]:
```
# any self transitions?
print(np.diag(adj))
```

```
[0 0 0 0 0 0 0 0]
```

Edge-weighted graphs

If the some of the connections between distibution 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.

For example, the graph below has weighted edges:

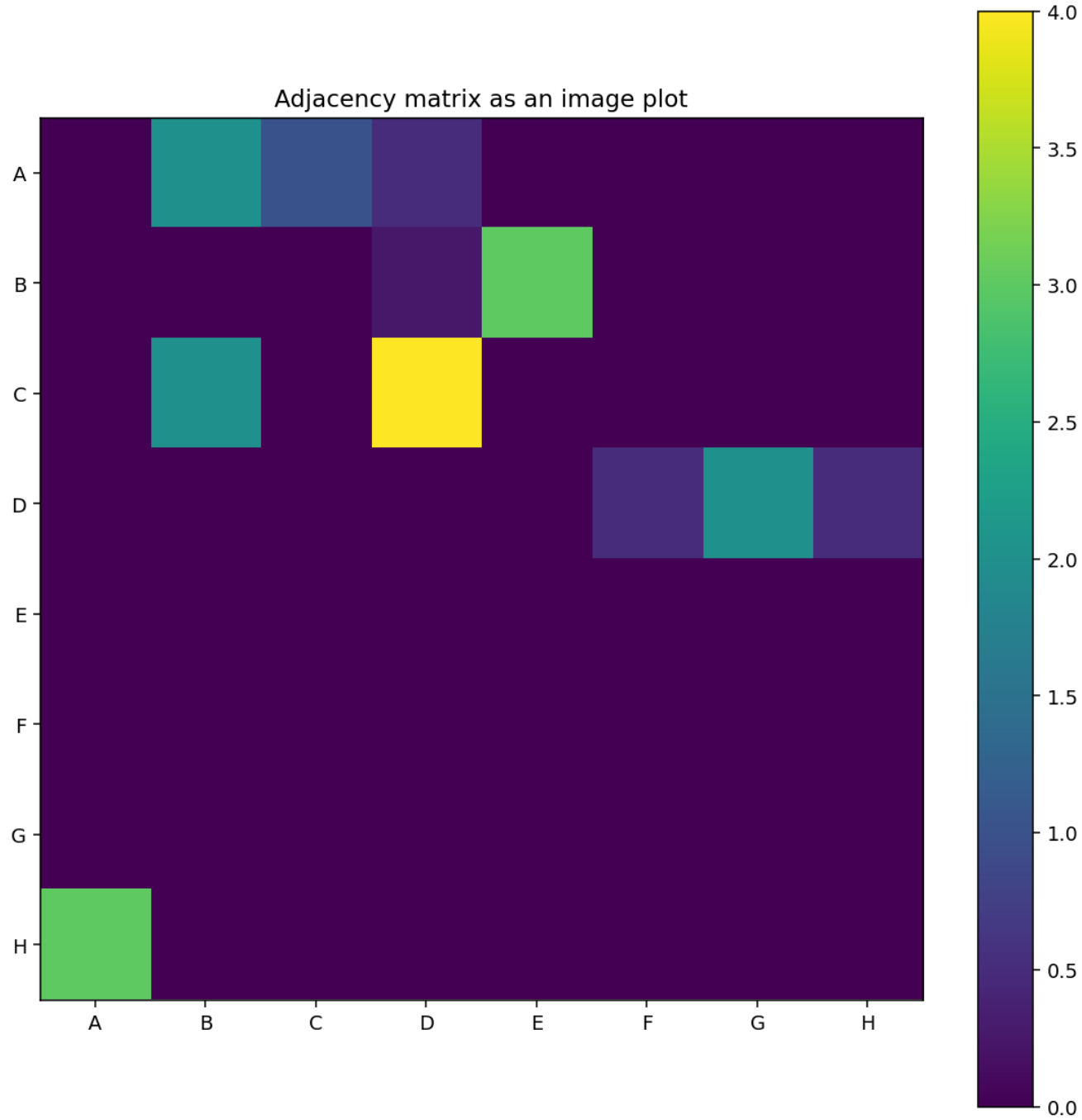


And the adjacency matrix is:

	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

The easiest way to visualise this matrix is by plotting it as an image:

In [6]:
```
A = [[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]]

# plot the adjacency matrix A as an image plot
fig = plt.figure(figsize=(10, 10))
ax = fig.add_subplot(1,1,1)
img = ax.imshow(A)
ax.set_xticks(np.arange(8))
ax.set_xticklabels("ABCDEFGH")
ax.set_yticks(np.arange(8))
ax.set_yticklabels("ABCDEFGH")
fig.colorbar(img)
ax.set_title("Adjacency matrix as an image plot");
```



We can think of graphs as representing flows of "mass" through a network of vertices.

- If the total flow out of a vertex is > 1, i.e. its row sums to > 1, then it is a **source** of mass; for example it is *manufacturing* things.
- If the total flow out of a vertex is < 1, i.e. its row sums to < 1, then it is a **sink**; for example it is *consuming* things.
- If the total flow out of the vertex is 1 exactly, i.e. its row sums to 1 exactly, then it conserves mass; it only ever *re-routes* things.

Can you see which vertices are sources and sinks in the above graph?

If the whole graph consists of vertices whose total outgoing weight is 1.0, and all weights are positive or zero, then the whole graph preserves mass under flow. Nothing is produced or consumed. Every row in the adjacency matrix A sums to 1. This is called a **conserving adjacency matrix**. We can normalise the rows of any positive matrix A (so long as each vertex has at least some flow out of it) to form a conserving adjacency matrix.

In [7]:
```
print(np.sum(A,axis=0)) # along rows
print(np.sum(A,axis=1)) # along collums
```

```
print(np.sum(A))
```

```
[3.   4.   1.   4.75 3.   0.5  2.   0.5 ]
[3.5  3.25 6.   3.   0.   0.   0.   3.  ]
18.75
```

Flow analysis: using matrices to model discrete problems

Previously, we have talked about how matrices perform geometrical transformations on vectors. Matrices are sometimes referred to as *linear maps* because they map from one vector to another, using only linear operations. The adjacency matrix does this too.

What does this mean? What vectors does it operate on? What does the geometrical transformation (linear mapping) represent?

At any point in time, we can write down the proportion of packages at each depot as a vector $\mathbf{x_t} \in \mathbb{R}^V$, where V is the number of vertices in the graph (number of depots), e.g.

$$\mathbf{x_t} = \begin{bmatrix} 0.05 & 0.15 & 0.30 & 0.20 & 0.03 & 0.12 & 0.08 & 0.07 \end{bmatrix}$$

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).

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

The switch of viewpoints from discrete to continuous (and vice versa) is a very powerful and fundamental step in data analysis. It might not seem like it, but the flow of packages can be modelled as some rigid rotation and scaling in a high-dimensional space.

We saw the same kind of application in Topic 1, when we transformed the translation problem from a problem of strings (discrete) to a problem in vector spaces (continuous) and made it solvable.

A great many problems can be represented this way:

- 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)

Simulation of changes in package distribution over time

Suppose we start with an *initial distribution* of packages: a vector $\mathbf{x_{t=0}}$. How many packages will be at each depot tomorrow? This will be given by the (row) vector $\mathbf{x_{t=1}}$, which can be computed as follows:

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

We can simulate the flow *over the whole network* in one go with just one matrix multiplication. This "rotates" the distribution of packages from today to tomorrow. The advantage of vectorised operations is that they can be accelerated using hardware such as a GPU (Graphics Processing Unit).

N.B. Because we are working with row vectors, the adjacency matrix post-multiplies the package-state vector. This is due to the way the adjacency matrix is defined. For other applications we often work with column vectors, in which case the matrix pre-multiplies the vector.

Important questions

There are some harder questions we can ask:

- What about in a week's time? What will $\mathbf{x_{t=7}}$ be?
- What about in *one hour's* time (i.e. a 24th of a day)? What will be $\mathbf{x_{t=1/24}}$ be?
- What about at time infinity $\mathbf{x_{t=\infty}}$? 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 $\mathbf{x_{t=0}}$, can we predict yesterday $\mathbf{x_{t=-1}}$?

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

New matrix operations

- Matrices can be exponentiated: $C = A^n$; this "repeats" the effect of matrix (e.g. $C = AAAA$)
- Matrices can be inverted: $C = A^{-1}$; this undoes the effect of a matrix
- We can find eigenvalues: $A\mathbf{x} = \lambda\mathbf{x}$; this identifies specific vectors $\mathbf{x}$ that are *only* scaled by a factor λ (not rotated) when transformed by matrix A.
- Matrices can be factorised: $A = U\Sigma V^T$; 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)

Since we have already defined matrix multiplication, 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. Matrix exponentiation is the **repeated application** of a matrix, and it can only be defined for square matrices because we'd otherwise change the dimensions after the first step and be unable to reapply the same matrix.

This answers the question, **"What about in a week's time? What will $\mathbf{x_{t=7}}$ be?"** We can simply apply the matrix seven times; raising it to the power of 7.

Matrix powers are very easy to compute for positive, whole powers:

In [8]:
```
# Define a function that raises a matrix A to the power n
def powm(A, n):
    B = np.eye(A.shape[0]) # start with identity
    for i in range(n):
        B = A @ B  # @ performs matrix multiplication
    return B
```

In [9]:
```
# Demonstrate raising a matrix A to powers from 0 to 5
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))
```

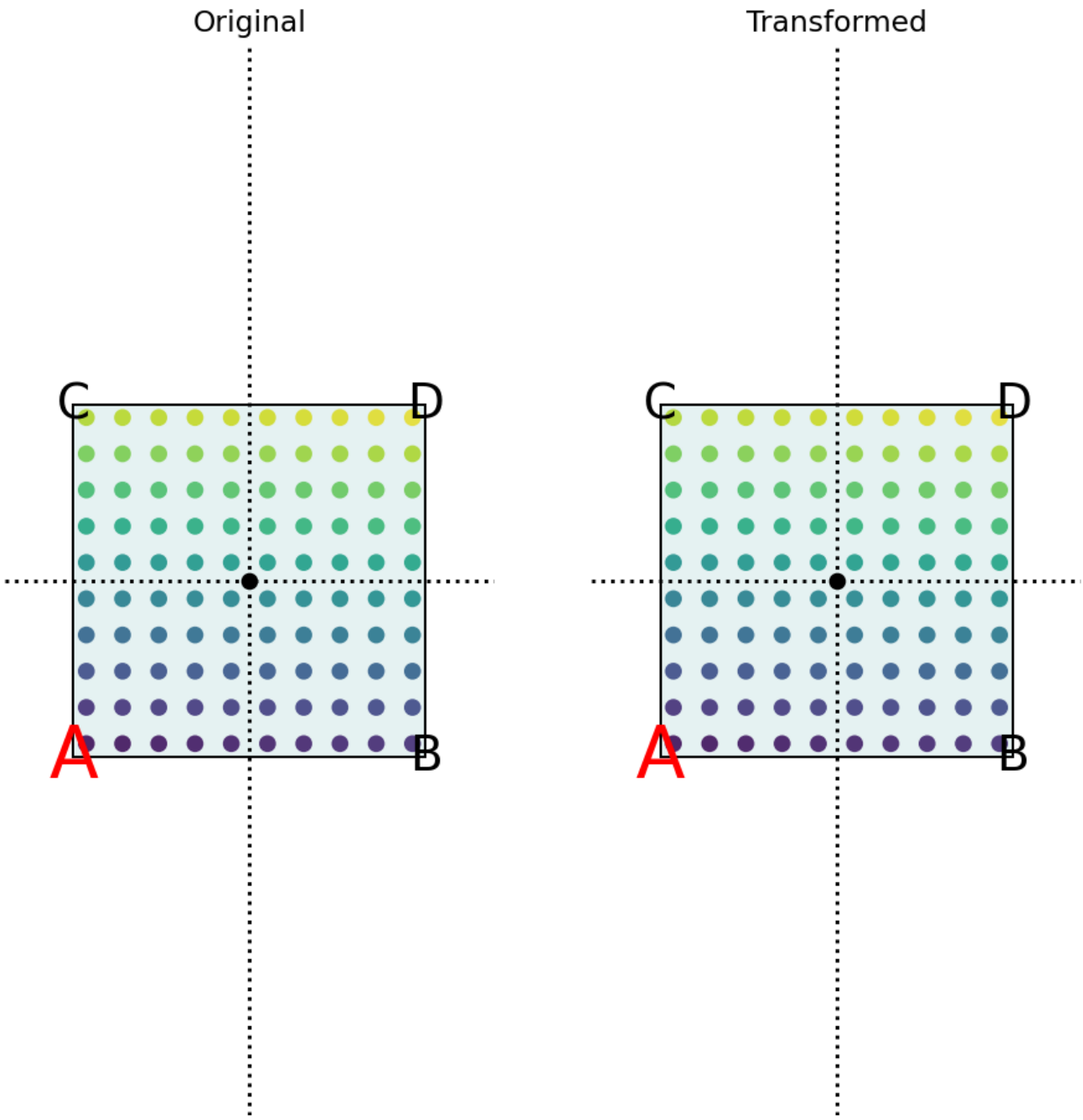
```
A
 [[ 1.5  0. ]
 [-1.1  0.1]]
A^0
 [[1. 0.]
 [0. 1.]]
A^1
 [[ 1.5  0. ]
 [-1.1  0.1]]
A^2
 [[ 2.25  0.  ]
 [-1.76  0.01]]
A^3
 [[ 3.38  0.  ]
 [-2.65  0.  ]]
A^4
 [[ 5.06  0.  ]
 [-3.98  0.  ]]
```

Matrix powers correspond to applying a matrix multiple times, compounding the effect.

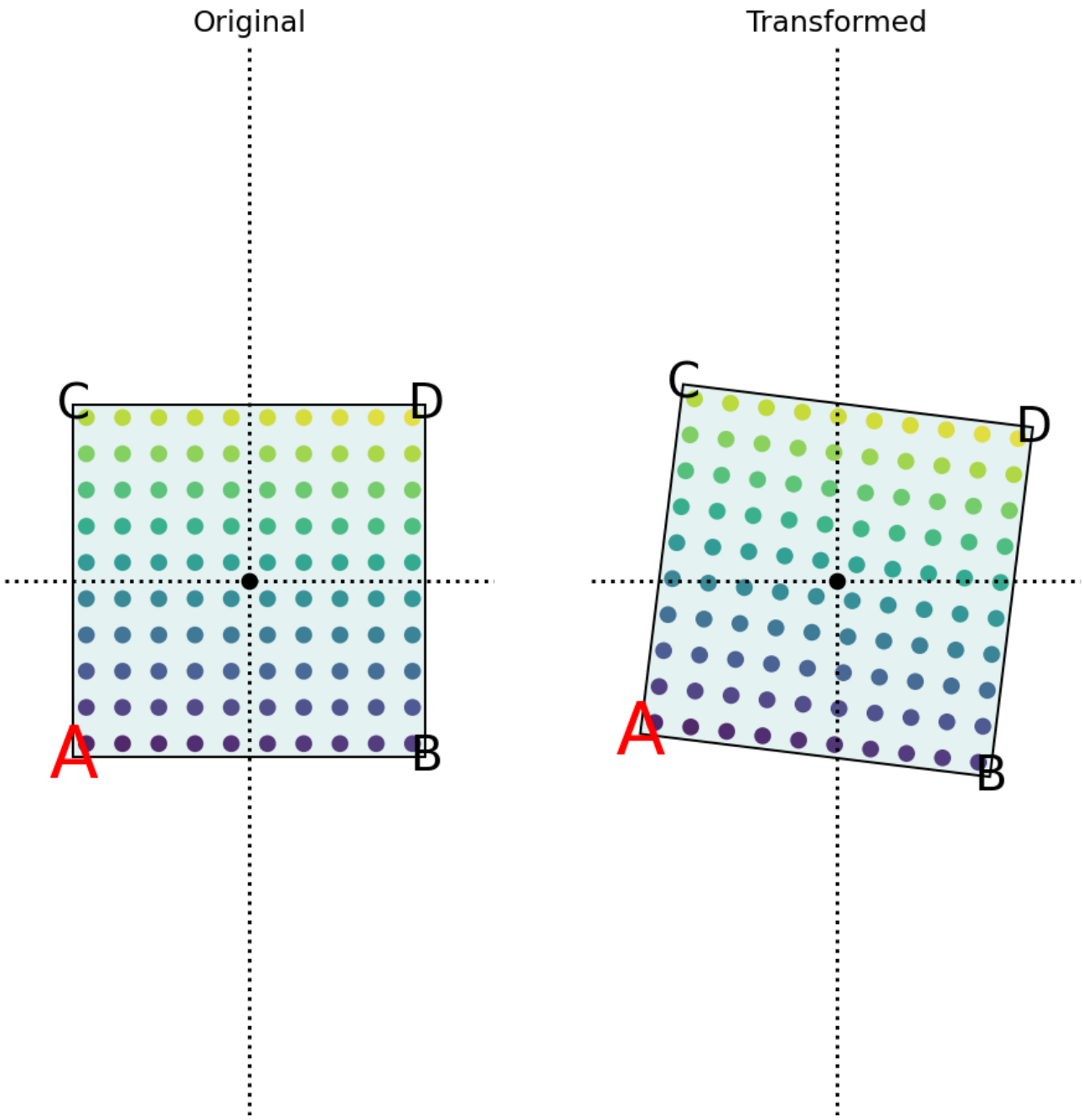
In [10]:
```
# Demonstrate the effect of applying a 7-degree rotation matrix repeatedly,
# i.e. raising it to powers from 0 to 5

rad = np.radians(7) # converts 7 degrees into radians.
c = np.cos(rad)
s = np.sin(rad)
rot = np.array([[c, s], [-s, c]])
print_matrix("A", rot)

# display plots of original and transformed matrix for several different powers
for i in range(7):
    show_matrix_effect(powm(rot, i), 'A^%d' % i)
```

```
A
 [[ 0.99  0.12]
 [-0.12  0.99]]
A^0
 [[1. 0.]
 [0. 1.]]
A^1
 [[ 0.99  0.12]
 [-0.12  0.99]]
A^2
 [[ 0.97  0.24]
 [-0.24  0.97]]
A^3
 [[ 0.93  0.36]
 [-0.36  0.93]]
A^4
 [[ 0.88  0.47]
 [-0.47  0.88]]
A^5
 [[ 0.82  0.57]
 [-0.57  0.82]]
A^6
 [[ 0.74  0.67]
 [-0.67  0.74]]
```

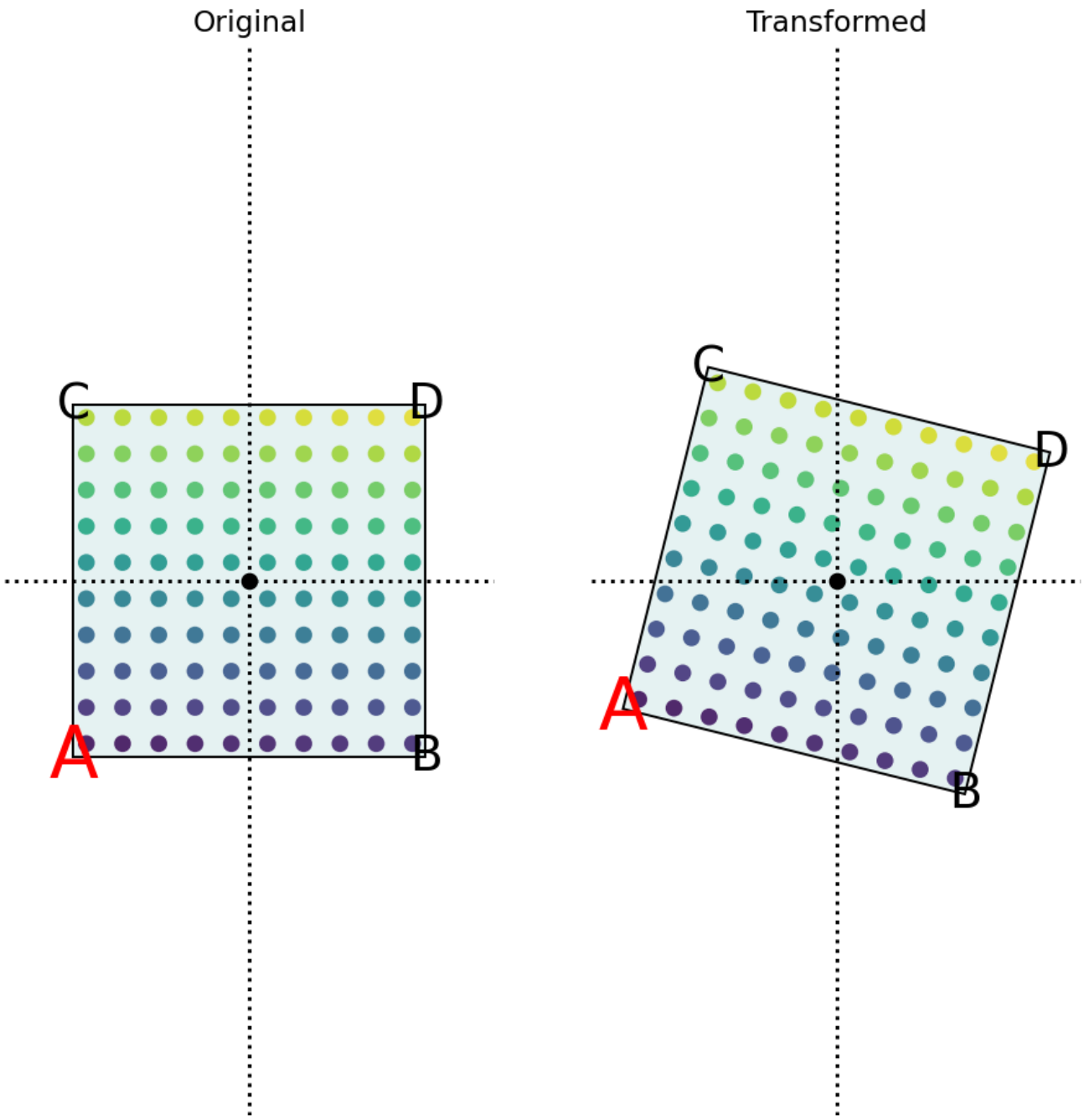
A^0



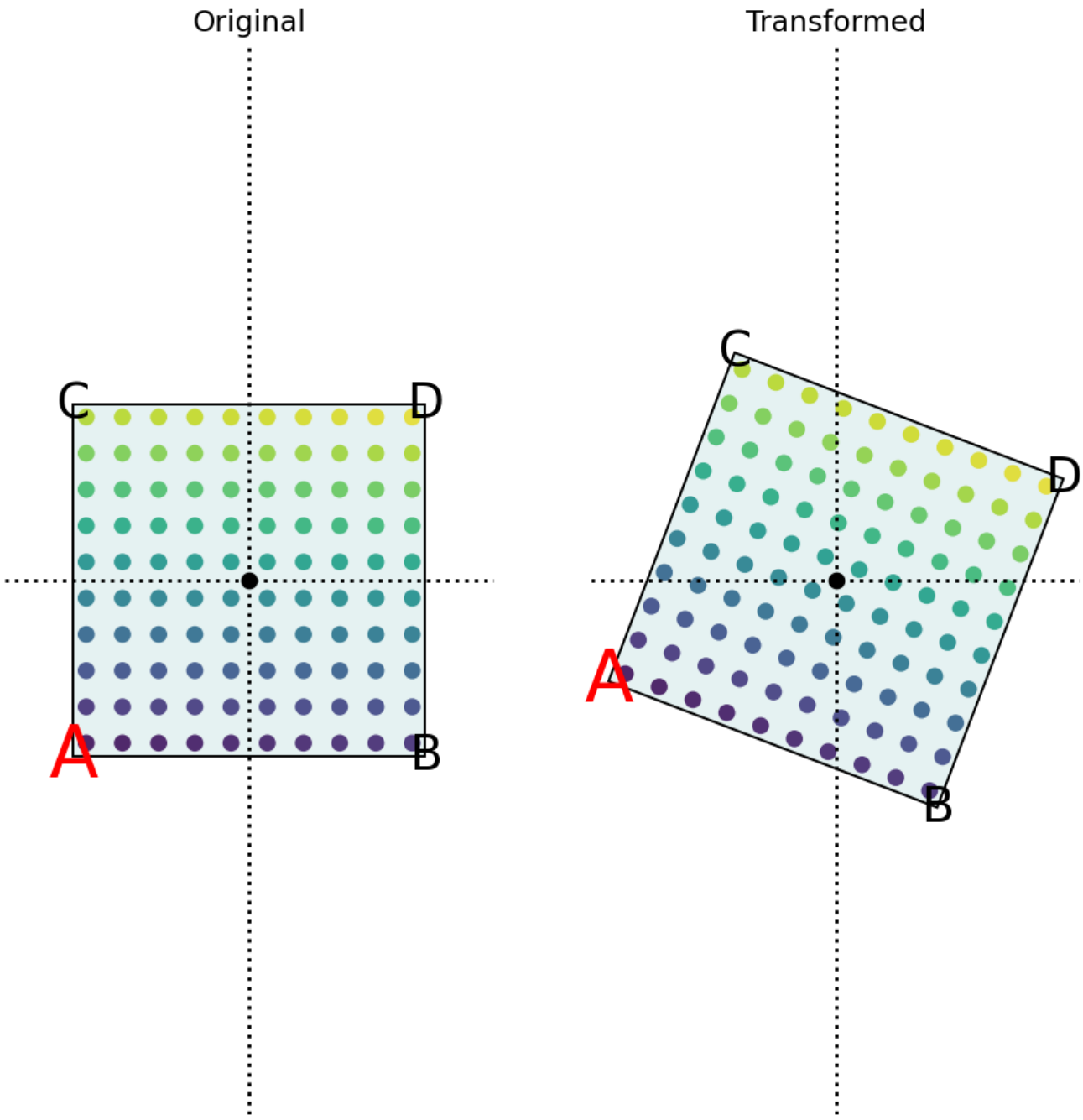
A^1



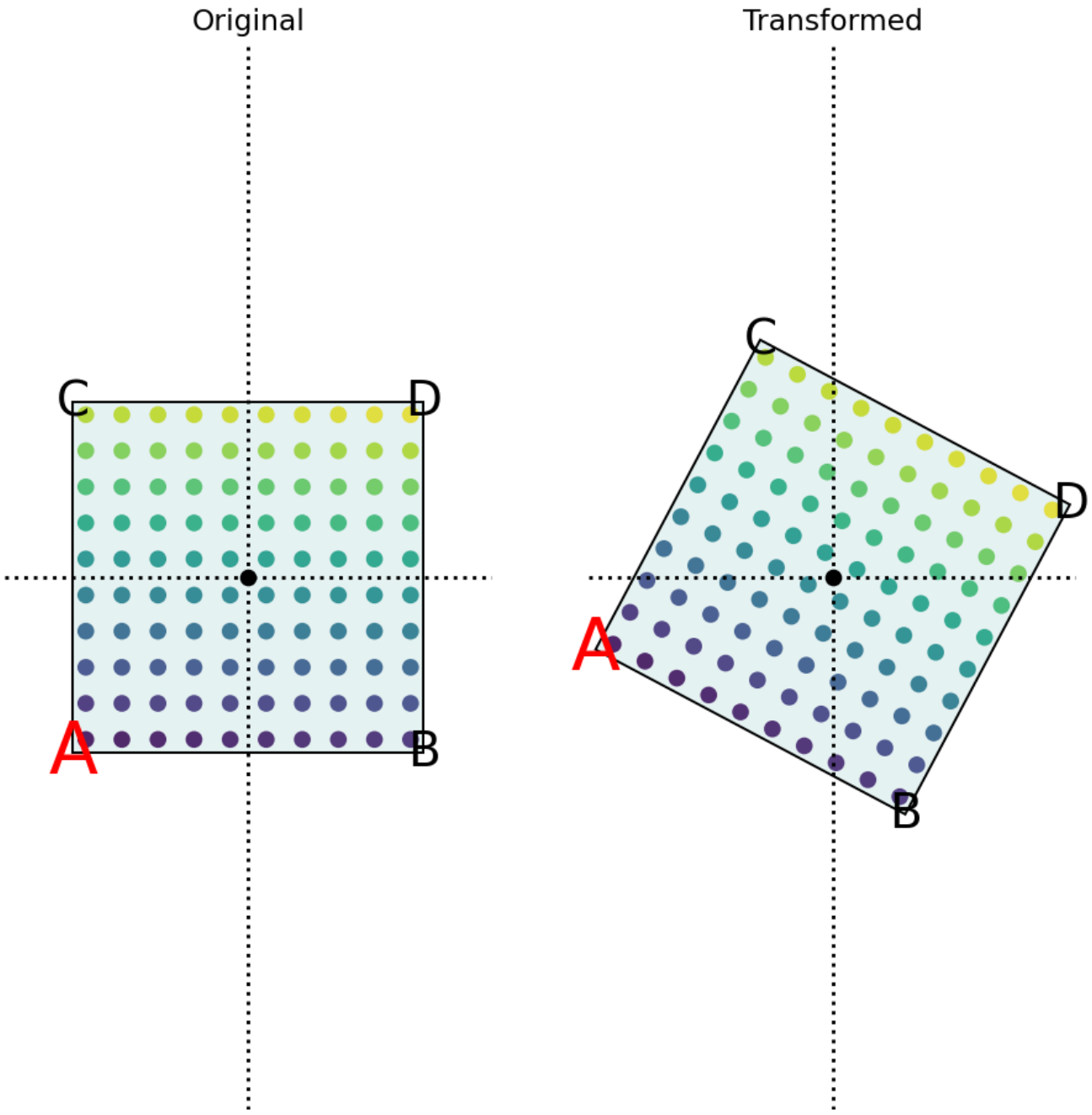
A^2



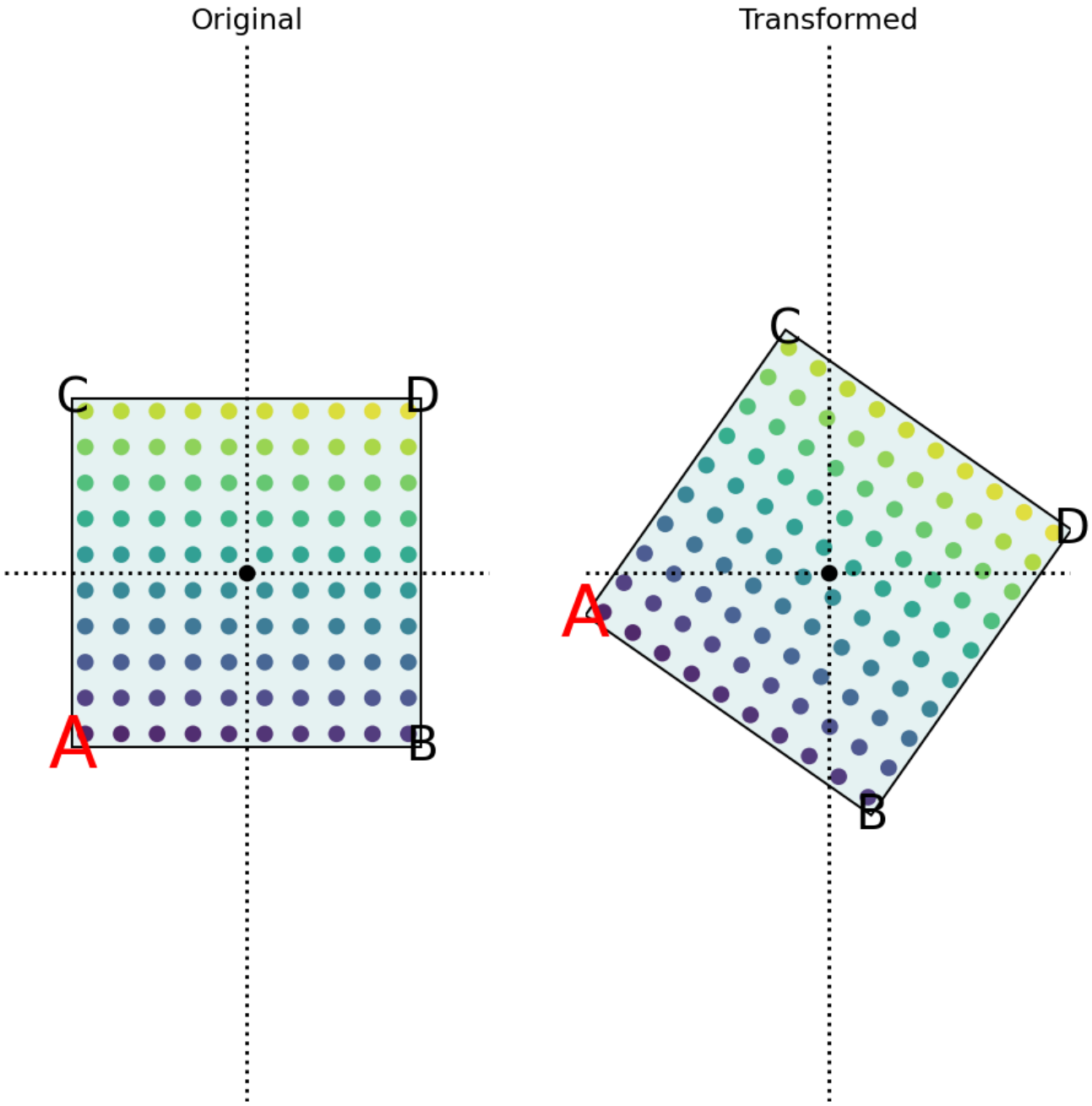
A^3

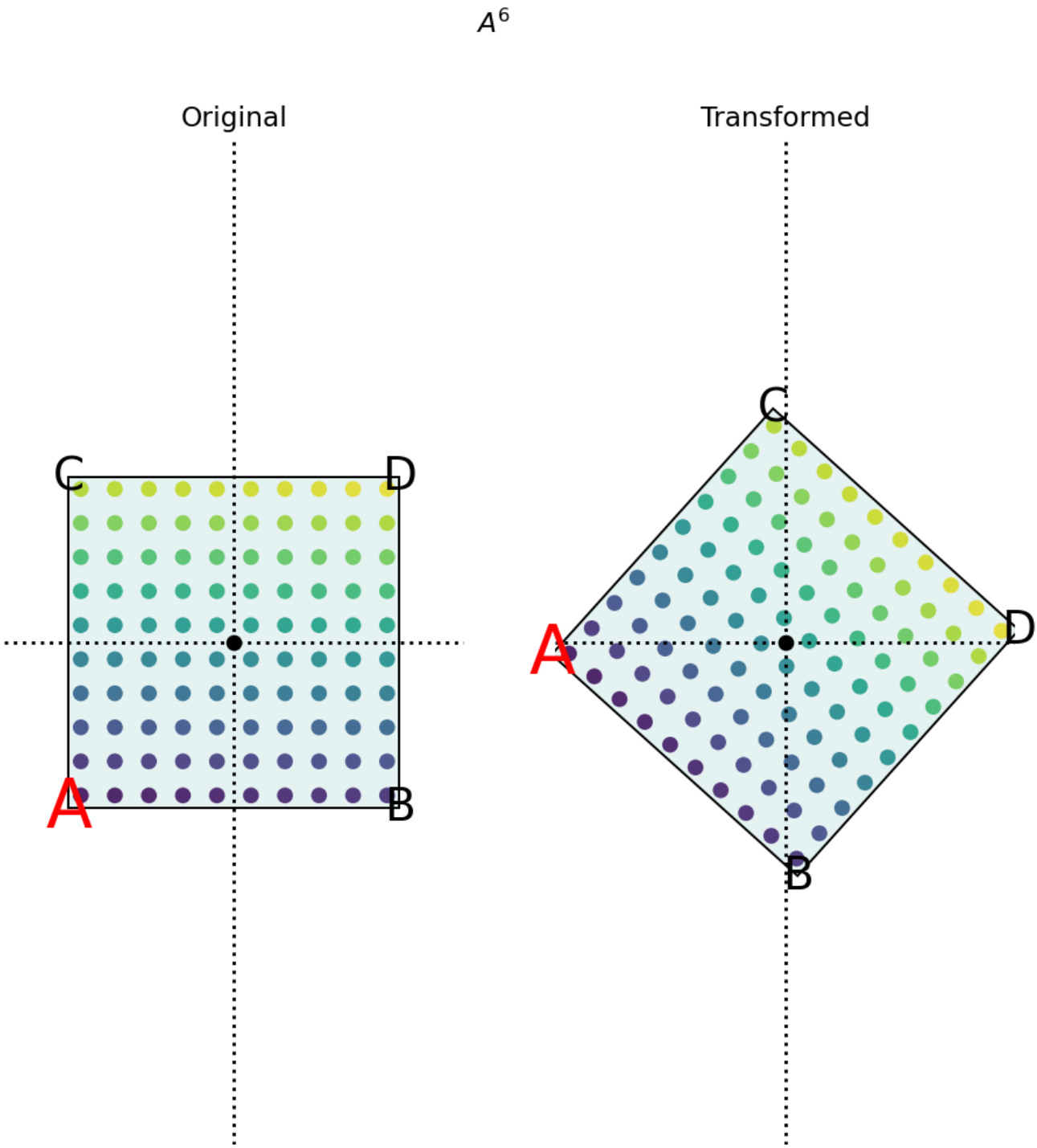


A^4



A^5





Stable point: all roads lead to Rome

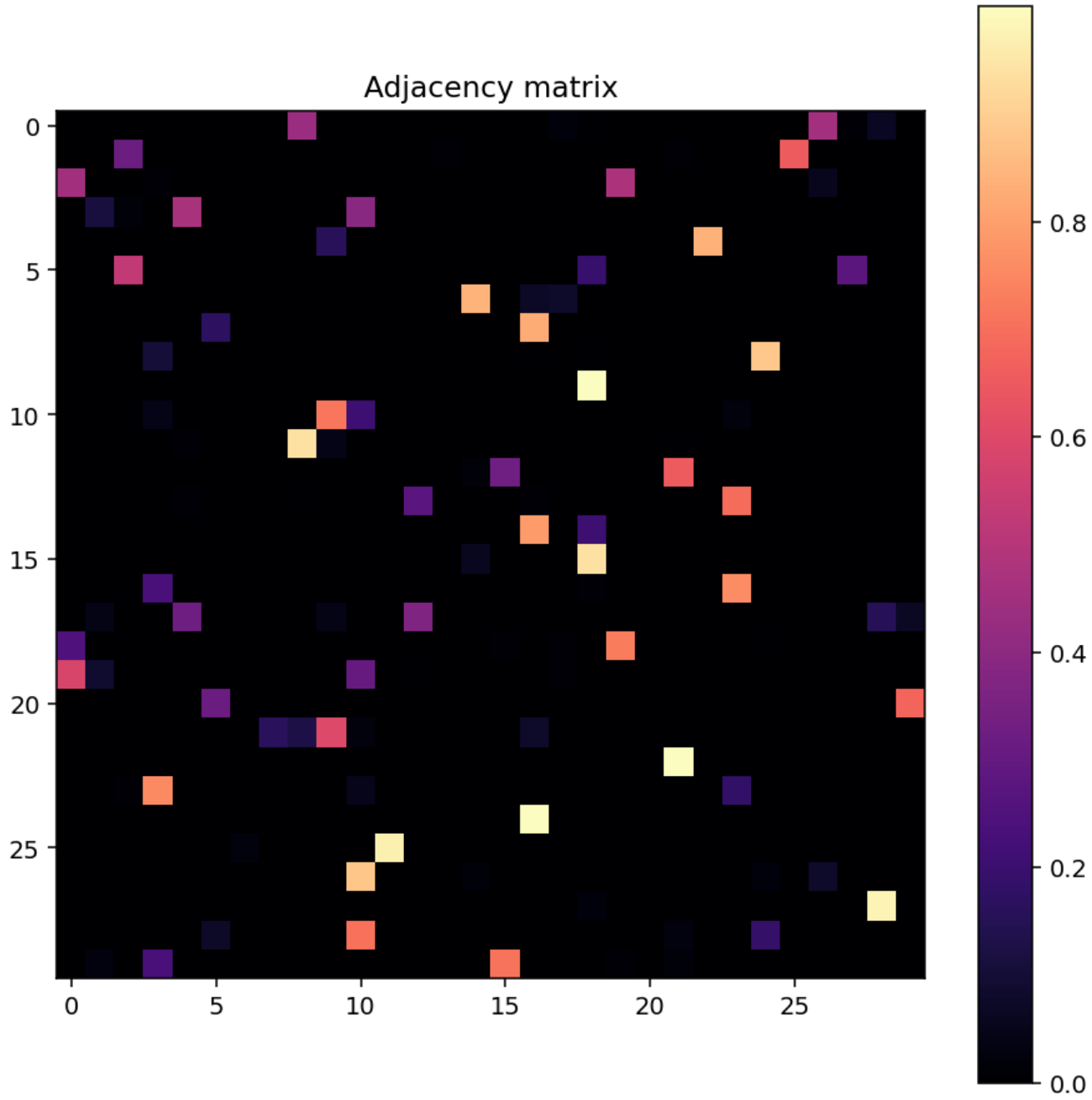
Here is a simple experiment. Imagine we have a conserving adjacency matrix modelling the flow of packages around a network of depots. If we start with different package distributions and then let the "natural" flow begin, what will happen?

In [11]:
```
n_nodes = 30

# generate a random transition matrix
adjacency = np.random.uniform(0, 1, (n_nodes, n_nodes))**50

# normalise rows to make a conserving adjacency matrix !
adjacency = (adjacency.T / np.sum(adjacency, axis=1)).T

# show the adjacency matrix
fig = plt.figure()
```

```
ax = fig.add_subplot(1,1,1)
img = ax.imshow(adjacency, cmap='magma')
ax.set_title("Adjacency matrix")
fig.colorbar(img)
```

Out[11]:

```
<matplotlib.colorbar.Colorbar at 0x1570c5c90>
```

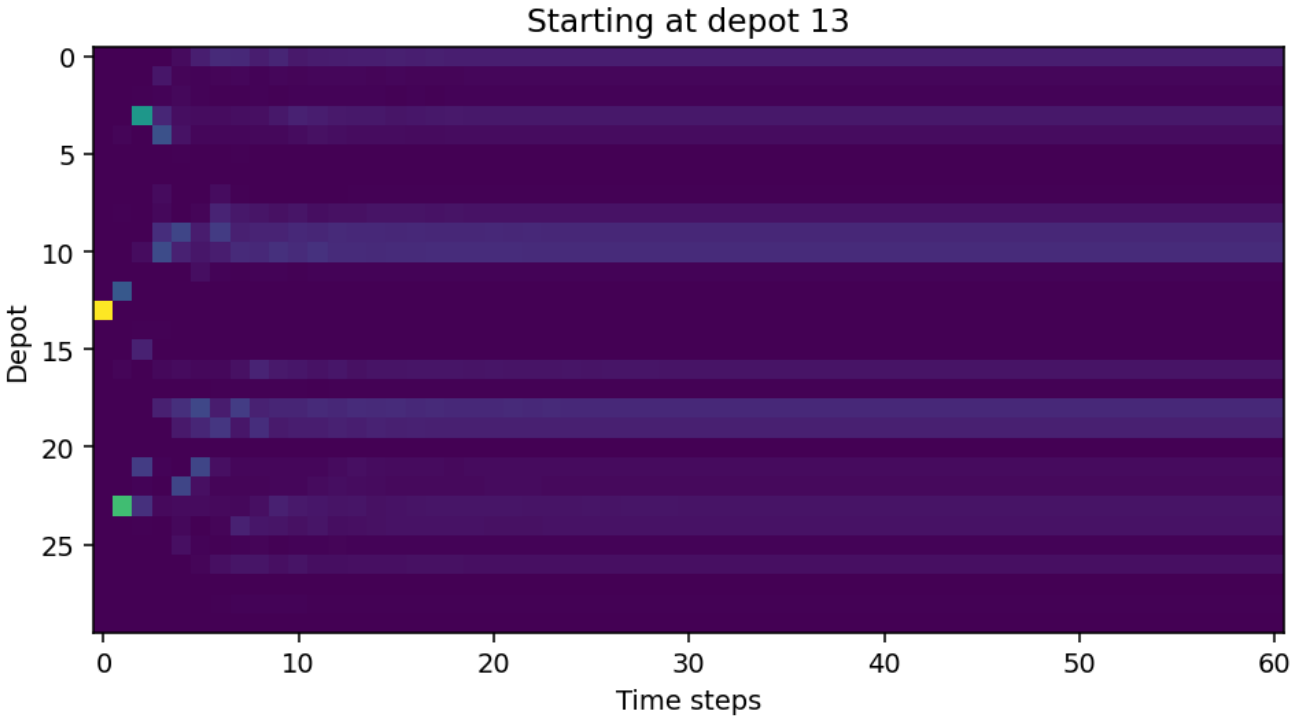


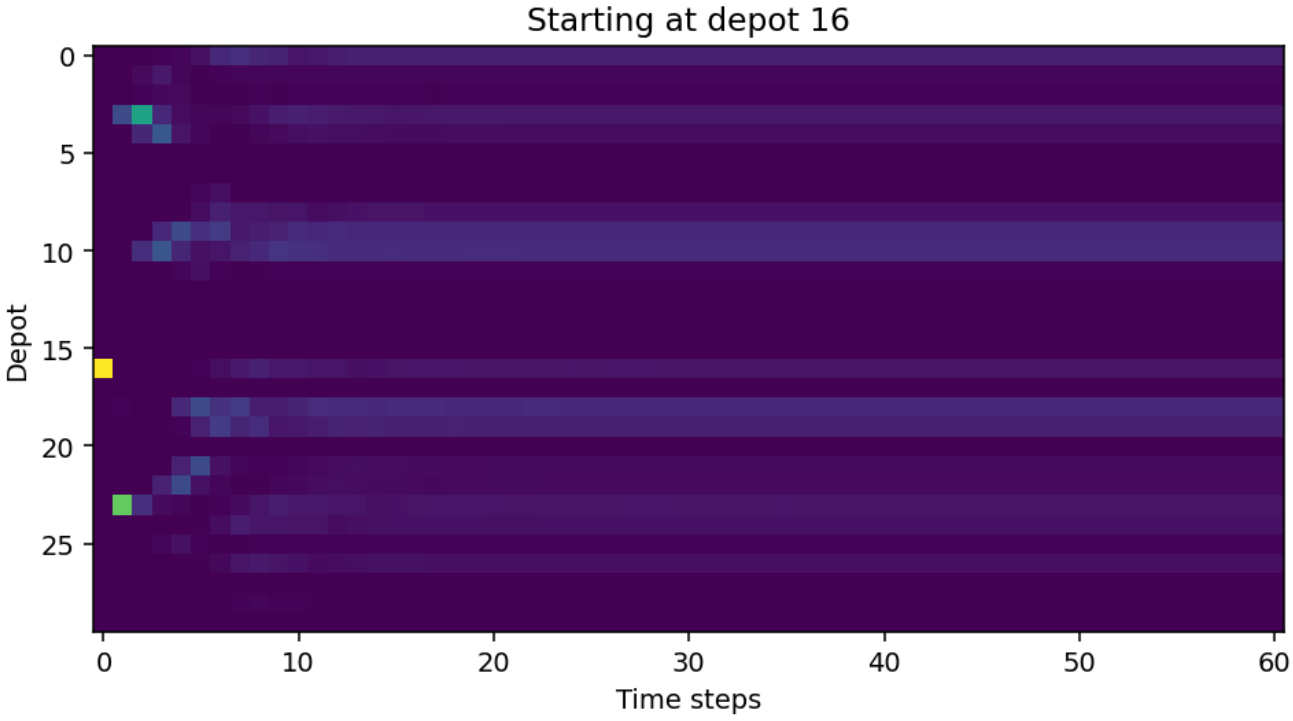
we do some experiments where we try starting the system in different states and see how it evolves

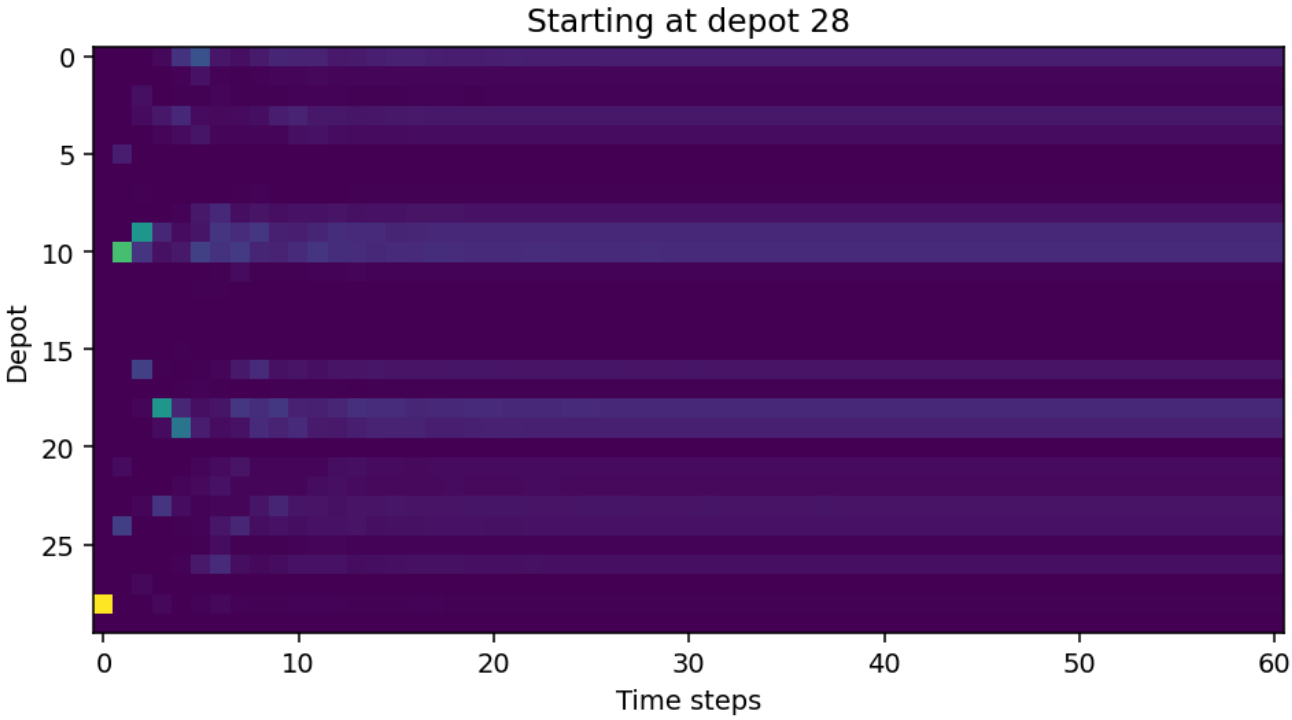
In [12]:

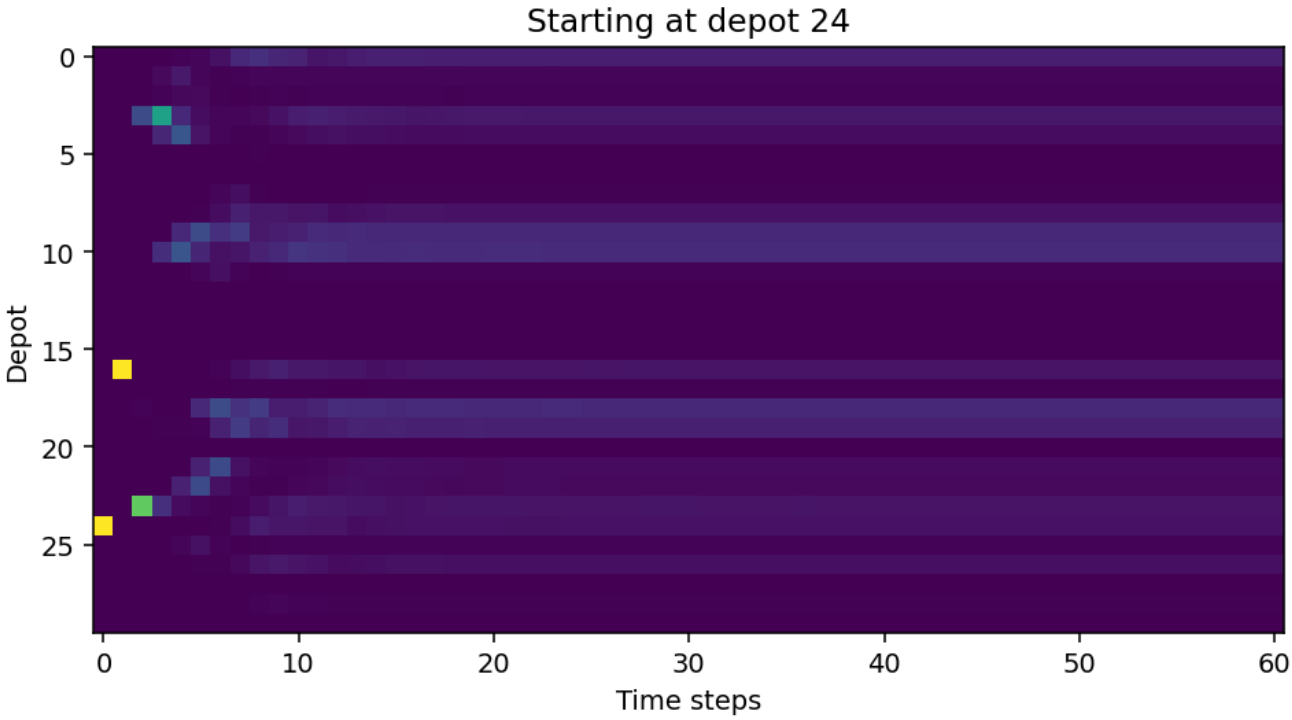
```
# now we do some experiments where we try starting the system in different states
# and see how it evolves (we can imagine a system of tanks, where we put dye in
# one tank to begin with, and see where it ends up after we turn the pump on)
timesteps = 60

# run the experiment a few times with different starting states
for j in range(5):
    test_vector = np.zeros((1, n_nodes))  # initialise a row vector

    # choose a random node to start all the packages in
    node = np.random.randint(0, n_nodes)
    test_vector[0][node] = 1

    fig = plt.figure()
    ax = fig.add_subplot(1, 1, 1)
```

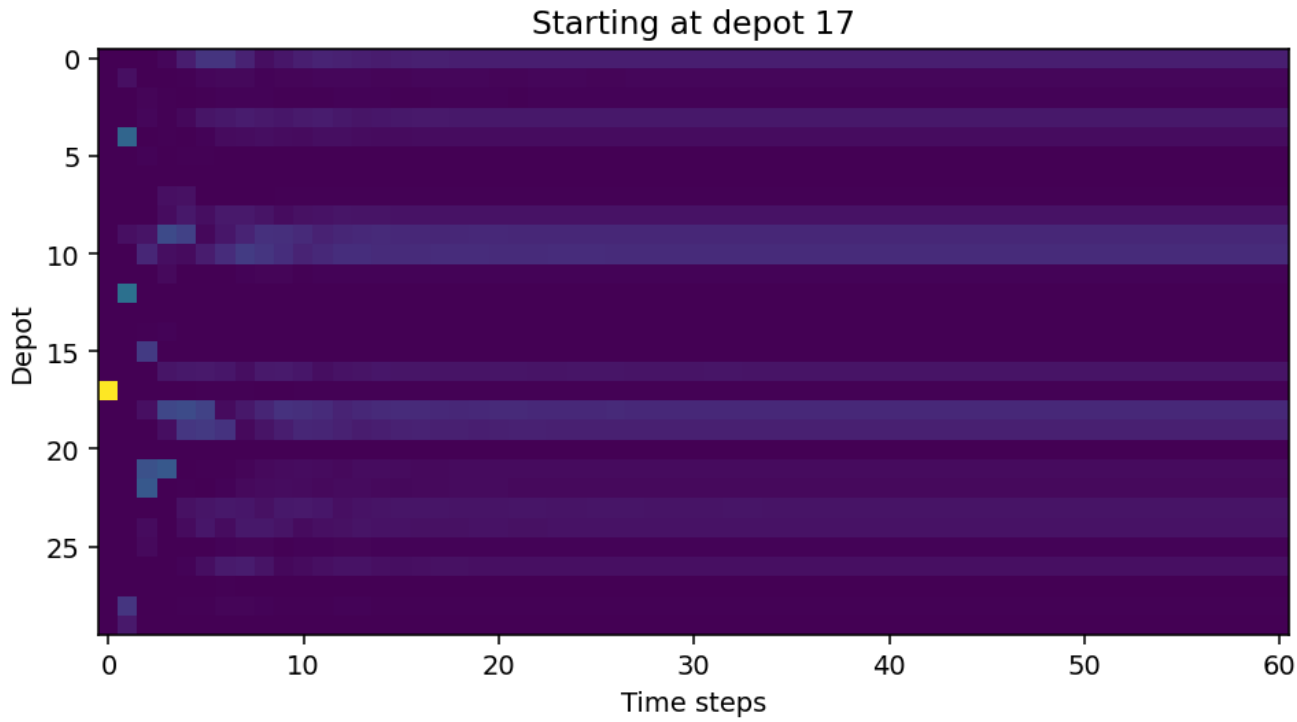
```
    # initialise a list to store vectors at every time point
    vectors = []
    vectors.append(test_vector.T) # store as column vector

    # let the system evolve over a number of timesteps
    for i in range(timesteps):
        test_vector = test_vector @ adjacency
        vectors.append(test_vector.T)

    # plot the state vector as a function of time
    all_vectors = np.hstack(vectors)
    ax.imshow(all_vectors)
    ax.set_xlabel("Time steps")
    ax.set_ylabel("Depot")
    ax.set_title("Starting at depot %d" % node)
```






Starting at depot 28
Depot
Time steps
0
5
10
15
20
25
0
10
20
30
40
50
60


Starting at depot 24
Depot
Time steps
0
5
10
15
20
25
0
10
20
30
40
50
60



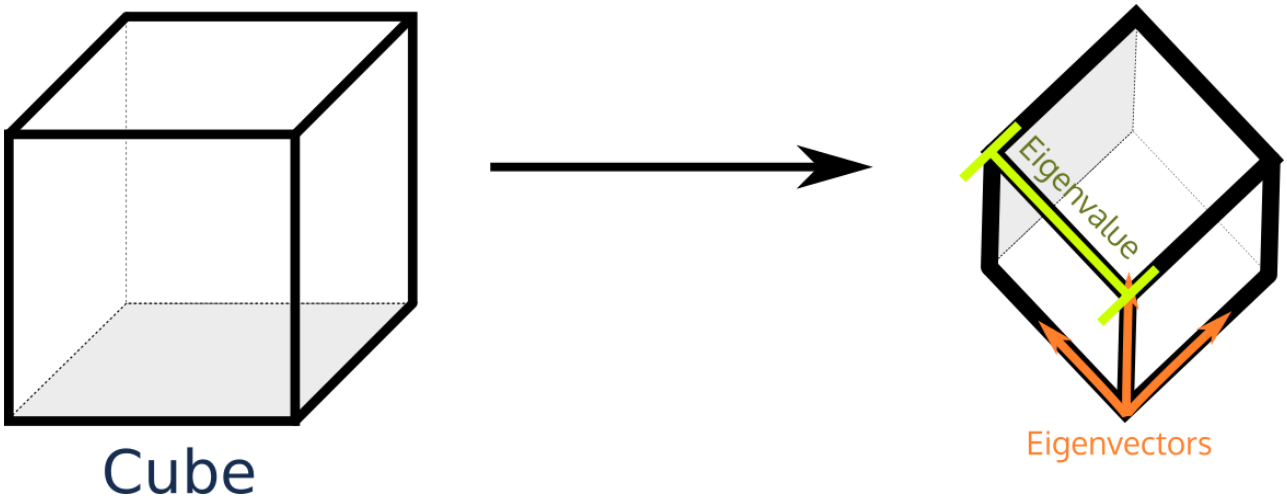
Surprised...? No matter what the initial package distribution is, after several days the distribution will settle down to a **steady state** (or stable point). This vector is one of the *eigenvectors* of the adjacency matrix. Applying the matrix again does not change the distrbution of the packages, i.e. $\mathbf{x}_{t=60} = \mathbf{x}_{t=59}A$

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.

Special vectors: 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, as they have some stability. The prefix **eigen** just means **characteristic** (from the German for "own"). The scaling factors that the matrix applies to its eigenvectors are called **eigenvalues**.

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 edges of this parallelepiped have unit length, then after the transformation their lengths will be equal to the eigenvalues. The youtube video series on linear algebra by 3blue1brown illustrates this particularly well.



When we are dealing with n-dimensional matrices, we have to imagine an n-dimensional parallelotope being transformed in the same way.

How to find the leading eigenvector: the power iteration method

Our example, "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. However, there is one important caveat ...

What happens if we apply a square matrix A to a vector $\mathbf{x}$ of any length, then take the resulting vector and apply A again, and so on?

Let's work with column vectors now, so that A pre-multiplies the vector. If we compute

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

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

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

The vector will be forced back to unit norm at each step, using the L_∞ norm. This process is called **power iteration**.

N.B. You can use any norm for normalisation. The infinity norm has been used traditionally, probably for reasons of efficiency.

Let's demonstrate power iteration with a random 3×3 matrix A:

In [13]:
```
def power_iterate(A, x, n):
    for i in range(n):
        x = A @ x # matrix multiplication
        x = x / np.linalg.norm(x, np.inf) # normalize x using the L-infinity norm during powe

    return x / np.linalg.norm(x, 2) # finally, divide by the L2 norm so x can be compared wit
```

In [14]:
```
# create a random 3 x 3 matrix
A = np.random.normal(0, 1, (3, 3))
# make it symmetric to ensure all eigenvalues are real (not complex)
A = A + A.T
print_matrix("A", A)
```

```
A
 [[-1.31 -0.12  1.17]
 [-0.12 -2.24 -0.2 ]
 [ 1.17 -0.2   0.13]]
```

In [15]:
```
# create a random 3-element column vector
random_vec = np.random.normal(0, 1, (3, 1))
print_matrix("\\bf{x_0}", random_vec)
```

```
\bf{x_0}
 [[0.36]
 [0.19]
 [0.67]]
```

In [16]:
```
# do power iteration on this vector for enough iterations to ensure it converges to the leadi
xn = power_iterate(A, random_vec, n=500)
print_matrix("\\bf{x_{n}}", xn)
```

```
\bf{x_{n}}
 [[0.07]
 [1.  ]
 [0.05]]
```

Regardless of which vector $\mathbf{x_0}$ 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). We know this **must** be true, because 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 $\mathbf{x}$ as follows:

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

where λ is the eigenvalue. We can calculate λ simply by dividing the vector $A\mathbf{x}$ element-wise by the vector $\mathbf{x}$:

In [17]:
```
ratio = (A @ xn) / xn # elementwise
print_matrix("\\frac{A\\bf x}{\\bf x}", ratio)
eigenvalue = ratio[0][0] # all elements of ratio will be the same
print_matrix("\lambda", eigenvalue)
```

```
\frac{A\bf x}{\bf x}
 [[-2.26]
 [-2.26]
 [-2.26]]
```

$\lambda = -2.256027636110371$

In [18]:
```
# sanity check
print((A @ xn))
print(eigenvalue * xn)
```

```
[[-0.14963627]
 [-2.2479452 ]
 [-0.11837252]]
[[-0.14963627]
 [-2.2479452 ]
 [-0.11837252]]
```

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`:

In [20]:
```
# 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=
[[-1.3132238  -0.12433635  1.1693901 ]
 [-0.12433635 -2.23704613 -0.20329184]
 [ 1.1693901  -0.20329184  0.12633055]]

Eigenvalues:
 [ 0.7983868  -1.96629854 -2.25602764]

Unit Eigenvectors (columns):
 [[ 0.48649904 -0.87115979  0.06632732]
 [-0.0782059   0.03218997  0.9964174 ]
 [ 0.87017385  0.48994329  0.05246945]]
```

The leading eigenvector and eigenvalue should match the results we obtained by power iteration.

Notice that our 3×3 matrix has 3 eigenvalues and 3 eigenvectors. In general, for an $n \times n$ matrix, `np.linalg.eig` will yield n eigenvalues and n eigenvectors. The eigenvectors are **orthogonal**, i.e. the dot product of any pair of eigenvectors is zero.

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

Formal definition of eigenvectors and eigenvalues

Now that we have explored the concepts of eigenvectors and eigenvalues through practical examples, we can move on to a formal definition:

Consider a vector function $f(\mathbf{x})$. There may exist vectors such that $f(\mathbf{x}) = \lambda\mathbf{x}$. The function maps these vectors to scaled versions of themselves. No rotation or skewing is applied, just pure scaling.

Any square matrix A represents a function $f(\mathbf{x})$ and may have vectors like this, such that

$$A\mathbf{x}_i = \lambda_i\mathbf{x}_i$$

Each vector $\mathbf{x}_i$ satisfying this equation is known as an **eigenvector** and each corresponding factor λ_i is known as an **eigenvalue**.

For any matrix, the **eigenvalues** are uniquely determined, but the eigenvectors are not. There may be many eigenvectors corresponding to any given eigenvalue (for example, pairs of vectors that point in opposite directions).

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

The eigendecomposition: revisiting the package distribution problem

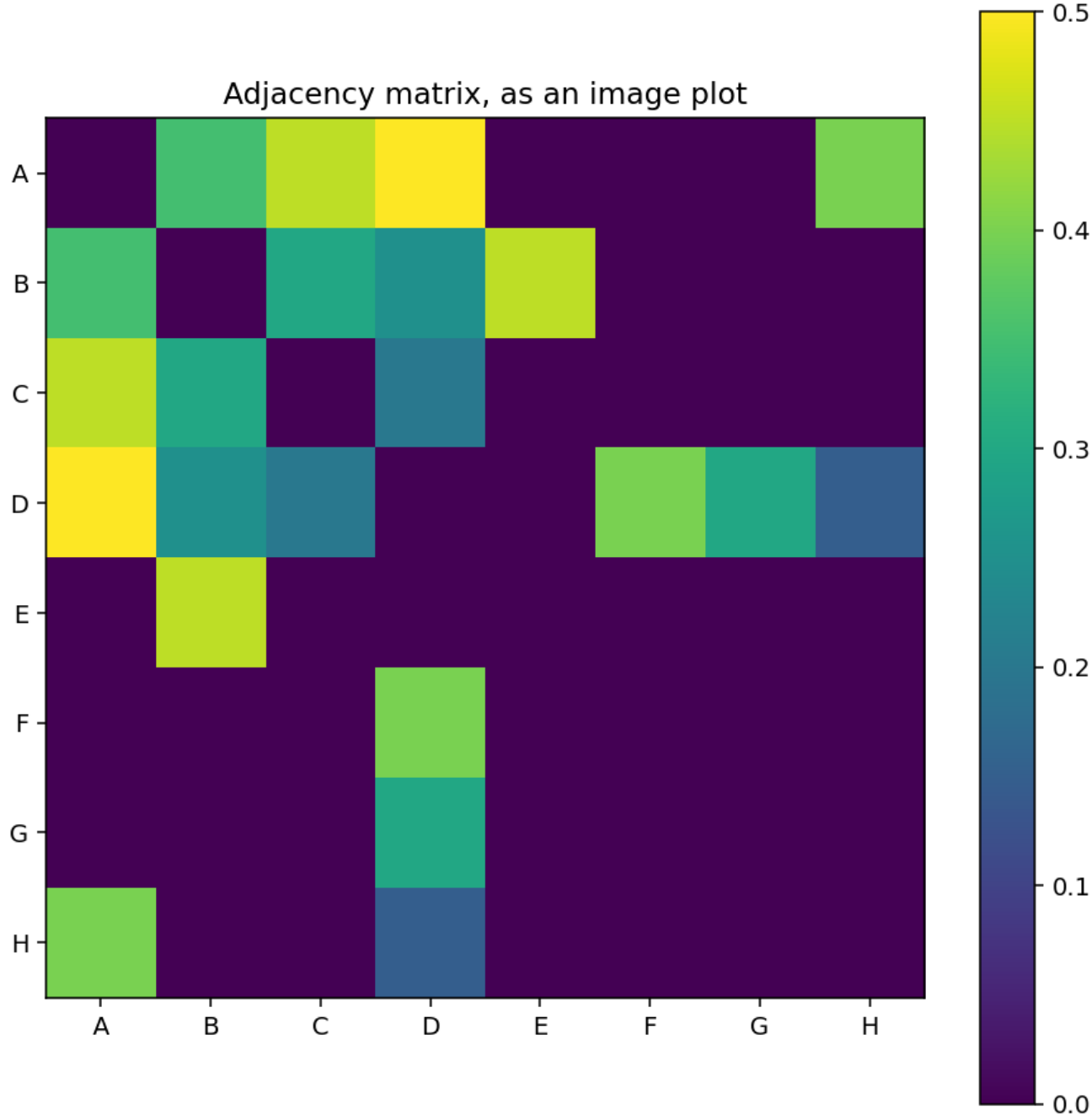
Let's revisit our package distribution example and compute all the eigenvectors and eigenvalues of an adjacency matrix.

We will work with an *undirected* (symmetric) graph to ensure that the eigenvalues are all real (an adjacency matrix whose eigenvalues are all complex would have no steady state, but instead would converge to an oscillatory state).

N.B. Making the adjacency matrix symmetric also means we don't have to worry about transposing it before using `np.linalg.eig` (the adjacency matrix is defined such that it operates on row vectors, but `np.linalg.eig` returns column eigenvectors).

In [21]:
```
# set up an adjacency matrix A to represent package flow
A = np.array([[0.0, 0.7, 0.9, 1.0, 0.0, 0.0, 0.0, 0.0],
              [0.0, 0.0, 0.0, 0.5, 0.9, 0.0, 0.0, 0.0],
              [0.0, 0.6, 0.0, 0.4 , 0.0, 0.0, 0.0, 0.0],
              [0.0, 0.0, 0.0, 0.0 , 0.0, 0.8, 0.6, 0.3],
              [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],
              [0.8, 0.0, 0.0, 0.0 , 0.0, 0.0, 0.0, 0.0]])

# make A symmetric (undirected)
A = (A + A.T) * 0.5

# plot adjacency matrix as image
```

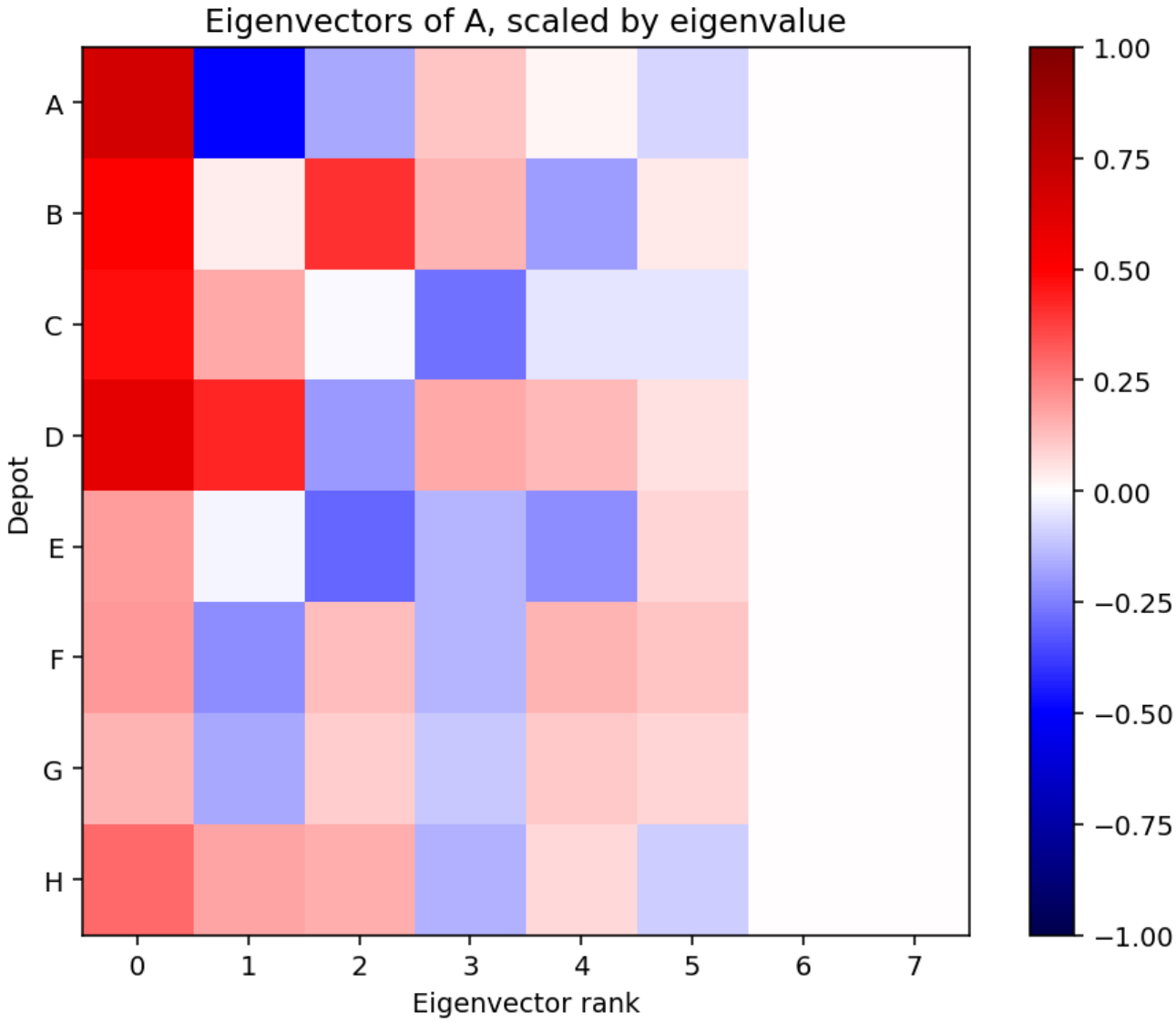
```
fig = plt.figure()
ax = fig.add_subplot(1,1,1)
img = ax.imshow(A)
ax.set_xticks(np.arange(8))
ax.set_xticklabels("ABCDEFGH")
ax.set_yticks(np.arange(8))
ax.set_yticklabels("ABCDEFGH")
fig.colorbar(img)
ax.set_title("Adjacency matrix, as an image plot");
```



In [22]:

```
# 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]
/var/folders/98/0yyrgrcx195f54qm57r6ygbr0000gn/T/ipykernel_65336/3274880211.py:10: UserWarnin
g: FixedFormatter should only be used together with FixedLocator
  ax.set_yticklabels("ABCDEFGH")
```



The leading eigenvector tells us what the steady state of the package distribution will be if we keep the system running for long enough. For this distribution network, most of the packages will pile up at A, B, C, D and H.

From the eigendecomposition we can get a feel for which eigenvectors are large and which are small. In the case of package distribution, small eigenvalues correspond to route patterns that are rarely used, and large eigenvalues to the dominant path of packages moving through the system.

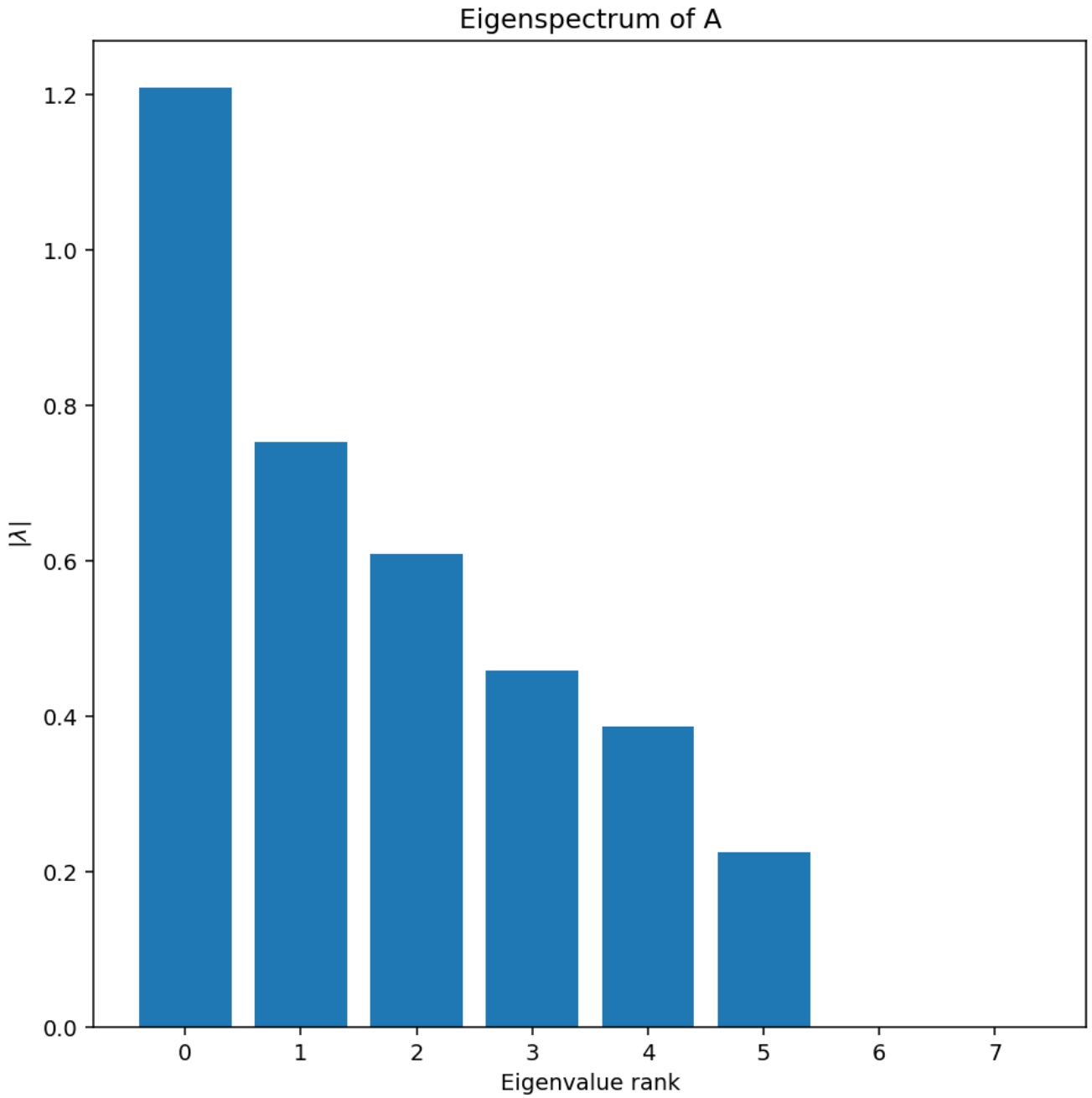
The eigenspectrum

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". As we shall see later, this can be useful in finding "simplified" versions of linear transforms.

In [23]:
```
# sort eigenvalues by absolute value
print_matrix("|\lambda_i|", np.abs(evals[np.argsort(-np.abs(evals))]))

# plot the eigenspectrum (magnitudes of eigenvalues in decreaing order)
fig = plt.figure()
ax = fig.add_subplot(1,1,1)
ax.bar(np.arange(len(evals)), np.abs(evals[np.argsort(-np.abs(evals))]))
```

```
ax.set_title("Eigenspectrum of A")
ax.set_xlabel("Eigenvalue rank")
ax.set_ylabel("$|\lambda|$");
```

```
|\lambda_i|
 [[1.21 0.75 0.61 0.46 0.39 0.22 0.   0.  ]]
```



Warning: Numerical instability of eigendecomposition algorithms

A word of warning: `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. `np.linalg.eig` is often sufficient for most purposes, but be careful how you use it.

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`.

Here we will just demonstarte that you need to be careful.

In [24]:
```
# compute eigendecomposition of A using np.linalg.eig
evals, evecs = np.linalg.eig(A)

# Get the indices of the eigenvalues in order of largest to smallest (absolute value)
```

```
order = np.argsort(-np.abs(evals))

# Reorder eigenvalues and eigenvectors from largest to smallest eigenvalue (absolute value)
evals = evals[order]
evecs = evecs[:, order]

# Display eigenvectors x_i (columns) in order from largest to smallest eigenvalue (absolute)
print("Eigenvectors computed with np.linalg.eig:")
print_matrix("\\bf{x_i}", evecs)

print()

# compute eigendecomposition of A using np.linalg.eigh (we can use this because A is real and
evalsh, evecsh = np.linalg.eigh(A)

# Get the indices of the eigenvalues in order of largest to smallest (absolute value)
order = np.argsort(-np.abs(evalsh))

# Reorder eigenvalues and eigenvectors from largest to smallest eigenvalue (absolute value)
evalsh = evalsh[order]
evecsh = evecsh[:, order]

# Display eigenvectors x_i (columns) in order from largest to smallest eigenvalue (absolute)
print("Eigenvectors computed with np.linalg.eigh:")
print_matrix("\\bf{x_i}", evecsh)
```

```
Eigenvectors computed with np.linalg.eig:
\bf{x_i}
 [[ 0.55  0.66  0.28 -0.24  0.06 -0.36  0.   -0.  ]
 [ 0.42 -0.04 -0.66 -0.32 -0.5   0.18  0.   -0.  ]
 [ 0.39 -0.23  0.02  0.6  -0.14 -0.23 -0.15 -0.6 ]
 [ 0.49 -0.56  0.32 -0.36  0.36  0.28 -0.   -0.  ]
 [ 0.16  0.03  0.49  0.31 -0.58  0.36  0.1   0.4 ]
 [ 0.16  0.3  -0.21  0.31  0.37  0.49  0.59 -0.05]
 [ 0.12  0.23 -0.16  0.23  0.28  0.37 -0.77  0.12]
 [ 0.24 -0.24 -0.26  0.33  0.2  -0.45  0.17  0.68]]

Eigenvectors computed with np.linalg.eigh:
\bf{x_i}
 [[ 0.55  0.66  0.28  0.24  0.06  0.36  0.   -0.  ]
 [ 0.42 -0.04 -0.66  0.32 -0.5  -0.18  0.   -0.  ]
 [ 0.39 -0.23  0.02 -0.6  -0.14  0.23  0.61  0.01]
 [ 0.49 -0.56  0.32  0.36  0.36 -0.28  0.    0.  ]
 [ 0.16  0.03  0.49 -0.31 -0.58 -0.36 -0.4  -0.01]
 [ 0.16  0.3  -0.21 -0.31  0.37 -0.49 -0.04  0.6 ]
 [ 0.12  0.23 -0.16 -0.23  0.28 -0.37 -0.01 -0.8 ]
 [ 0.24 -0.24 -0.26 -0.33  0.2   0.45 -0.68 -0.01]]
```

Notice how the first six eigenvectors are very similar for both algorithms, but the last two are quite different. These correspond to the smallest eigenvalues. Let's compute the dot product of the last two eigenvectors for both algorithms:

In [25]:
```
print("Dot product of last two eigenvectors obtained with np.linalg.eig:")
print(np.dot(evecs[:,6], evecs[:,7]))

print()

print("Dot product of last two eigenvectors obtained with np.linalg.eigh:")
print(np.dot(evecsh[:,6], evecsh[:,7]))
```

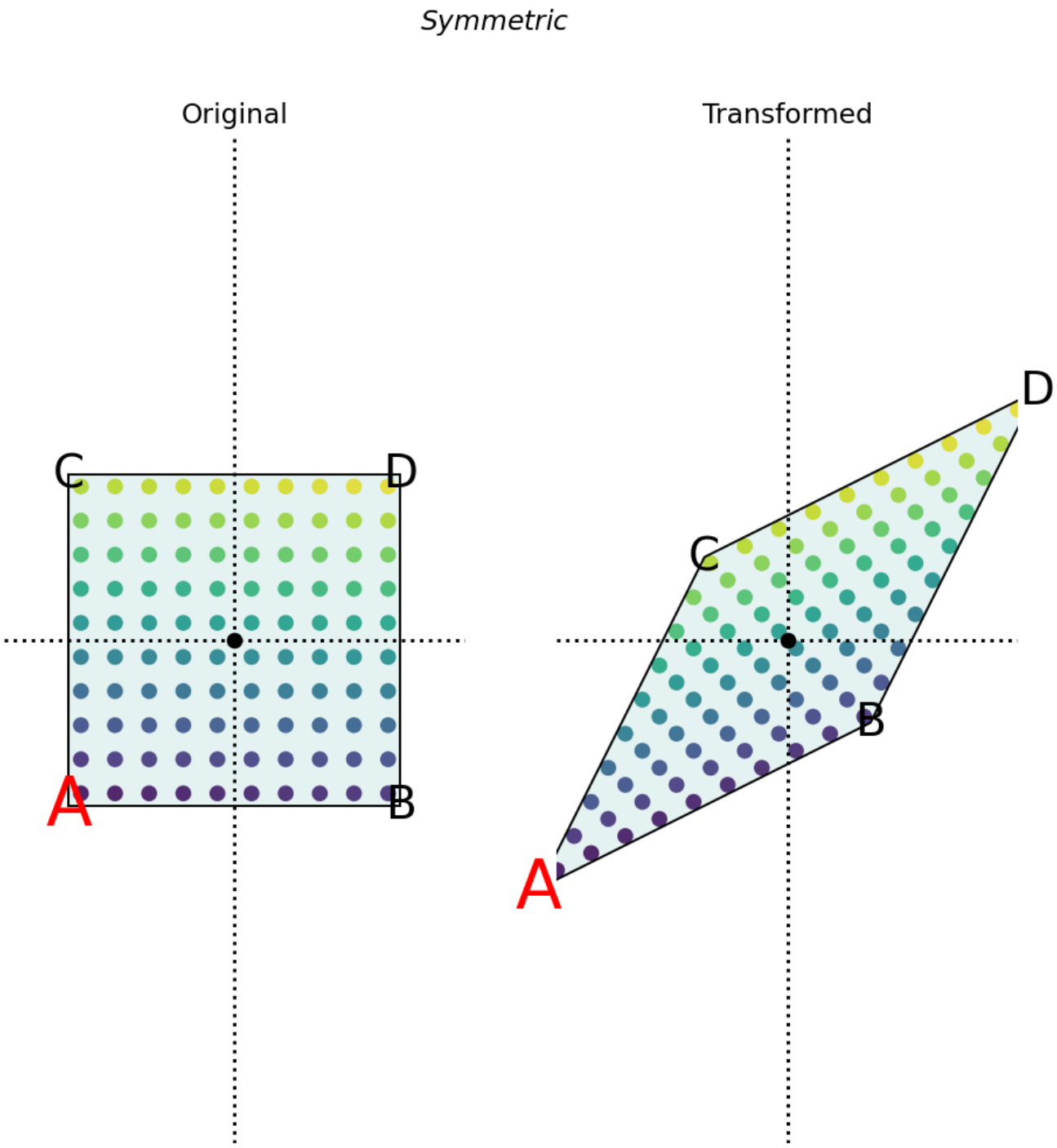
```
Dot product of last two eigenvectors obtained with np.linalg.eig:
0.12491457428359586

Dot product of last two eigenvectors obtained with np.linalg.eigh:
-1.457167719820518e-16
```

Notice how rounding errors have caused the last two eigenvectors from `np.linalg.eig` to be non-orthogonal, whereas the more stable `np.linalg.eigh` yields orthogonal eigenvectors with a dot product close to zero.

Eigenvectors and linear maps - more intuition

In [26]:
```
A = np.array([[1,0.5],[0.5,1]])
#A = np.array([[1,0],[0,1]])
#A = np.array([[1,0],[1,0]])

show_matrix_effect(A, "Symmetric")   # rotate, then scale

evals, evecs = np.linalg.eig(A) # compute eigendecomposition of A using np.linalg.eig
order = np.argsort(-np.abs(evals)) # Get the indices of the eigenvalues in order of largest t
evals = evals[order] # Reorder eigenvalues and eigenvectors from largest to smallest eigenval
evecs = evecs[:, order] # # Reorder eigenvalues and eigenvectors from largest to smallest eig
print("Eigenvectors computed with np.linalg.eig:") # Display eigenvectors x_i (columns) in or
print_matrix("\\bf{x_i}", evecs)
print_matrix("\lambda_i", evals)
```

```
Symmetric
 [[1.  0.5]
 [0.5 1. ]]
Eigenvectors computed with np.linalg.eig:
\bf{x_i}
 [[ 0.71 -0.71]
 [ 0.71  0.71]]
\lambda_i
 [[1.5 0.5]]
```



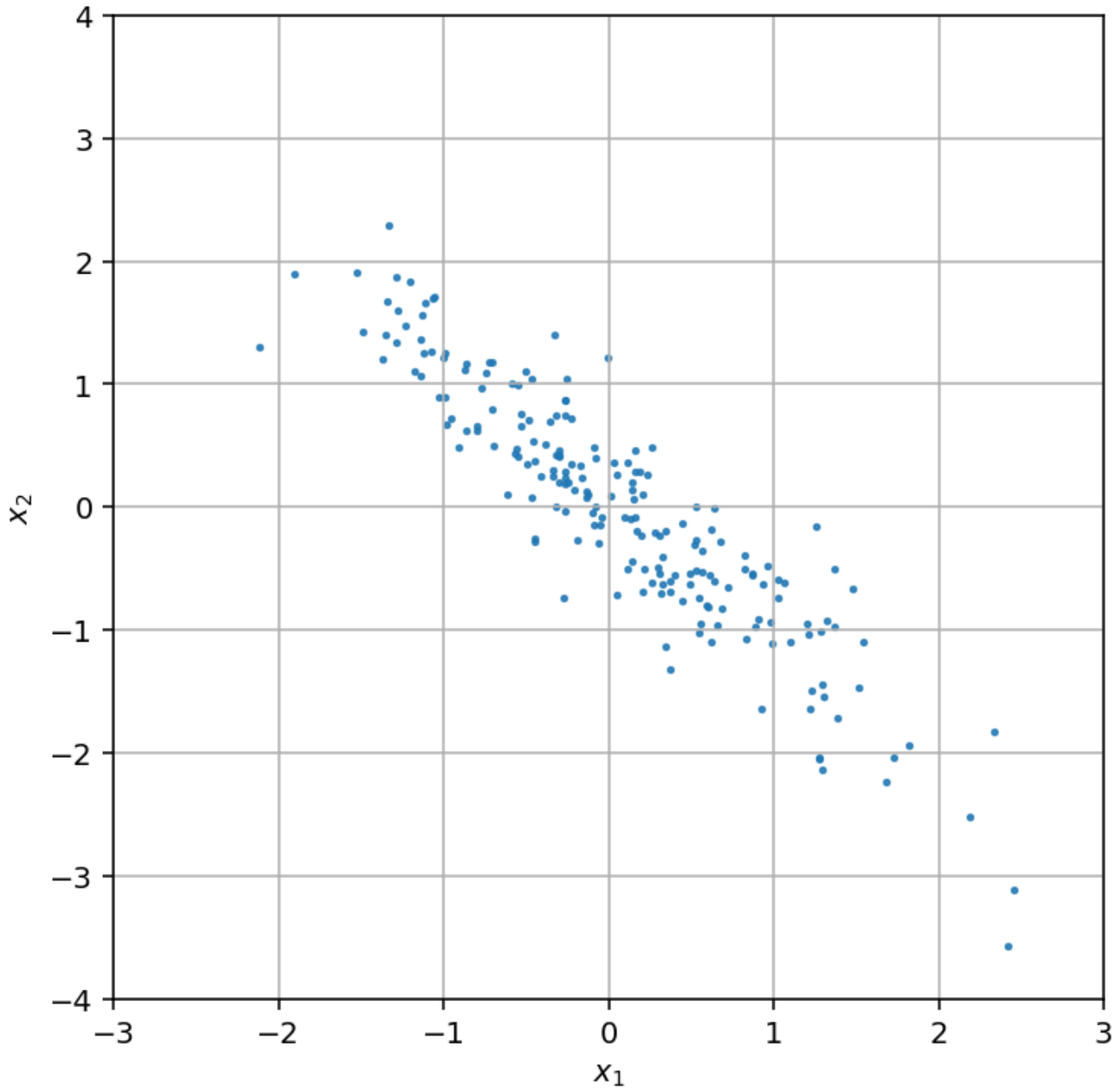
Principal Component Analysis (PCA) - analysing data with linear matrices

Let's generate a dataset in $\mathbb{R}^2$:

In [27]:
```
# Construct a dataset, collect it in a matrix of 200 random data points.vectors (x_1, x_2), wi
x = np.random.normal(0, 1, (200, 2)) @ np.array([[0.05, 0.4], [-0.9, 1.0]])  # don't try to ui

print("First 10 observations...\n%s" % x[0:10,:])

# plot the dataset
fig = plt.figure(figsize=(6, 6))
ax = fig.add_subplot(1, 1, 1)
```

```
ax.scatter(x[:, 0], x[:, 1], c='C0', label="Original data", s=3, alpha=0.8)
ax.set_xlim(-3, 3)
ax.set_ylim(-4, 4)
#
ax.grid()
plt.xlabel('$x_1$')
plt.ylabel('$x_2$');
```

```
First 10 observations...
[[ 1.26042045 -0.15419179]
 [ 0.97761239 -0.94162265]
 [-0.26038222  0.86320063]
 [-1.12564576  1.25784762]
 [ 1.36457186 -0.50185491]
 [ 0.04576022  0.2567817 ]
 [-0.22723504  0.34480761]
 [-0.86273032  0.62623212]
 [-1.52336285  1.90224105]
 [-2.1183187   1.3012353 ]]
```



Mean vector

Given our straightforward definition of vectors, we can define some **statistics** that generalise the statistics of ordinary real numbers. These just use the definition of vector addition and scalar multiplication, along with the outer product.

The **mean vector** of a collection of N vectors is the sum of the vectors multiplied by $\frac{1}{N}$:

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

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.

If we have vectors stacked up in a matrix X, one vector per row, `np.mean(x, axis=0)` will calculate the mean vector for us.

Covariance matrix

As well as the **mean vector**, we can also generalise the idea of **variance**, which measures the spread of a dataset, to the multidimensional case. Variance (in the 1D case, i.e. $N \times 1$) is the sum of squared differences of each element from the mean of the vector:

$$\sigma^2 = \frac{1}{N-1} \sum_{i=0}^{N-1} (\mathbf{x_i} - \mu)^2$$

,

where μ is the mean of this $N \times 1$ data. This is a measure of how "spread out" a vector of values $\mathbf{x}$ is. The **standard deviation** σ is the square root of the **variance** and is more often used because it is in the same units as the elements of $\mathbf{x}$.

In the multidimensional case, to get a useful measure of spread of a $N \times d$ data matrix X (N d-dimensional vectors) we need to compute the *covariance* of every dimension with every other dimension. This is the average squared difference of each column of data from the average of every column. This forms a 2D array Σ, which has entries in element i, j:

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

,

where μ_i is the mean of dimension i. It is a *special form* of matrix: it is square, symmetric and positive semi-definite.

It tells us *how much correlation* there is between the variables in a data set. We can plot a representation of the covariance matrix as an ellipse which aligns with the distribution of the data points. Uncorrelated data is represented by a circle, whereas strongly correlated data is represented by a long thin ellipse.

Hint: The eigenvectors of the covariance matrix, scaled by their eigenvalues, form the principal axes of the ellipse.

Decomposition of the covariance matrix into its eigenvectors and eigenvalues

The eigenvectors of the covariance matrix are called the **principal components**, and 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.

The direction of principal component i is given by the eigenvector $\mathbf{x}_i$, and the length of the component is given by $\sqrt{\lambda_i}$.

Now we calculate the covariance matrix:

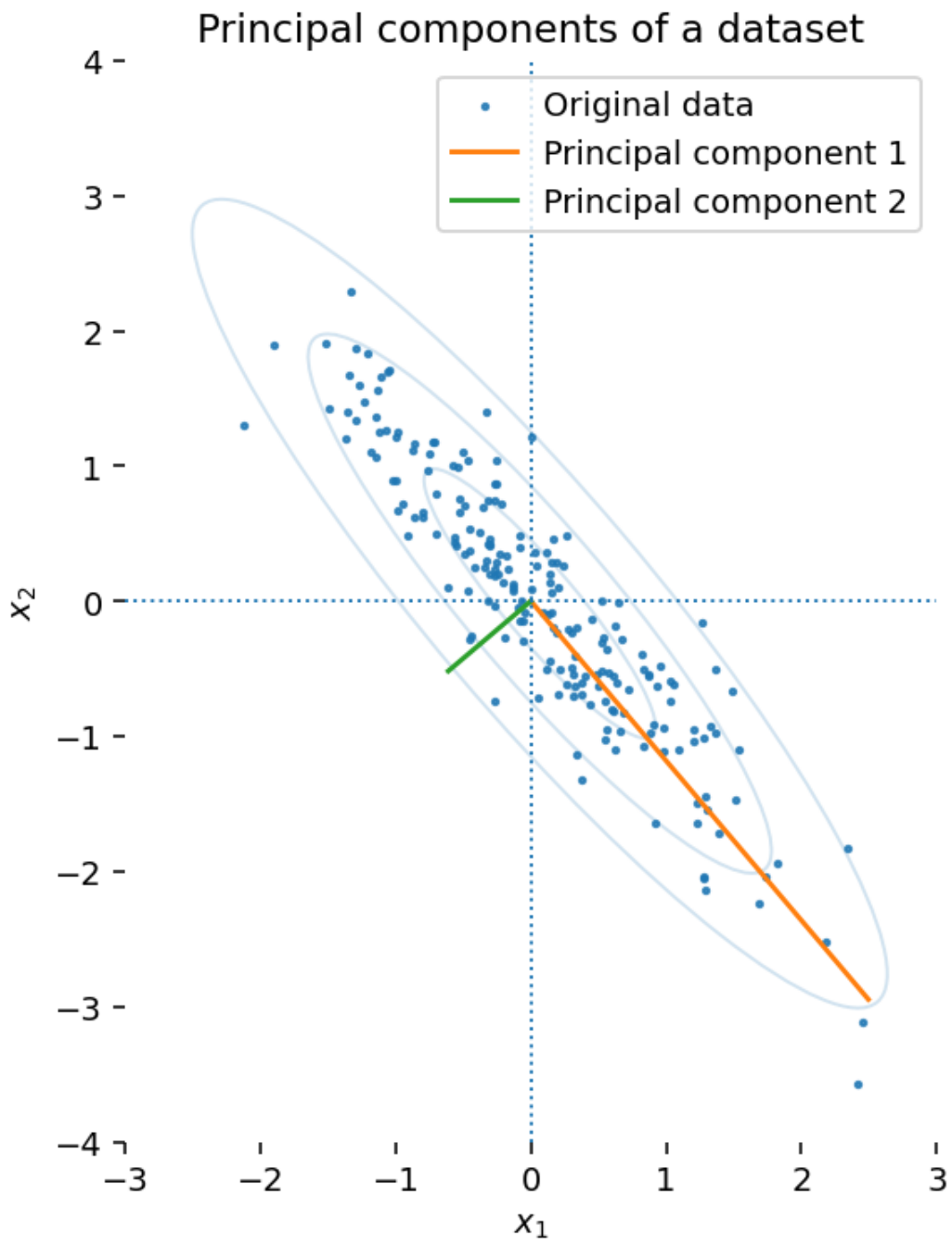
In [28]:
```
# Compute the covariance matrix for x and display it
print_matrix("\Sigma", np.cov(x, rowvar=False))  # rowvar=False means variables are stored as
```

```
\Sigma
 [[ 0.73 -0.78]
 [-0.78  0.99]]
```

Calculate the eigenvectors and eigenvalues of the covariance matrix and plot these as the principal axes of an ellipse:

In [29]:
```
# These are helper functions. You do not need to undersatnd these in detail.
from matplotlib.patches import Ellipse

def eigsorted(cov):
    vals, vecs = np.linalg.eigh(cov)
    order = vals.argsort()[::-1]
    return vals[order], vecs[:, order]

def cov_ellipse(ax, x, nstd=1, **kwargs):
    cov = np.cov(x.T)
    mu = np.mean(x, axis=0)
    vals, vecs = eigsorted(cov)
    theta = np.degrees(np.arctan2(*vecs[:, 0][::-1]))
    w, h = 2 * nstd * np.sqrt(vals)
    ell = Ellipse(xy=(mu[0], mu[1]), width=w, height=h, angle=theta, **kwargs)

    ax.add_artist(ell)
    return mu,cov
```

In [30]:
```
#from jhwutils import ellipse as ellipse

# Compute the eigenvalues and eigenvectors of the 2D covariance matrix
evals, evecs = np.linalg.eig(np.cov(x, rowvar=False))

evals_sorted = evals[np.argsort(-np.abs(evals))]
evecs_sorted = evecs[:, np.argsort(-np.abs(evals))]

# Plot the dataset
fig = plt.figure(figsize=(6, 6))
ax = fig.add_subplot(1, 1, 1)
ax.scatter(x[:, 0], x[:, 1], c='C0', label="Original data", s=3, alpha=0.8)
plt.xlabel('$x_1$')
plt.ylabel('$x_2$')

# Plot ellipses to show covariance of data
cov_ellipse(ax, x[:, 0:2], 1, facecolor='none', edgecolor='C0', alpha=0.2)
cov_ellipse(ax, x[:, 0:2], 2, facecolor='none', edgecolor='C0', alpha=0.2)
cov_ellipse(ax, x[:, 0:2], 3, facecolor='none', edgecolor='C0', alpha=0.2)

# Plot eigenvectors of covariance matrix * 3 (so they reach the radius of the 3rd ellipse)
def plot_evec(ix):
    ax.plot([0, evecs_sorted[0, ix] * 3 * np.sqrt(evals_sorted[ix])],
            [0, evecs_sorted[1, ix] * 3 * np.sqrt(evals_sorted[ix])], color='C'+str(ix+1),
             label='Principal component {ix}'.format(ix=ix+1))

plot_evec(0)
plot_evec(1)
# fix details
ax.set_xlim(-3, 3)
ax.set_ylim(-4, 4)

ax.axhline(0, lw=1, ls=':')
ax.axvline(0, lw=1, ls=':')
ax.set_aspect(1.0)
ax.legend()
ax.set_frame_on(False)
ax.set_title("Principal components of a dataset");
```



In Euclidean space, the principal components of a 2D dataset are its semi-major and semi-minor axes.

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. We can do this as follows:

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

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).

Let's try it:

In [31]:
```
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 =
[[-0.76306849  0.64631763]
 [-0.64631763 -0.76306849]]

Eigenvalues =
[0.0711517  1.65694998]

[[ 0.73358164 -0.78209132]
 [-0.78209132  0.99452004]]
Q
 [[-0.76  0.65]
 [-0.65 -0.76]]
\Lambda
 [[0.07 0.  ]
 [0.   1.66]]
\Sigma = Q\Lambda Q^T
 [[ 0.73 -0.78]
 [-0.78  0.99]]
```

Hey presto! We've recovered the covariance matrix.

Approximating a matrix

Imagine we started with a very high dimensional data set, so Σ 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 and use these to reconstruct an *approximation* to Σ. Providing we keep the largest principal components, we will probably retain most of the information.

Let's demonstrate this with our 2D covariance matrix. We can approximate the matrix by *truncating* the list of eigenvalues and eigenvectors, i.e. setting the smallest of the eigenvalues and eigenvectors to 0.

In [32]:
```
print("Eigenvectors =\n%s\n" % evecs)
print("Eigenvalues =\n%s\n" % evals)
#print(np.linalg.norm(evecs[:,1]))

# truncate small eigenvalues to 0
approx_evals = np.where(np.abs(evals)<1e-1, 0, evals)
print_matrix("\hat{\Lambda}", np.diag(approx_evals))

# construct an approximation to A using the truncated eigenvalue matrix
approx_A = evecs @ np.diag(approx_evals) @ evecs.T
print_matrix("\hat{\Sigma} = Q \hat{\Lambda}Q", approx_A)
```

```
Eigenvectors =
[[-0.76306849  0.64631763]
 [-0.64631763 -0.76306849]]

Eigenvalues =
[0.0711517  1.65694998]

\hat{\Lambda}
 [[0.   0.  ]
 [0.   1.66]]
\hat{\Sigma} = Q \hat{\Lambda}Q
 [[ 0.69 -0.82]
 [-0.82  0.96]]
```

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

The **eigenspectrum** gives us an idea of how simply a matrix could be approximated:

- 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.

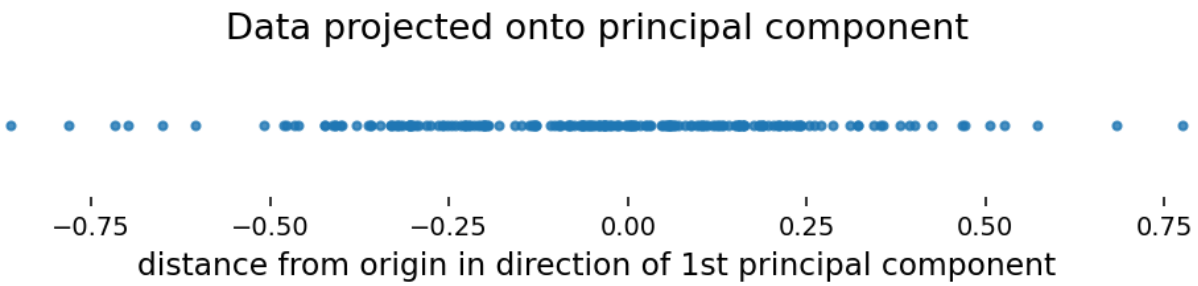
Dimensionality reduction

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.

Let's use our highly correlated 2D data set as an example. Above, we discarded the smaller of the two eigenvectors, leaving us with a single eigenvalue/eigenvector pair. We can project the original 2D dataset onto the remaining eigenvector to yield a 1D dataset as follows:

In [33]:
```
x_projected = x @ evecs[:, 0] # remember x is a matrix with row vectors

print("The eigenvector (row vector) \n%s\n" %evecs[:, 1])
print("A few data points\n%s\n" % x[0:3, :])

# Plot the dataset
fig = plt.figure(figsize=(9, 1))
ax = fig.add_subplot(1, 1, 1)
ax.scatter(x_projected, np.zeros_like(x_projected), c='C0', label="Original data", s=10, alpha
ax.set_xlabel('distance from origin in direction of 1st principal component', fontsize=12)
ax.set_yticks(np.arange(0))
ax.set_frame_on(False)
plt.title('Data projected onto principal component', fontsize=14);
```

```
The eigenvector (row vector)
[ 0.64631763 -0.76306849]

A few data points
[[ 1.26042045 -0.15419179]
 [ 0.97761239 -0.94162265]
 [-0.26038222  0.86320063]]
```



In the example above, we reduced a 2D dataset to 1D, but a more common application is the reduction of a high dimensional data set to 2D. If we can reduce a dataset to 2D, we can visualise it by plotting it in a 2D coordinate system. This often reveals structure in the data, such as clusters of data points. We will explore this in this week's lab.

Uses of eigendecomposition

Matrix decomposition is an *essential* tool in data analysis. It can be extremely powerful and is efficient to implement. Systems such as recommenders (e.g. Netflix, YouTube, Amazon, etc.), search engines (Google), image compression algorithms, machine learning tools and visualisation systems apply these decompositions *extensively*.

Google was wholly built around matrix decomposition algorithms; that's what "PageRank" is. This is what allowed Google to race ahead of their competitors in the early days of the search wars.

The eigendecomposition can be used *anywhere* there is a system modelled as a linear transform (any linear map $\mathbb{R}^N \to \mathbb{R}^N$). It lets us predict behaviour over different time scales (e.g. very short term or very long term). For instance, we can:

- Find "modes" or "resonances" in a system (e.g physical model of a bridge).
  - For example, every room has a set of "eigenfrequencies" - the acoustic resonant modes. A linear model of the acoustics of the room could be written as a matrix, and the resonant frequencies extracted directly via the eigendecomposition. Alvin Lucier's analogue power iterations
- Predict the behaviour of feedback control systems: is the autopilot going to be stable or unstable?
- Partition graphs and cluster data (spectral clustering).
- Predict flows on graphs.
- Perform Principal Component Analysis on high-dimensional data sets for exploratory data analysis, 2D visualisation or data compression.

As soon as we can write down a matrix A we can investigate its properties with the eigendecomposition. ***It is the microscope of the linear algebraist.***

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:

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

It is also equal to the sum of the eigenvalues of A

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

The trace can be thought of as measuring the **perimeter** of the parallelotope of a unit cube transformed by the matrix. [Strictly, it is *proportional* to the perimeter, with the constant of proportionality being $\mathrm{Perimiter}(A) = 2^{n-1}\mathrm{Tr}(A)$].

Determinant

The determinant $\det(A)$ is an important property of square matrices. It can be thought of as the **volume** of the parallelotope of a unit cube transformed by the matrix -- it measures how much the space expands or contracts after the linear transform.

It is equal to the product of the eigenvalues of the matrix.

$$\det(A) = \prod_{i=1}^{n} \lambda_i \quad [\spadesuit]$$

If any eigenvalue λ_i of A is 0, the determinant $\det(A) = 0$, and the transformation collapses at least one dimension to be completely flat. This means that 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 positive definite matrix A has the property $\mathbf{x}^T A\mathbf{x} > 0$ for all (nonzero) $\mathbf{x}$. This tells us that the dot product of $\mathbf{x}$ with $A\mathbf{x}$ must be positive (N.B. $A\mathbf{x}$ is the vector obtained by transforming $\mathbf{x}$ with A). This can only happen if the angle θ between $\mathbf{x}$ and $A\mathbf{x}$ is less than $\frac{\pi}{2}$, since $\mathbf{x}^T A\mathbf{x} = |\mathbf{x}||A\mathbf{x}| \cos\theta$. That means that A does not rotate $\mathbf{x}$ through more than 90°.

Positive definiteness will be an important concept when we discuss **covariance matrices** (important in statistical data analysis) and **Hessian matrices** (important in numerical optimisation).

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.

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 represent 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

We have seen four basic algebraic operations on matrices:

- scalar multiplication cA;
- matrix addition $A + B$;
- matrix multiplication BA
- matrix transposition A^T

There is a further important operation: **inversion** A^{-1}, defined such that:

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

The equivalent of division for matrices is left-multiplication by the inverse. This has the effect reversing or undoing the effect of the original matrix.

Inversion is only defined for certain kinds of matrices, as we will see below.

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:

In [34]:
```
# Create a matrix A of random numbers
A = np.random.normal(0, 1, (3, 3))
print_matrix("A", A)
# Now invert it
print_matrix("A^{-1}", np.linalg.inv(A))
```

```
A
 [[-0.81 -1.23 -1.08]
 [-0.63  0.45 -0.24]
 [-0.52 -0.44  0.26]]
A^{-1}
 [[-0.01 -0.87 -0.85]
 [-0.31  0.84 -0.53]
 [-0.56 -0.31  1.25]]
```

Inversion as "undo"

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

In [35]:
```
# Verify that left-multiplying by the inverse "undoes" the transformation applied by A to a r
# Create a random vector
x = np.random.normal(0, 1, (3, 1))
print_matrix("x", x)
```

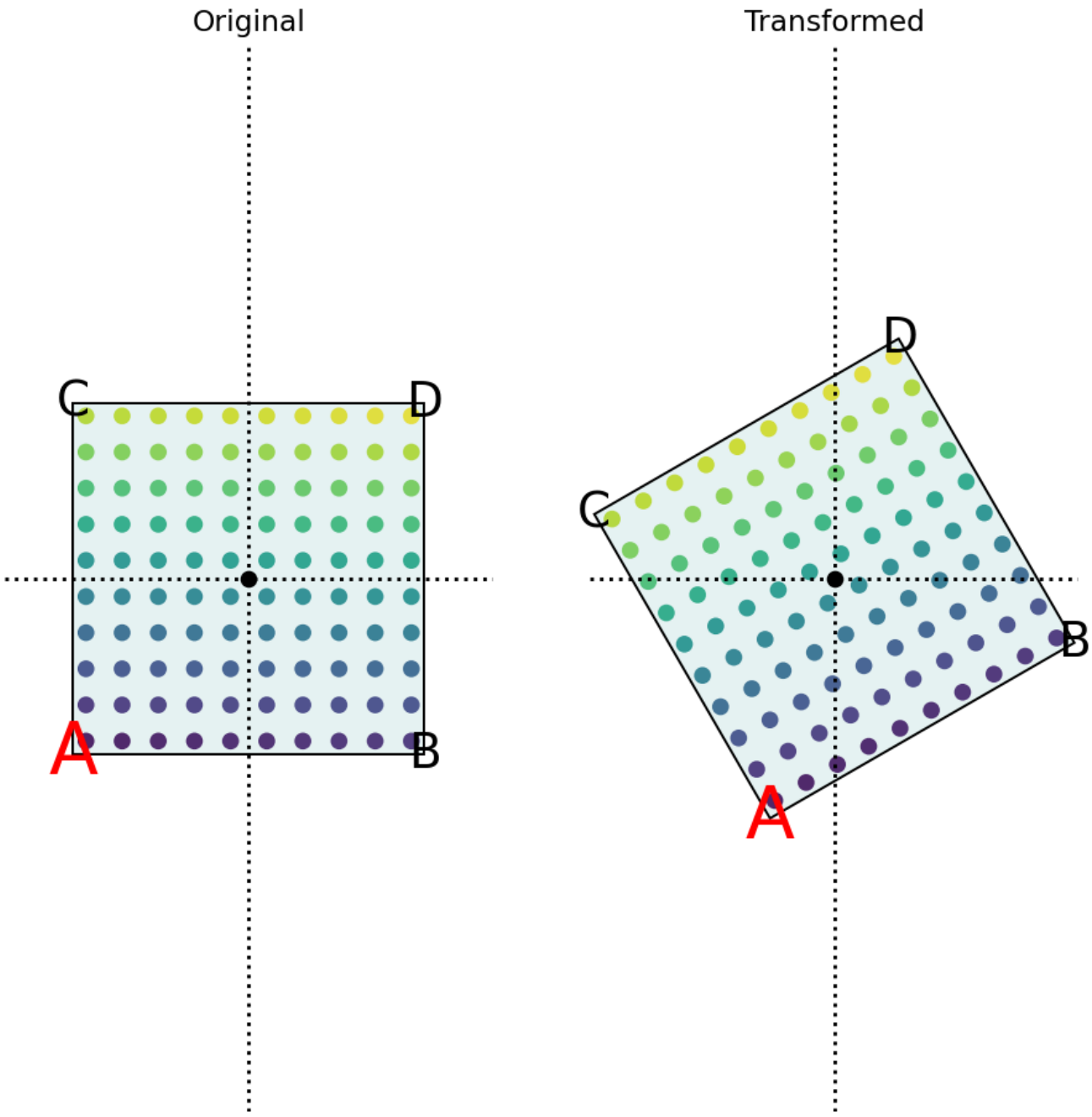
```
# Tranform it
print_matrix("Ax", A @ x)
# Left-multiply by the inverse to recover the original vector
print_matrix("A^{-1}(Ax)", np.linalg.inv(A) @ (A @ x))
```

```
x
 [[-1.2 ]
 [-0.67]
 [-1.95]]
Ax
 [[3.9 ]
 [0.92]
 [0.4 ]]
A^{-1}(Ax)
 [[-1.2 ]
 [-0.67]
 [-1.95]]
```

Inversion creates a matrix that "undoes" the transformation performed by another matrix, so long as no information was lost in the transformation. We can illustrate this geometrically:

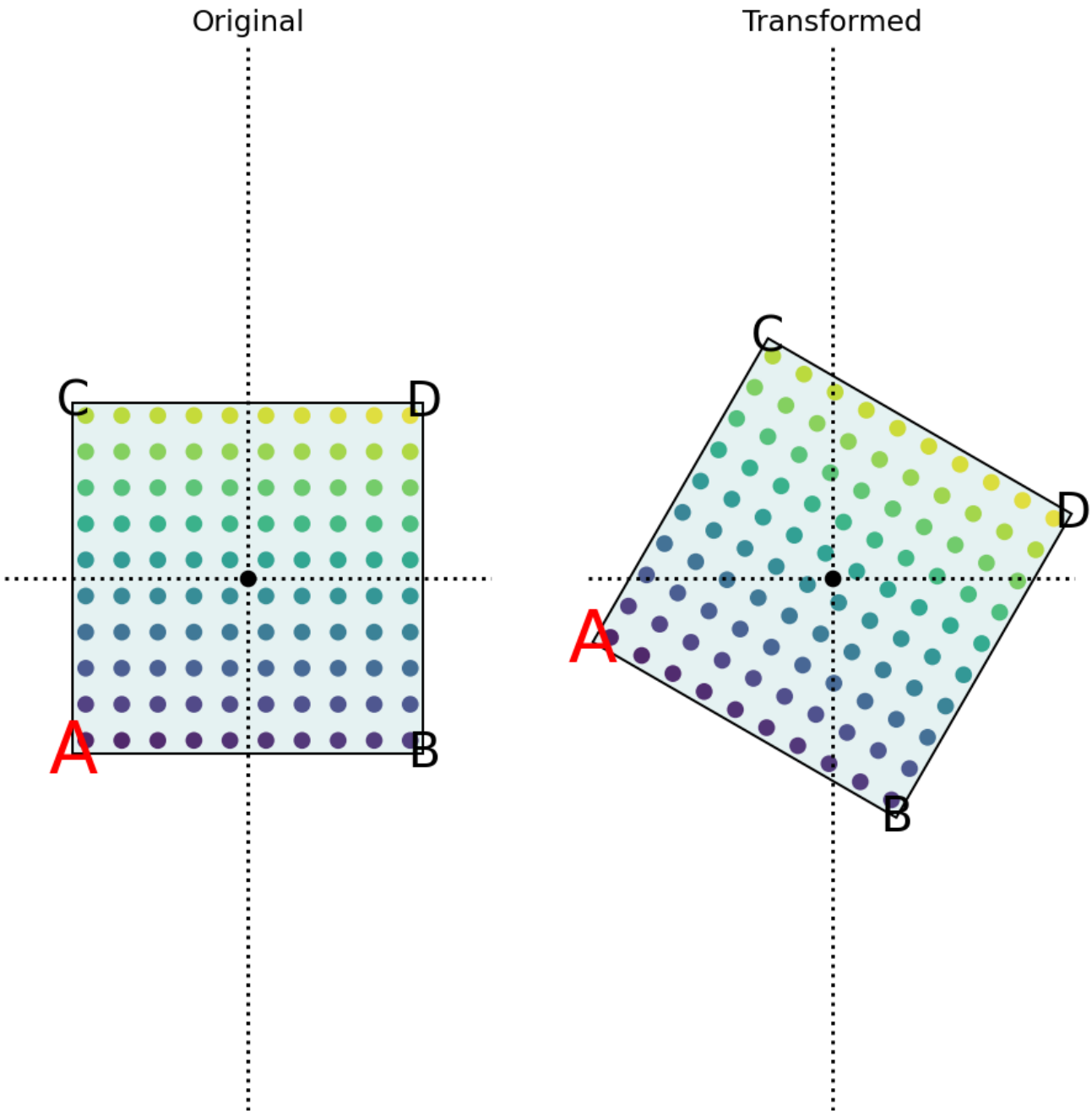
In [36]:
```
# Create a matrix that performs a 30 degree rotation
d30 = np.radians(30)
cs = np.cos(d30)
ss = np.sin(d30)
rot_mat = np.array([[cs, -ss], [ss, cs]])
# Display the effect of this matrix
show_matrix_effect(rot_mat)
```

```
 [[ 0.87 -0.5 ]
 [ 0.5   0.87]]
```



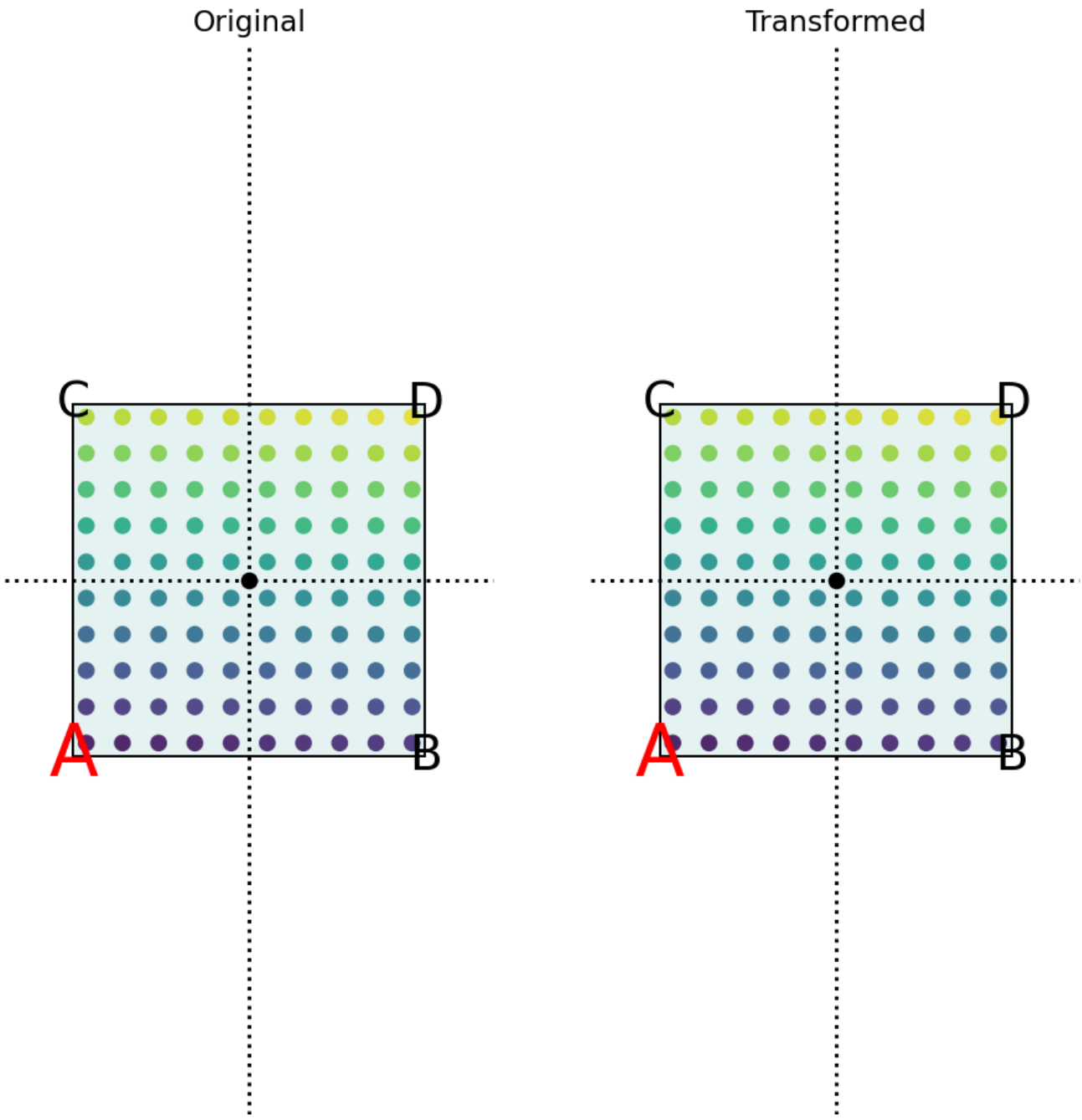
In [37]:
```
# Compute the inverse of the 30 degree rotation matrix and display its effect
show_matrix_effect(np.linalg.inv(rot_mat))
```

```
[[ 0.87   0.5 ]
[-0.5    0.87]]
```



In [38]:
```
# Show the combined effect of the 30 degree rotation matrix, left-multiplied by its inverse
show_matrix_effect(np.linalg.inv(rot_mat) @ rot_mat)
```

```
[[1. 0.]
[0. 1.]]
```



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: $\det(A) \neq 0$. Why?

A matrix which is non-square maps vectors of dimension m to dimension n. This means the transformation collapses or creates dimensions. Such a transformation is not uniquely reversible.

For a matrix to be invertible it must represent a **bijection** (a function that maps every member of a set onto exactly one member of another set).

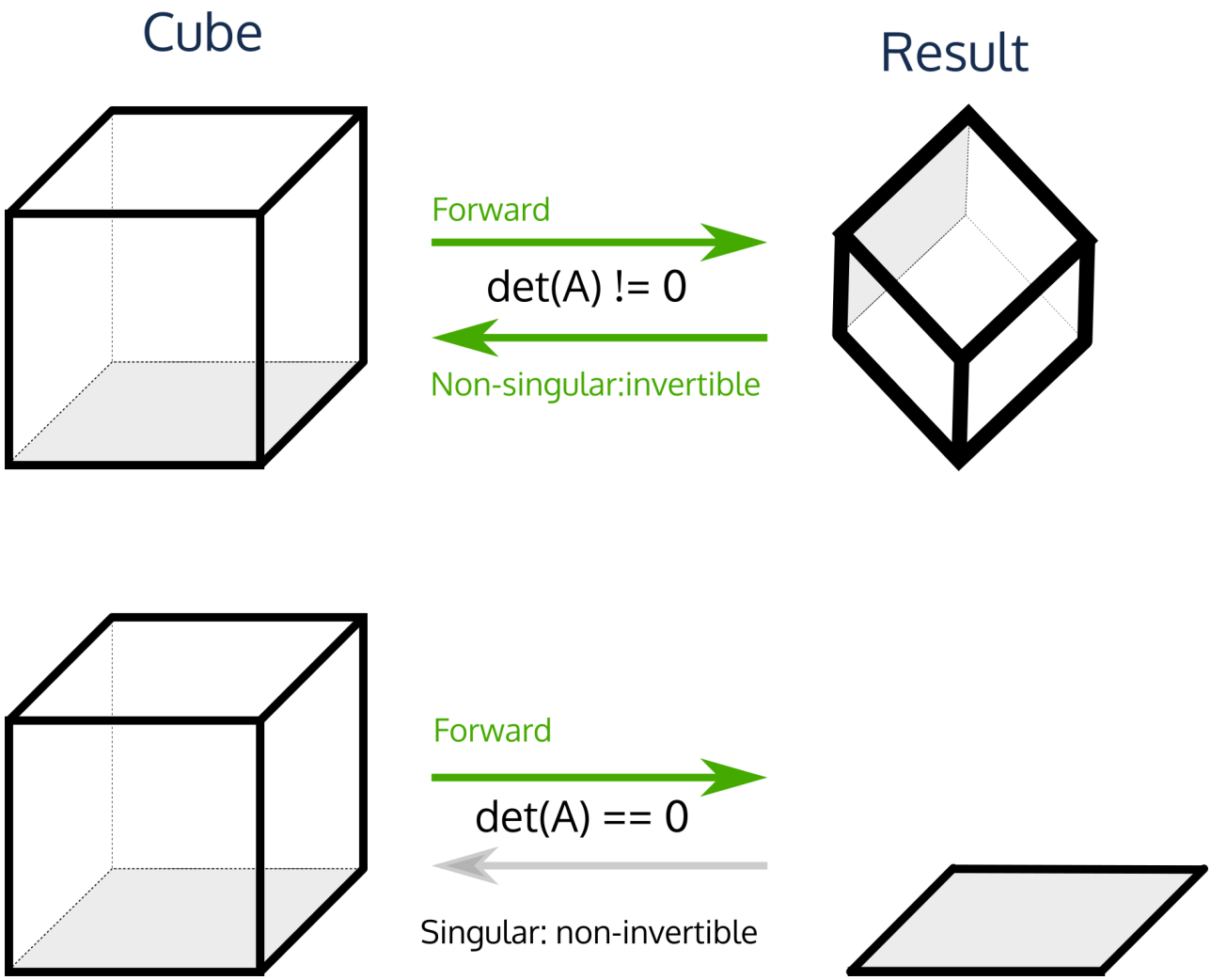
Singular and non-singular matrices

A matrix with $\det(A) = 0$ is called **singular** and has no inverse.

A matrix which is invertible is called **non-singular**.

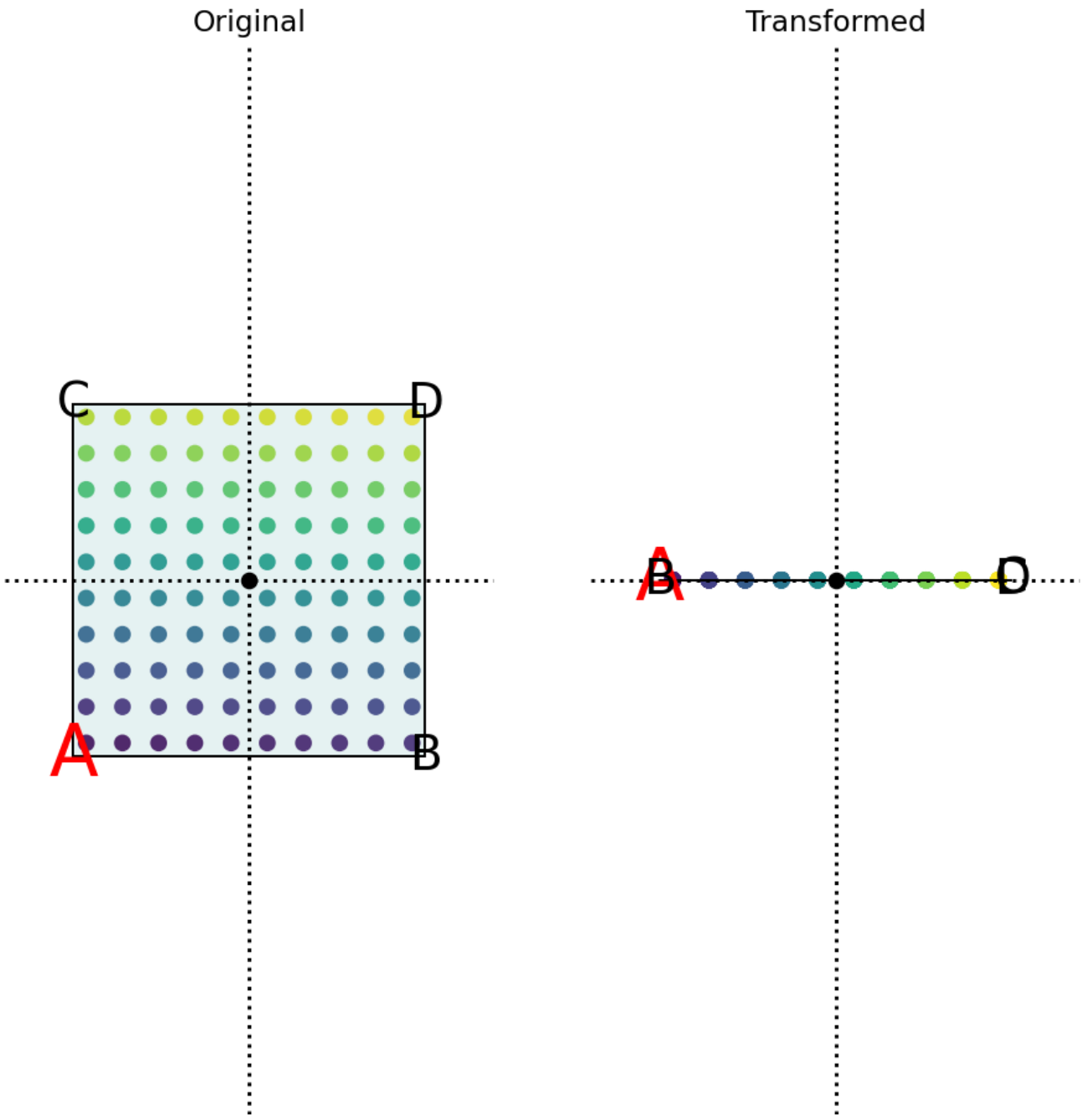
The geometric intuition for this is simple. Going back to the paralleogram model, a matrix with zero determinant has at least one zero eigenvalue. This means that at least one of the dimensions of the parallelepiped has been squashed to nothing at all. Therefore it is impossible to reverse the transformation, because information was lost in the forward transform.

All of the original dimensions must be preserved in a linear map for inversion to be meaningful; this is the same as saying $\det(A) \neq 0$.



In [39]:
```
# The following matrix is singular
sing_mat = np.array([[0.0, 1],
                     [0, 0.0]])
# Display its effect
show_matrix_effect(sing_mat)
```

```
[[0. 1.]
[0. 0.]]
```



In [40]:
```
#Let's not run it

## Attempting to invert a singular matrix throws an error
#try:
#    show_matrix_effect(np.linalg.inv(sing_mat))
#except:
#    print("LinAlgError: Singular matrix")
```

Numerical stability of matrix inversion algorithms

Because matrix operations involve *lots* of repeated floating point operations there are many opportunities for roundoff to accumulate. There is a great deal of importance in finding matrix algorithms which are **numerically stable**; that is they converge to the right answer reliably.

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 $O(n^3)$ (one of the few problems for which a tight polynomial time bound is known).

Special cases

Just as for multiplication, there are many special kinds of matrices for which much faster inversion is possible. These include, among many others:

- orthogonal matrix (rows and columns are all orthogonal unit vectors): $O(1)$, $A^{-1} = A^T$
- diagonal matrix (all non-diagonal elements are zero): $O(n)$, $A^{-1} = \frac{1}{A}$ (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**.

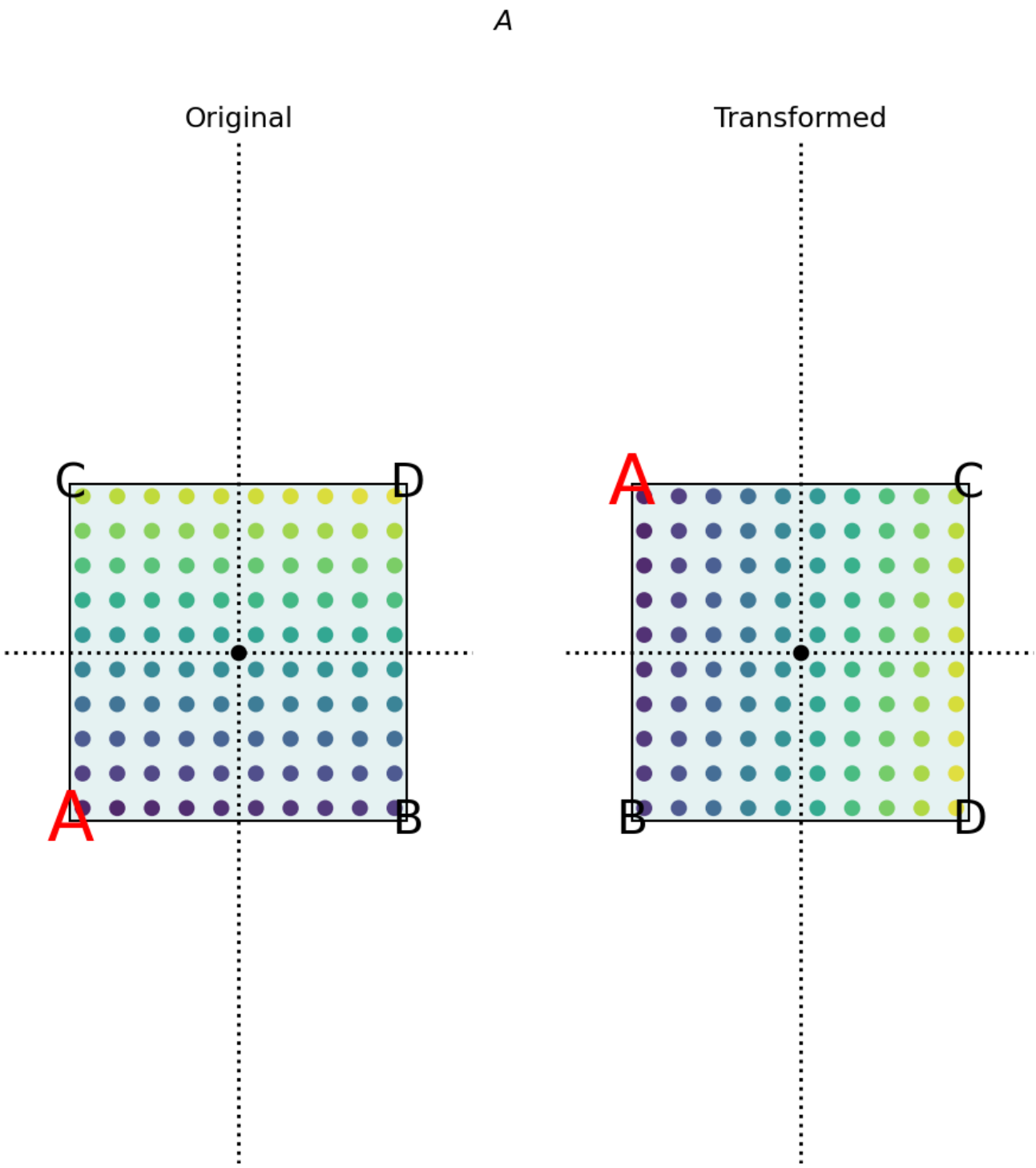
Special cases: orthogonal matrices

An **orthogonal matrix** is a special matrix form that has $A^T = A^{-1}$; that is the transpose and the inverse are equivalent. All of its component eigenvectors are **orthogonal** to each other (at 90 degrees; have an inner product of 0), and all of its eigenvalues are 1 or -1. An orthogonal matrix transforms a cube to a cube. It has a determinant of 1 or -1. Any purely rotational matrix is an orthogonal matrix.

Orthogonal matrices can be inverted trivially, since transposition is essentially free in computational terms.

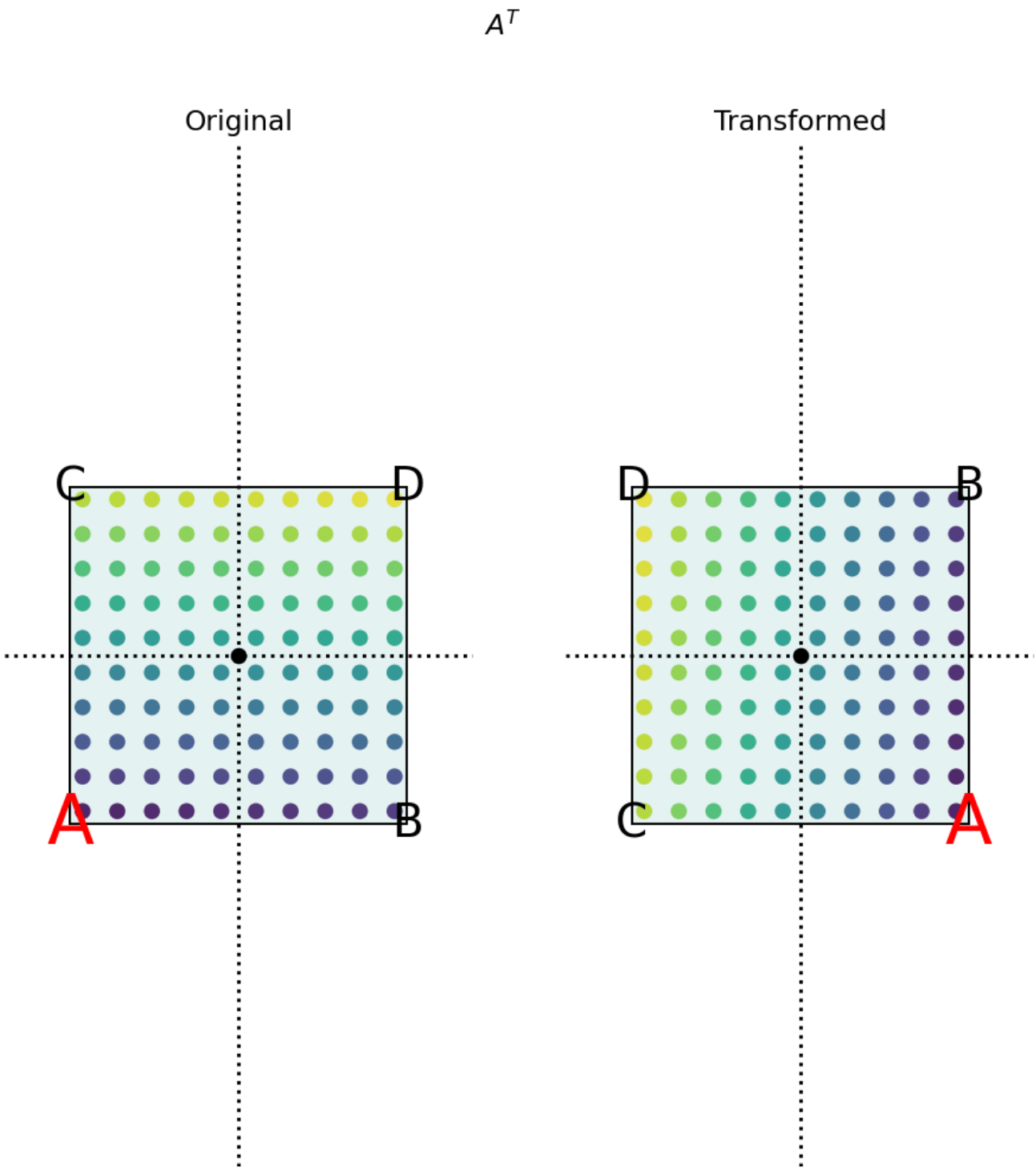
In [41]:
```
# Create an orthogonal matrix and show its effect
A = np.array([[0, 1, 0],
              [-1, 0, 0],
              [0, 0, 1]])
show_matrix_effect(A, "A")
```

```
A
 [[ 0  1  0]
 [-1  0  0]
 [ 0  0  1]]
```



In [42]:
```
# Show the effect of its transpose and show that the inverse is the same as the transpose
show_matrix_effect(A.T, "A^T")
print_matrix("A^{-1}", np.linalg.inv(A))
```

```
A^T
 [[ 0 -1  0]
 [ 1  0  0]
 [ 0  0  1]]
A^{-1}
 [[-0. -1. -0.]
 [ 1.  0.  0.]
 [ 0.  0.  1.]]
```



Special cases: Diagonal matrices

The inverse of a diagonal matrix is another diagonal matrix whose diagonal elements are the reciprocal of each of the diagonal elements of the original matrix:

$$A_{ii}^{-1} = \frac{1}{A_{ii}}$$

This is $O(n)$ to compute, so much faster than standard inversion.

In [43]:
```
# Create a diagonal matrix
d = np.diag([-1, 2, -3, 4])
print_matrix('D', d)
# Compute the reciprocal, elementwise
print_matrix('\\frac{1}{D}', np.diag(1/np.diag(d)))
# Comput the inverse
print_matrix('D^{-1}', np.linalg.inv(d))
```

```
D
 [[-1  0  0  0]
 [ 0  2  0  0]
 [ 0  0 -3  0]
 [ 0  0  0  4]]
\frac{1}{D}
 [[-1.    0.    0.    0.  ]
 [ 0.    0.5   0.    0.  ]
 [ 0.    0.   -0.33  0.  ]
 [ 0.    0.    0.    0.25]]
D^{-1}
 [[-1.   -0.   -0.   -0.  ]
 [ 0.    0.5   0.    0.  ]
 [-0.   -0.   -0.33 -0.  ]
 [ 0.    0.    0.    0.25]]
```

Issue with inversion of sparse matrices

The inverse of a **sparse matrix** is in general **not sparse**; it will (most likely) be dense. This means that sparse matrix algorithms virtually never involve a direct inverse, as 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!

Revisting the package distribution problems: "predicting" the past with inversion

Revisiting the problem of package distribution, we can use inversion to solve the problem of predicting the distribution at $\mathbf{x_{t=-1}}$ given $\mathbf{x_{t=0}}$. Applying A to $\mathbf{x}$ took us "one step" into the future. Applying A^{-1} takes us one step into the past:

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

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

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

However, be aware that inversion is highly problematic from a numerical point of view, and roundoff error will lead to severely distorted results with repeated negative powers of a matrix.

Linear systems

Imaginge a system where there is an input, x, and an output, y.

One way of looking at matrices is as a collection of weights of components of a vector. Consider:

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

This represents a linear map operating on 3D vectors $\mathbf{x} \in \mathbb{R}^3$. It produces a 3D vectors $\mathbf{y} \in \mathbb{R}^3$, each component of which is a **weighted sum** of the components of the input vector.

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

The coefficients of the matrix represent the weighting to be applied.

In [44]:
```
A = np.array([[0.5, 1.0, 2.0],
              [1.0, 0.5, 0.0],
              [0.6, 1.1, -0.3]])

x = np.array([[1], [-1], [1]])

print_matrix("A", A)
print_matrix("\\bf x", x)
print_matrix("y=A{\\bf x}", A @ x)
```

```
A
 [[ 0.5  1.   2. ]
 [ 1.   0.5  0. ]
 [ 0.6  1.1 -0.3]]
\bf x
 [[ 1]
 [-1]
 [ 1]]
y=A{\bf x}
 [[ 1.5]
 [ 0.5]
 [-0.8]]
```

In the package distribution system:

NB! This is exactly what happens in the package distribution example; the matrix represents the equations that define the flow between depots. You could interpret one row as follows:

$$\begin{bmatrix} 0.9 & 0.0 & 0.0 & 0.2 & 0.0 & 0.05 & 0.08 & 0.0 \end{bmatrix}$$

> Tomorrow, Depot A will have 90% of Depot A's stock, 20% of Depot D's stock, 5% of Depot F's stock and 8% of Depot G's stock.

$$y_1 = 0.9x_1 + 0.2x_4 + 0.05x_6 + 0.08x_7$$

This is the form that simultaneous equations take. If we know A and $\mathbf{y}$, we can ask what $\mathbf{x}$ will satisfy the following equation (or if there even exists an $\mathbf{x}$ that would satisfy it):

$$A\mathbf{x} = \mathbf{y}$$

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

Solving linear systems

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

$$\begin{aligned} A^{-1}A\mathbf{x} &= A^{-1}\mathbf{y} \\ I\mathbf{x} &= A^{-1}\mathbf{y} \\ \mathbf{x} &= A^{-1}\mathbf{y} \end{aligned}$$

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

In [45]:
```
# try for a different y and find what x is
y = np.array([[10], [1], [12]])
A_inv = np.linalg.inv(A)
x_solved = A_inv @ y
print_matrix("A", A)
print_matrix("{\\bf y}", y)
# find the correct x
print_matrix("{\\bf x}=A^{-1}{\\bf y}", x_solved)
```

```
# verify that we have solved these equations by checking that Ax = y
print_matrix("A{\\bf x}", A @ x_solved)
```

```
A
 [[ 0.5  1.   2. ]
 [ 1.   0.5  0. ]
 [ 0.6  1.1 -0.3]]
{\bf y}
 [[10]
 [ 1]
 [12]]
{\bf x}=A^{-1}{\bf y}
 [[-6.03]
 [14.05]
 [-0.52]]
A{\bf x}
 [[10.]
 [ 1.]
 [12.]]
```

Approximate solutions for linear systems

In practice, linear systems are almost never solved with a direct inversion. The numerical problems in inverting high dimensional matrices will make the result highly unstable, and tiny variations in $\mathbf{y}$ might lead to wildly different solutions for $\mathbf{x}$.

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.

These algorithms search the possible space of $\mathbf{x}$ to find solutions that minimise $||A\mathbf{x} - \mathbf{y}||_2^2$ by adjusting the value of $\mathbf{x}$ repeatedly.

The reason these iterative approximation algorithms can work when inversion is numerically impossible is that they only have to solve for one *specific* pair of vectors $\mathbf{x}, \mathbf{y}$. They do not have to create an inversion A^{-1} that inverts the problem for all possible values of $\mathbf{y}$, just the specific $\mathbf{y}$ seen in the problem. This problem is much more constrained and therefore much more stable.

Singular value decomposition

Eigendecompositions only apply to diagonalizable matrices; which are a subset of square matrices. But the ability to "factorise" matrices in the way the eigendecomposition does is enormously powerful, and 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 A. It is the powerhouse of computational linear algebra.

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

$$A = U\Sigma V^T, \quad [\spadesuit]$$

where

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

A **unitary** matrix is one whose conjugate transpose is equal to its inverse. 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.

The diagonal of the matrix Σ is the set of **singular values**, which are closely related to the eigenvalues, but are *not* quite the same thing (except for special cases like when A is a positive semi-definite symmetric matrix)! The **singular values** are always positive real numbers.

We can compute the SVD with `np.linalg.svd`:

In [46]:
```
def print_svd(A):

    # Print A
    print_matrix("A", A)

    # Take SVD
    u, sigma, vt = np.linalg.svd(A)

    # Print U, then norms of rows and norms of columns in U
    print_matrix("U", u)
    print_matrix("\|U\|_{rows}", np.linalg.norm(u, axis=1))
    print_matrix("\|U\|_{cols}", np.linalg.norm(u, axis=0))

    # By default, sigma will be a vector of the diagonal values
    # np.diag(sigma) converts the vector back to a diagonal matrix
    sigma_diag = np.zeros_like(A)
    sqr = min(A.shape[0], A.shape[1])
    sigma_diag[:sqr, :sqr] = np.diag(sigma)

    # Print sigma (as a diagonal matrix)
    print_matrix("\Sigma", sigma_diag)

    # Print V, then norms of rows and norms of columns in V
    print_matrix("V", vt.T)
    print_matrix("\|V\|_{rows}", np.linalg.norm(vt.T, axis=1))
    print_matrix("\|V\|_{cols}", np.linalg.norm(vt.T, axis=0))

    def recompose(u, sigma, vt):
        return u @ sigma @ vt

    # Print recomposed version of A (A = U Sigma V)
    print_matrix("U \Sigma V^T", recompose(u, sigma_diag, vt))
```

In [47]:
```
# Create a square matrix A
A = np.array([[0.5, 2, 3],
              [4,   5, 6],
              [7,   8, 9]])
# Take the SVD and print all the decomposed parts
print_svd(A)
```

```
A
 [[0.5 2.  3. ]
 [4.  5.  6. ]
 [7.  8.  9. ]]
U
 [[-0.2   0.93  0.3 ]
 [-0.52  0.16 -0.84]
 [-0.83 -0.33  0.45]]
\|U\|_{rows}
 [[1. 1. 1.]]
\|U\|_{cols}
 [[1. 1. 1.]]
\Sigma
 [[16.8   0.    0.  ]
 [ 0.    1.44  0.  ]
 [ 0.    0.    0.06]]
V
 [[-0.48 -0.83 -0.3 ]
 [-0.57  0.03  0.82]
 [-0.67  0.56 -0.49]]
\|V\|_{rows}
 [[1. 1. 1.]]
\|V\|_{cols}
 [[1. 1. 1.]]
U \Sigma V^T
 [[0.5 2.  3. ]
 [4.  5.  6. ]
 [7.  8.  9. ]]
```

In [48]:

```
# Create a non-square 2 x 3 matrix A
A = np.array([[0.1, 0.2, 0.3],
              [0.4, 0.5, 0.6]])
# Take the SVD and print all the decomposed parts
print_svd(A)
```

```
A
 [[0.1 0.2 0.3]
 [0.4 0.5 0.6]]
U
 [[-0.39 -0.92]
 [-0.92  0.39]]
\|U\|_{rows}
 [[1. 1.]]
\|U\|_{cols}
 [[1. 1.]]
\Sigma
 [[0.95 0.   0.  ]
 [0.   0.08 0.  ]]
V
 [[-0.43  0.81  0.41]
 [-0.57  0.11 -0.82]
 [-0.7  -0.58  0.41]]
\|V\|_{rows}
 [[1. 1. 1.]]
\|V\|_{cols}
 [[1. 1. 1.]]
U \Sigma V^T
 [[0.1 0.2 0.3]
 [0.4 0.5 0.6]]
```

In [49]:

```
# Create a non-square 3 x 2 matrix A
A = np.array([[0.1, 0.2, 0.3],
              [0.4, 0.5, 0.6]]).T
# Take the SVD and print all the decomposed parts
print_svd(A)
```

```
A
 [[0.1 0.4]
 [0.2 0.5]
 [0.3 0.6]]
U
 [[-0.43  0.81  0.41]
 [-0.57  0.11 -0.82]
 [-0.7  -0.58  0.41]]
\|U\|_{rows}
 [[1. 1. 1.]]
\|U\|_{cols}
 [[1. 1. 1.]]
\Sigma
 [[0.95 0.  ]
 [0.   0.08]
 [0.   0.  ]]
V
 [[-0.39 -0.92]
 [-0.92  0.39]]
\|V\|_{rows}
 [[1. 1.]]
\|V\|_{cols}
 [[1. 1.]]
U \Sigma V^T
 [[0.1 0.4]
 [0.2 0.5]
 [0.3 0.6]]
```

Relation to eigendecomposition

The SVD is the same as:

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

In [50]:
```python
# Create a 3 x 2 matrix A
A = np.array([[0.1, 0.2, 0.3],
              [0.4, 0.5, 0.6]]).T

print_matrix('A', A)
print_matrix('AA^T', A @ A.T)
print_matrix('A^TA ', A.T @ A)

# Compute eigenvalues and eigenvectors of the matrices A A^T and A^T A
l_evals, l_evecs = np.linalg.eig(A @ A.T)
r_evals, r_evecs = np.linalg.eig(A.T @ A)

# compare eigenvector matrices to U and V from SVD
u, sigma, vt = np.linalg.svd(A)
print_matrix("\\text{eigenvectors of}\ AA^T", l_evecs)
print_matrix("U", u)

print_matrix("\\text{eigenvectors of}\ A^TA", np.fliplr(r_evecs)) # flip for comparison with 
print_matrix("V^T", vt)

# compare eigenvalues to Sigma
print_matrix("\sqrt{|\lambda|} \; \\text{of} \; AA^T", np.sqrt(np.abs(l_evals)))
print_matrix("\sqrt{|\lambda|} \; \\text{of} \; A^TA", np.sqrt(np.abs(r_evals))[::-1]) # flip
print_matrix("\Sigma", sigma)
```

```
A
 [[0.1 0.4]
 [0.2 0.5]
 [0.3 0.6]]
AA^T
 [[0.17 0.22 0.27]
 [0.22 0.29 0.36]
 [0.27 0.36 0.45]]
A^TA
 [[0.14 0.32]
 [0.32 0.77]]
\text{eigenvectors of}\ AA^T
 [[-0.43 -0.81  0.41]
 [-0.57 -0.11 -0.82]
 [-0.7   0.58  0.41]]
U
 [[-0.43  0.81  0.41]
 [-0.57  0.11 -0.82]
 [-0.7  -0.58  0.41]]
\text{eigenvectors of}\ A^TA
 [[-0.39 -0.92]
 [-0.92  0.39]]
V^T
 [[-0.39 -0.92]
 [-0.92  0.39]]
\sqrt{|\lambda|} \; \text{of} \; AA^T
 [[0.95 0.08 0.  ]]
\sqrt{|\lambda|} \; \text{of} \; A^TA
 [[0.95 0.08]]
\Sigma
 [[0.95 0.08]]
```

Very special case: symmetric, positive semi-definite matrix

For a symmetric, positive semi-definite matrix A,

- the eigenvectors are the columns of U or the columns of V.
- the eigenvalues are in Σ.

In [51]:
```python
# create a symmetric positive semi-definite 3 x 3 matrix
A = np.array([[2, 1, 0],
              [1, 3, 1],
              [0, 1, 4]])
print_matrix("A", A)

# Find eigenvectors and eigenvalues
evals, evecs = np.linalg.eig(A)

print("------------------")
print_matrix("\lambda (unordered)", evals)
print_matrix("\\bf{x} (unordered)", evecs)
print_matrix("\lambda", evals[::-1])       # Reverse order of eigenvalues to get them in size
print_matrix("\\bf{x}", np.fliplr(evecs))  # Flip eigenvector matrix left right to correspond

print("------------------")
# Take the SVD
u, sigma, vt = np.linalg.svd(A)
print_matrix("\Sigma", sigma)
print_matrix("U", u)
print_matrix("V", vt.T)
```

```
A
 [[2 1 0]
 [1 3 1]
 [0 1 4]]
------------------
\lambda (unordered)
 [[1.27 3.   4.73]]
\bf{x} (unordered)
 [[-0.79 -0.58  0.21]
 [ 0.58 -0.58  0.58]
 [-0.21  0.58  0.79]]
\lambda
 [[4.73 3.   1.27]]
\bf{x}
 [[ 0.21 -0.58 -0.79]
 [ 0.58 -0.58  0.58]
 [ 0.79  0.58 -0.21]]
------------------
\Sigma
 [[4.73 3.   1.27]]
U
 [[ 0.21 -0.58  0.79]
 [ 0.58 -0.58 -0.58]
 [ 0.79  0.58  0.21]]
V
 [[ 0.21 -0.58  0.79]
 [ 0.58 -0.58 -0.58]
 [ 0.79  0.58  0.21]]
```

SVD decomposes *any* matrix into three matrices with special forms

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.

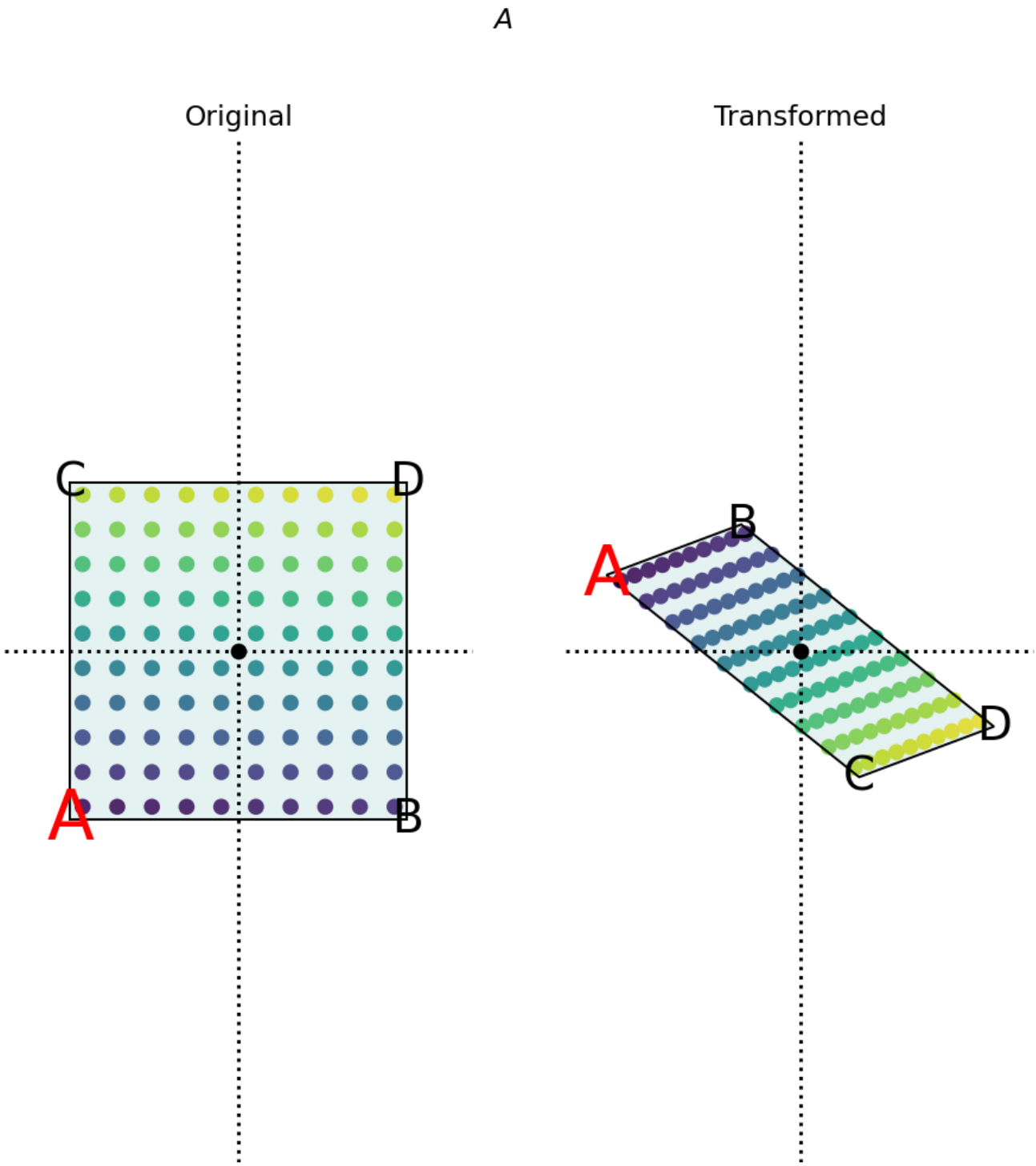
- U is orthogonal, so is a pure rotation matrix,
- Σ is diagonal, so is a pure scaling matrix,
- V is orthogonal, so is a pure rotation matrix.

Rotate, scale, rotate

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

In [52]:
```python
# Create a matrix A that performs a rotation and scaling transformation
A = np.array([[0.4, 0.75],
              [0.15, -0.6]])

show_matrix_effect(A, "A")
```

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



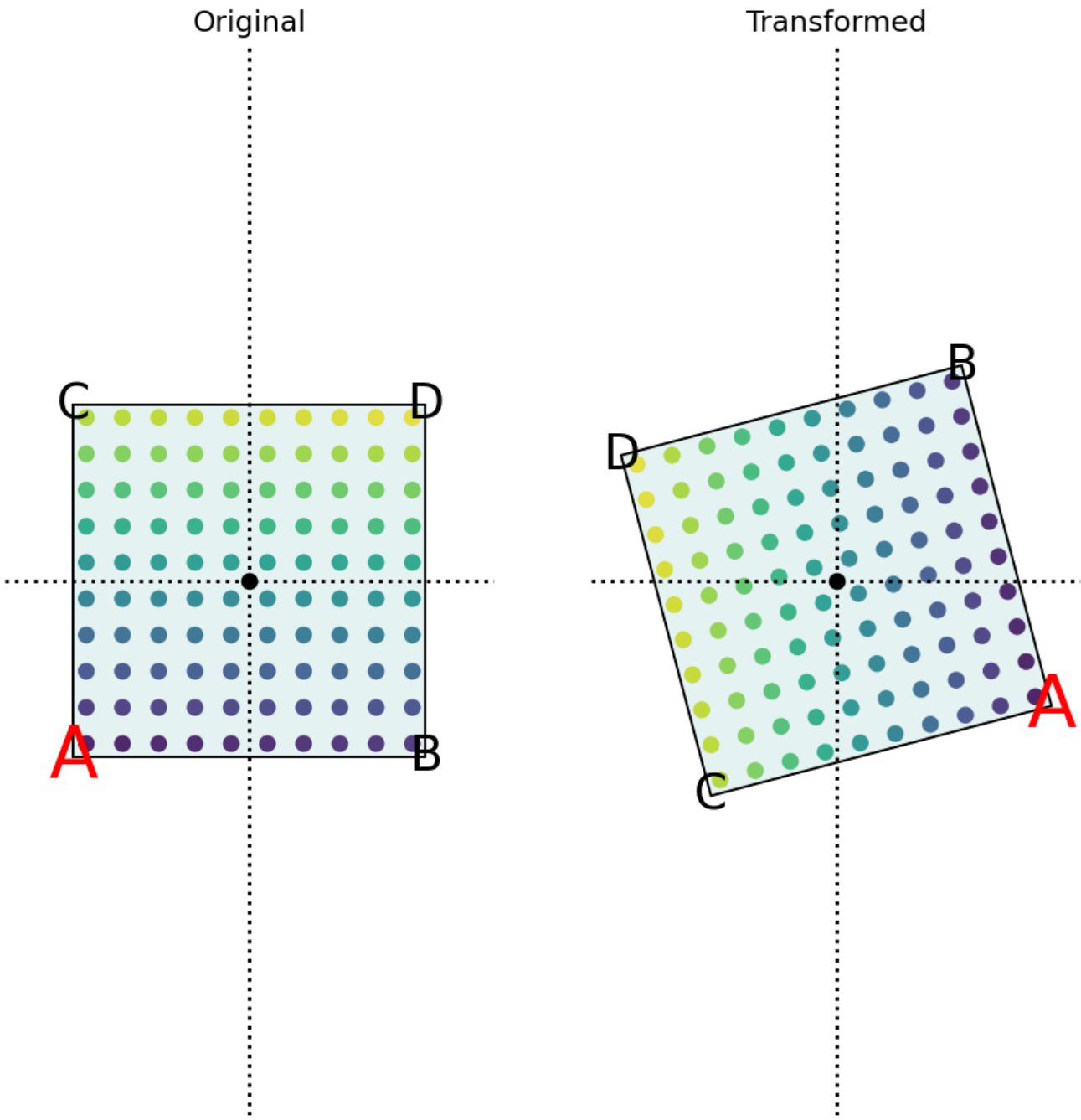
Let's break this transformation down into its consitutent parts:

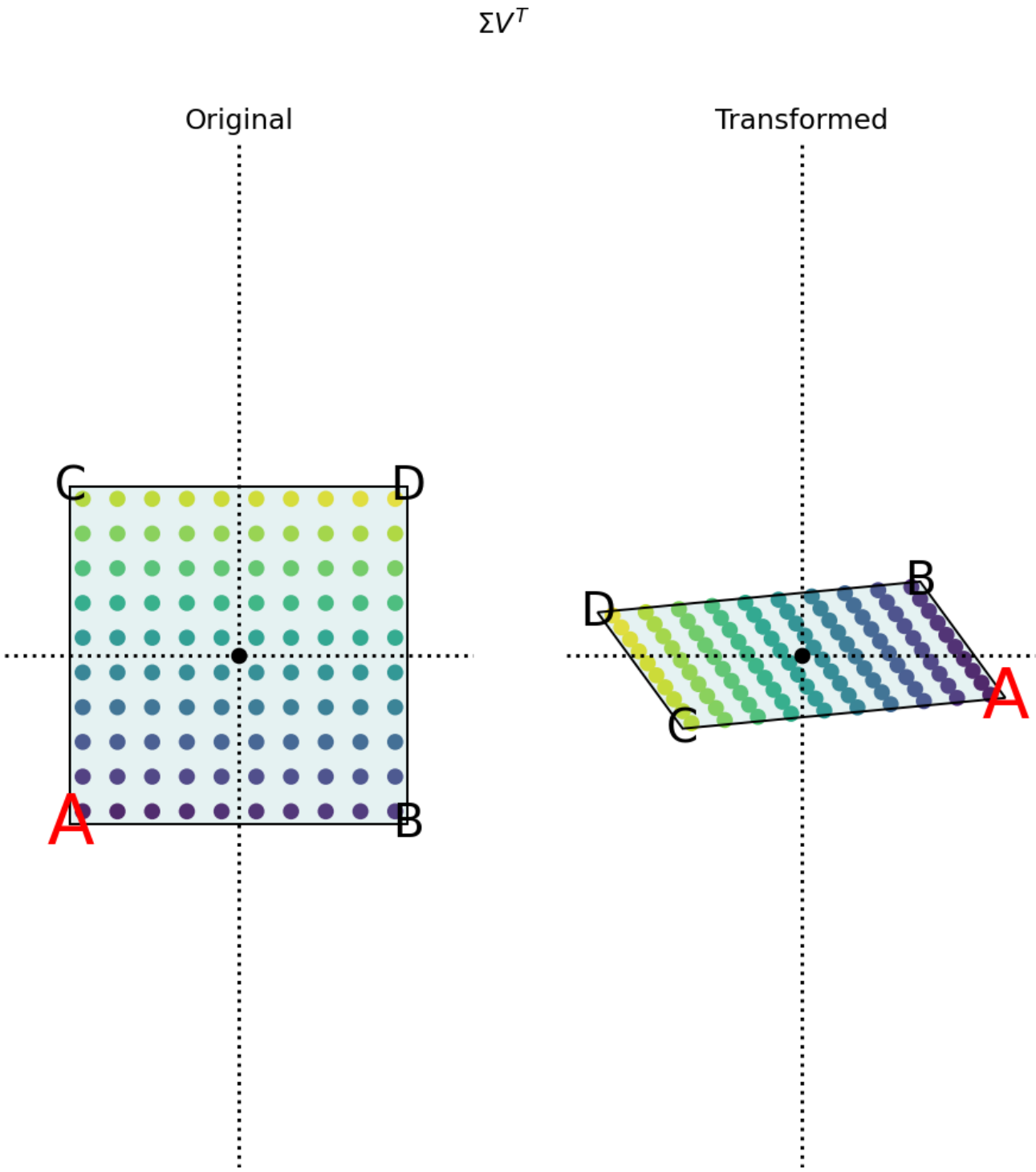
- V^T: rotation ,
- Σ: scaling,
- U: rotation.

In [53]:
```
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)))
```

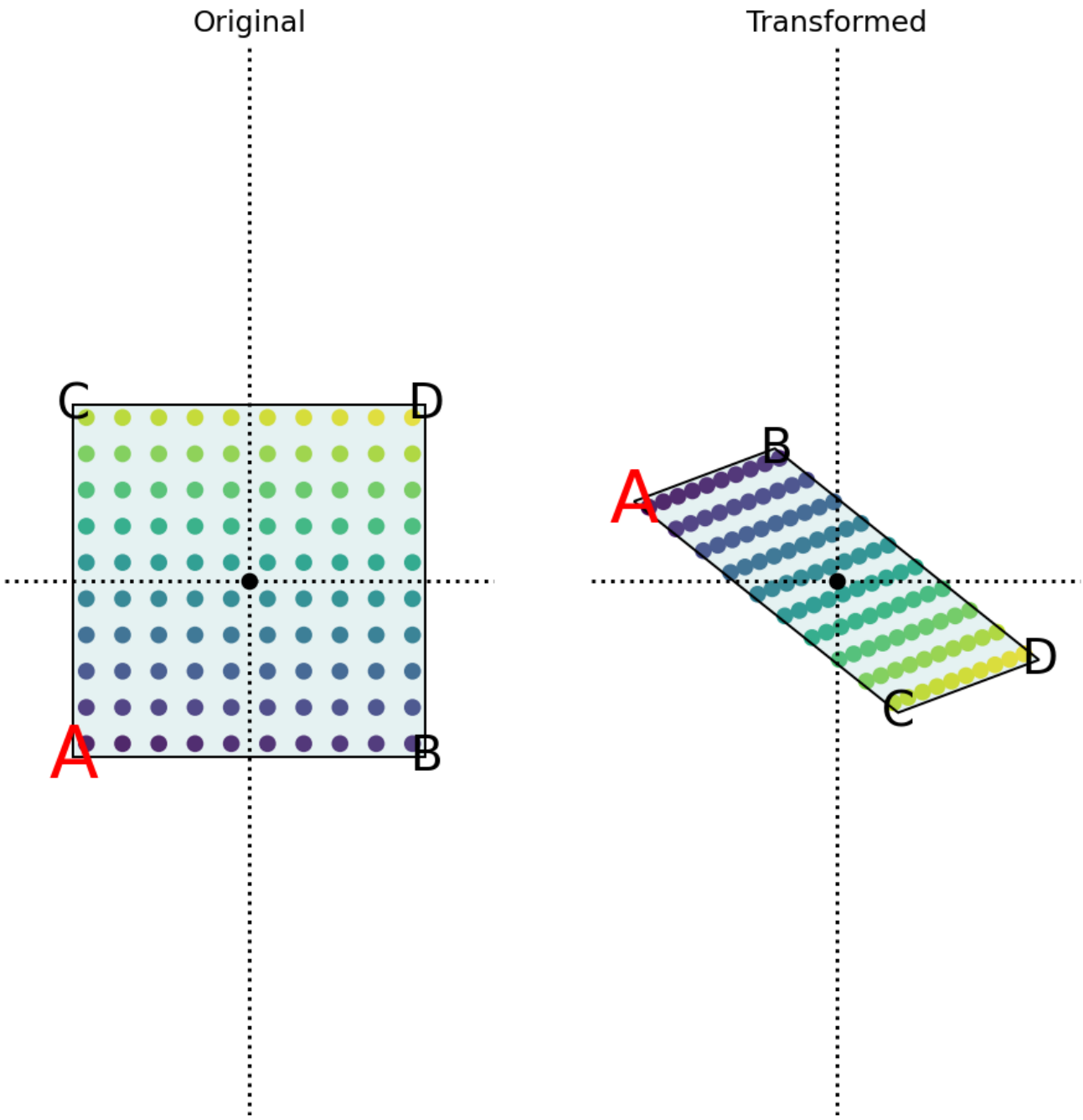
```
V^T
 [[-0.26 -0.97]
 [ 0.97 -0.26]]
\Sigma V^T
 [[-0.25 -0.96]
 [ 0.34 -0.09]]
U \Sigma V^T
 [[ 0.4   0.75]
 [ 0.15 -0.6 ]]
Determinant of vt: 1.0
Determinant of u: -1.0
Determinant of Sigma: 0.3525
```

V^T




ΣV^T
Original
Transformed
C
D
A
B
D
B
C
A

$U\Sigma V^T$



Using the SVD

Many matrix operations are trivial once we have the factorisation of the matrix into the three components. But some can only be performed on *on certain types of matrices*.

Fractional powers

We can use the SVD to compute interesting matrix functions like the square root of a matrix $A^{1/2}$. In fact, we can use the SVD to raise a matrix to any power, *in a single operation*, provided it is a **square symmetric matrix**.

We can use the SVD to:

- raise a matrix to a fractional power, e.g. $A^{1/2}$, which will "part do" an operation,

- invert a matrix: 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:

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

For a symmetric matrix, this is the same as:

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

Note: $A^{1/2}$ is not the elementwise square root of each element of A!

Rather, we must compute the elementwise square root of Σ, then compute $A^{1/2} = U\Sigma^{1/2}V^T$.

In [54]:
```
def powm(A, n): # generalised matrix power
    u, sigma, vt = np.linalg.svd(A)
    sigma_n = np.diag(sigma**n)
    return u @ sigma_n @ vt
    #return vt.T @ sigma_n @ u.T  # gives the same result as (u @ sigma_n @ vt) for a symmetr
```

In [55]:
```
# Create a square, symmetric matrix
A = np.array([[0.5, -1],
              [-1, 3]])

print_matrix("A", A)

# Compute A^(1/2)
a_half = powm(A, 0.5)
print_matrix("A^{1/2}", a_half)

# A^(1/2) A^(1/2) should be equal to A
print_matrix("A^{1/2} A^{1/2}", a_half @ a_half)

# Compute A^2
print_matrix("A^2", powm(np.array(A), 2))

# AA should be equal to A^2
print_matrix("AA", A @ A)
```

```
A
 [[ 0.5 -1. ]
 [-1.   3. ]]
A^{1/2}
 [[ 0.54 -0.45]
 [-0.45  1.67]]
A^{1/2} A^{1/2}
 [[ 0.5 -1. ]
 [-1.   3. ]]
A^2
 [[ 1.25 -3.5 ]
 [-3.5  10.  ]]
AA
 [[ 1.25 -3.5 ]
 [-3.5  10.  ]]
```

Inversion - relation to SVD

We can efficiently invert a matrix once it is in SVD form. For a non-symmetric matrix, we use:

$$A^{-1} = V\Sigma^{-1}U^T \quad [\spadesuit]$$

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

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 $O(n^3)$.

In [56]:
```
# Inversion using the SVD
def invert_by_svd(u, sigma, vt):
    sigma_inv = np.zeros((u.shape[0], vt.shape[0]))
    sqr = min(vt.shape[0], u.shape[0])
    sigma_inv[:sqr, :sqr] = np.diag(1.0/sigma)
    return vt.T @ sigma_inv @ u.T

# Create a non-symmetric, square matrix that we want to invert
A = np.array([[1,  2, 3],
              [4, -5, 6],
              [7, -8, 9.5]])

# Take the SVD
u, sigma, vt = np.linalg.svd(A)

print_matrix("A", A)

# Comput the inverse of A using two different methods, and compare
print_matrix("V \Sigma^{-1} U^T", invert_by_svd(u, sigma, vt))
print_matrix("A^{-1}", np.linalg.inv(A))
```

```
A
 [[ 1.   2.   3. ]
 [ 4.  -5.   6. ]
 [ 7.  -8.   9.5]]
V \Sigma^{-1} U^T
 [[ 0.03 -2.46  1.54]
 [ 0.23 -0.66  0.34]
 [ 0.17  1.26 -0.74]]
A^{-1}
 [[ 0.03 -2.46  1.54]
 [ 0.23 -0.66  0.34]
 [ 0.17  1.26 -0.74]]
```

Pseudo-inverse (non-symmetric)

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

This means we can solve (approximately) systems of equations where the number of input variables is different to the number of output variables. The **pseudo-inverse** or **Moore-Penrose pseudo-inverse** generalises inversion to (some) non-square matrices.

We can find approximate solutions for $\mathbf{x}$ in the equation:

$$A\mathbf{x} = \mathbf{y},$$

or in fact simultaneous equations of the type

$$AX = Y$$

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`.

In [57]:
```
# Create a non-square matrix, so that we can find the pseudo-inverse
A = np.array([[0.5, 1.0, 2.0],
              [1.0, 0.5, 0.0]])

print_matrix("A", A)
```

```
# Create a vector y
y = np.array([[2], [5]])

print_matrix("\\bf{y}", y)

# We want to know what x is, so we have to invert A using the pseudo-inverse
A_pinv = np.linalg.pinv(A) # pseudo-inverse is called pinv in NumPy

print_matrix("A^{+}", A_pinv)

# Compute x
x = A_pinv @ y

print_matrix("\\bf{x} = A^{+} \\bf{y}", x)

# check that Ax = y
print_matrix("A \\bf{x}", A @ x)
```

```
A
 [[0.5 1.  2. ]
 [1.  0.5 0. ]]
\bf{y}
 [[2]
 [5]]
A^{+}
 [[-0.07  0.85]
 [ 0.13  0.29]
 [ 0.45 -0.36]]
\bf{x} = A^{+} \bf{y}
 [[ 4.13]
 [ 1.73]
 [-0.9 ]]
A \bf{x}
 [[2.]
 [5.]]
```

Non-symmetric case: If the problem does not have an exact solution, the pseudo-inverse will give the closest result according to the L_2 norm. In other words, it will find the vector $\mathbf{x}$ that minimises the distance $\|A\mathbf{x} - \mathbf{y}\|_2$:

In [58]:
```
# Create a non-square matrix, so that we can find the pseudo-inverse
A = np.array([[0.5, 1.0, 2.0],
              [1.0, 0.5, 0.0]]).T

print_matrix("A", A)

# Create a vector y
y = np.array([[1], [2], [3]])

print_matrix("\\bf{y}", y)

# We want to know what x is, so we have to invert A using the pseudo-inverse
A_pinv = np.linalg.pinv(A)

print_matrix("A^{+}", A_pinv)

# Compute x
x = A_pinv @ y

print_matrix("\\bf{x} = A^{+} \\bf{y}", x)

# check that Ax = y
# won't be exact, as there are no exact solutions
print_matrix("A \\bf{x}", A @ x)
```

```
A
 [[0.5 1. ]
 [1.  0.5]
 [2.  0. ]]
\bf{y}
 [[1]
 [2]
 [3]]
A^{+}
 [[-0.07  0.13  0.45]
 [ 0.85  0.29 -0.36]]
\bf{x} = A^{+} \bf{y}
 [[1.55]
 [0.36]]
A \bf{x}
 [[1.13]
 [1.73]
 [3.1 ]]
```

- the $\|A\mathbf{x} - \mathbf{y}\|_2$ distance plays a massive role in data science and machine learning.

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**.

For example, a 4 x 4 matrix with rank 2 will take vectors in $\mathbb{R}^4$ and output vectors in a $\mathbb{R}^2$ subspace (a plane) of $\mathbb{R}^4$. The orientation of that plane will be given by the first two eigenvectors of the matrix.

Condition number of a matrix

The **condition number** number of a matrix is the ratio of the largest singular value to the smallest.

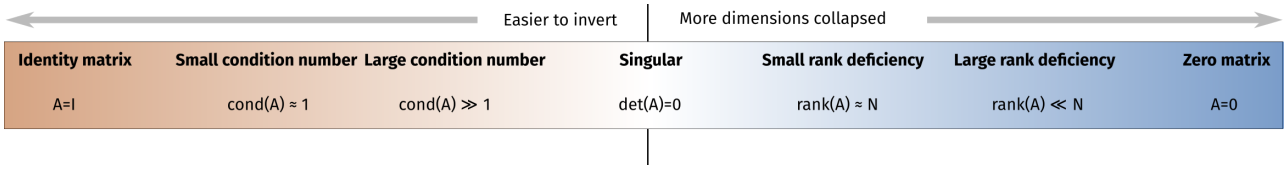
- This is only defined for full rank matrices.
- The condition number measures how sensitive inversion of the matrix is to small changes.
- A matrix with a small condition number is called **well-conditioned** and is unlikely to cause numerical issues.
- A matrix with a large condition number is **ill-conditioned**, 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.

Rank and **condition numbers** extend this concept.



- 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.

In [59]:
```
print("Well-conditioned matrix")
well_cond = np.random.normal(0, 1, (3, 3))
print("Condition number:", np.linalg.cond(well_cond))
print_matrix("A", well_cond)
```

```
Well-conditioned matrix
Condition number: 111.56152773799619
A
 [[ 0.19 -2.32  1.8 ]
 [-0.74  0.52  1.14]
 [-0.15  0.9  -0.44]]
```

In [60]:
```
# verify that condition number is ratio of largest to smallest singular values
u, sigma, vt = np.linalg.svd(well_cond)
print(sigma)
print("Ratio of max to min singular values:", np.max(sigma)/np.min(sigma))
```

```
[3.11055861 1.45358931 0.027882  ]
Ratio of max to min singular values: 111.56152773799617
```

In [61]:
```
# check that, for a well-conditioned matrix, if we apply the matrix twice,
# followed by the inverse twice, this results in the identity matrix I
print_matrix("A^{-1}A^{-1}AA", np.linalg.inv(well_cond) @ np.linalg.inv(well_cond) @ well_cond
```

```
A^{-1}A^{-1}AA
 [[ 1.  0.  0.]
 [-0.  1.  0.]
 [-0.  0.  1.]]
```

In [62]:
```
print("Ill-conditioned matrix")
ill_cond = np.random.normal(2000000, 0.002, (3, 3))
print("Condition number:", np.linalg.cond(ill_cond))
print_matrix("A", ill_cond)
```

```
Ill-conditioned matrix
Condition number: 3265884988.485249
A
 [[2000000. 2000000. 2000000.]
 [2000000. 2000000. 2000000.]
 [2000000. 2000000. 2000000.]]
```

In [63]:
```
u, sigma, vt = np.linalg.svd(ill_cond)
print(sigma)
print("Ratio of max to min singular values:", np.max(sigma)/np.min(sigma))
```

```
[6.00000000e+06 4.24468936e-03 1.83717431e-03]
Ratio of max to min singular values: 3265884988.485249
```

In [64]:
```
# for an ill-conditioned matrix, if we apply the matrix twice,
# followed by the inverse twice, we don't get the identity matrix I,
# due to floating point error accumulation
print_matrix("A^{-1}A^{-1}AA", np.linalg.inv(ill_cond) @ np.linalg.inv(ill_cond) @ ill_cond @
```

```
A^{-1}A^{-1}AA
 [[ 276.33  275.33  275.33]
 [ -17.33  -16.33  -17.33]
 [-110.68 -110.68 -109.68]]
```

In [65]:
```
# this matrix is rank 2 and projects 3D space onto a plane
A = np.array([[1,  2,  3],
```

```
              [2,  4,  6],
              [0, -1, -5]])

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

# one of the singular values is zero
print_matrix("\Sigma", sigma)
```

```
\Sigma
 [[9.59 2.02 0.  ]]
```

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. This will be covered in the lab.

Given a data set stored in matrix X, we can compute:

$$X^{\mathrm{w}} = (X - \mu)\Sigma^{-1/2} \quad [\spadesuit]$$

where μ is the **mean vector**, i.e. a row vector containing the mean of each column in X, and Σ is the **covariance matrix** (not the matrix of singular values).

This equation takes each column of X and subtracts the mean of that column from every element in the column, so that each column is centred on zero. Then it multiplies by the inverse square root of the covariance matrix, which is a bit like dividing each column of X by its standard deviation to normalise the spread of values in each column. However, it is more rigorous than that, because it also removes any correlations between the columns.

To summarise, 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**.

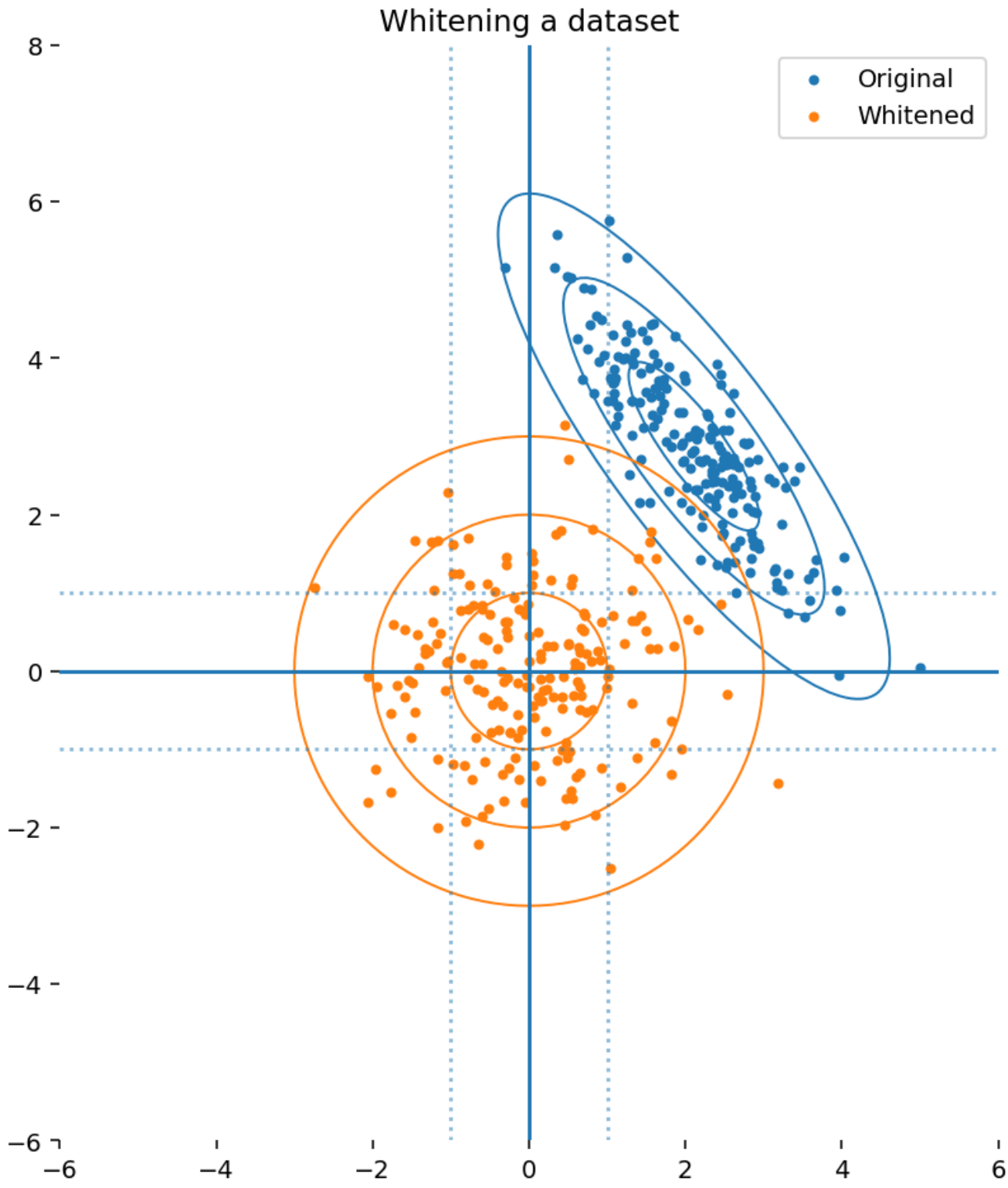
Removing the mean is easy. The difficult bit is computing the inverse square root of the covariance matrix. N.B. this is *definitely not* the inverse square root of the elements of the covariance matrix!

Fortunately, we can compute it easily by taking the SVD of the covariance matrix. We compute the inverse square root in one step, by taking the reciprocal of the square root of each of the singular values and then reconstructing.

The whitened version of the data set will have all linear correlations removed. This is an important preprocessing step when applying machine learning and statistical analysis algorithms.

In [66]:
```
from jhwutils import ellipse as ellipse

fig = plt.figure(figsize = (8, 8))
ax = fig.add_subplot(1, 1, 1)

# generate a random data set with some correlations
x = np.random.normal(0, 1, (200, 2)) @ np.array([[0.1, 0.5], [-0.9, 1.0]]) + np.array([2, 3])

# plot the data set with some ellipses to indicate the covariance
ax.scatter(x[:, 0], x[:, 1], c='C0', label="Original", s=10)
cov_ellipse(ax, x[:, 0:2], 1, facecolor='none', edgecolor='C0')
cov_ellipse(ax, x[:, 0:2], 2, facecolor='none', edgecolor='C0')
cov_ellipse(ax, x[:, 0:2], 3, facecolor='none', edgecolor='C0')
```

```
###################

# center around the mean
center_x = x - np.mean(x, axis=0)

# remove covariance via the SVD
u, sigma, vt = np.linalg.svd(np.cov(center_x, rowvar=False))
# here's the magic trick:
sigma_inv_sqr = vt.T @ np.diag(1.0 / np.sqrt(sigma)) @ u.T
white_x = center_x @ sigma_inv_sqr

###################

# plot the whitened data with some ellipses to indicate the covariance
ax.scatter(white_x[:, 0], white_x[:, 1], c='C1', label="Whitened", s=10)
cov_ellipse(
    ax, white_x[:, 0:2], 1, facecolor='none', edgecolor='C1')
cov_ellipse(
    ax, white_x[:, 0:2], 2, facecolor='none', edgecolor='C1')
cov_ellipse(
    ax, white_x[:, 0:2], 3, facecolor='none', edgecolor='C1')

# fix details
ax.set_xlim(-6, 6)
ax.set_ylim(-6, 8)

ax.axhline(1, ls=':', alpha=0.5)
ax.axvline(1, ls=':', alpha=0.5)
ax.axhline(-1, ls=':', alpha=0.5)
ax.axvline(-1, ls=':', alpha=0.5)

ax.axhline(0)
ax.axvline(0)
ax.set_aspect(1.0)
ax.legend()
ax.set_frame_on(False)
ax.set_title("Whitening a dataset");
```



Resources

- Matrix decompositions [**Recommended**]
- Eigenvectors and eigenvalues
- The Singular Value Decomposition
- A tutorial on principal components analysis
- A tutorial on the singular value decomposition ### Beyond the course
- A tutorial on spectral graph theory and graph Laplacians

- **Introduction to Linear Algebra** by Gilbert Strang. The standard reference text book for linear algebra.

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]: