lecture_7_matrices_ii_clean

April 14, 2021

1 Lecture 7: Computational Linear Algebra II

1.1 Data Science Fundamentals

1.2 ## Linear systems, inversion and matrix decompositions

DSF - University of Glasgow - Chris McCaig - 2020/2021

1.3 Summary

By the end of this unit you should know:

- the basic notation for matrices
- the view of matrices as linear maps
- how basic geometric transforms are implemented using matrices
- how matrix multiplication is defined and its algebraic properties
- the basic anatomy of matrices how discrete problems can be modelled using continuous mathematics, i.e. using matrices
  - how graphs can be represented as matrices

[ ]:
```
import IPython.display
IPython.display.HTML("""
<script>
  function code_toggle() {
    if (code_shown){
      $('div.input').hide('500');
      $('#toggleButton').val('Show Code')
    } else {
      $('div.input').show('500');
      $('#toggleButton').val('Hide Code')
    }
    code_shown = !code_shown
  }

  $( document ).ready(function(){
    code_shown=false;
    $('div.input').hide()
  });
</script>
```

```
<form action="javascript:code_toggle()"><input type="submit" id="toggleButton"␣
↪value="Show Code"></form>""")
```

[ ]:
```
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, 4.0), dpi=180)
```

2 Matrices and linear operators

2.1 Uses of matrices

We have seen that (real) vectors represent elements of a vector space as 1D arrays of real numbers (and implemented as ndarrays of floats).

Matrices represent **linear maps** as 2D arrays of reals; $\mathbb{R}^{m\times n}$.

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

The operations represented by matrices are a particular class of functions on vectors – "rigid" transformations. Matrices are a very compact way of writing down these operations.

2.1.1 Operations with matrices

There are many things we can do with matrices:

- They can be added and subtracted $C = A + B$
  - $(\mathbb{R}^{n\times m}, \mathbb{R}^{n\times m}) \to \mathbb{R}^{n\times m}$
- They can be scaled with a scalar $C = sA$
  - $(\mathbb{R}^{n\times m}, \mathbb{R}) \to \mathbb{R}^{n\times m}$
- They can be transposed $B = A^T$; this exchanges rows and columns
  - $\mathbb{R}^{n\times m} \to \mathbb{R}^{m\times n}$
- They can be *applied to vectors* $\vec{y} = A\vec{x}$; this **applies** a matrix to a vector.
  - $(\mathbb{R}^{n\times m}, \mathbb{R}^{m}) \to \mathbb{R}^{n}$
- They can be *multiplied together* $C = AB$; this **composes** the effect of two matrices
  - $(\mathbb{R}^{p\times q}, \mathbb{R}^{q\times r}) \to \mathbb{R}^{p\times r}$

2.2 Intro to matrix notation

We write matrices as a capital letter:

$$A \in \mathbb{R}^{n\times m} = \begin{bmatrix} a_{1,1} & a_{1,2} & \dots & a_{1,m} \\ a_{2,1} & a_{2,2} & \dots & a_{2,m} \\ \dots & & & \\ a_{n,1}+ & a_{n,2} & \dots & a_{n,m} \end{bmatrix}, \; a_{i,j} \in \mathbb{R}$$

(although we don't usually write matrices with capital letters in code – they follow the normal rules for variable naming like any other value)

A matrix with dimension $n \times m$ has n rows and m columns (remember this order – it is important!). Each element of the matrix A is written as $a_{i,j}$ for the ith row and jth column.

Matrices correspond to the 2D arrays / rank-2 tensors we are familiar with from earlier. But they have a very rich mathematical structure which makes them of key importance in computational methods. *Remember to distinguish 2D arrays from the mathematical concept of matrices. Matrices (in the linear algebra sense) are represented by 2D arrays, as real numbers are represented by floating-point numbers*

[ ]:
```
a = np.array([[1, 2, 3],
              [4, 5, 6],
              [7, 8, 9]])

print_matrix("A", a)
# note that code indexes from 0, whereas mathematical notation indexes from 1!

print_matrix("A_{1,3}", a[0,2]) # index row i=0, column j=2
```

2.2.1 Matrices as maps

We saw vectors as **points in space**. Matrices represent **linear maps** – these are functions applied to vectors which outputs vectors. In the standard notation, matrices are *applied* to vectors by multiplying them:

$$A\mathbf{x} = f(\vec{x})$$

This is equivalent to applying some function $f(\vec{x})$ to the vectors. Matrices represent functions mapping vectors to vectors in a very compact form, and they capture a special set of functions that preserve important properties of the vectors they act on. We'll see how matrix-vector multiplication is defined algorithmically shortly.

Effect of matrix transform Specifically, a $n \times m$ matrix A represents a function $f(\mathbf{x})$ taking m dimensional vectors to n dimensional vectors, ($\mathbb{R}^m \rightarrow \mathbb{R}^n$) such that all straight lines remain straight and all parallel lines remain parallel, and the origin does not move (i.e. that the zero vector $\vec{0^m}[0,0,\ldots,0] \rightarrow \vec{0^n}[0,0,\ldots,0]$).

Linearity This is equivalent to saying that:

$$f(\vec{x}+\vec{y}) = f(\vec{x}) + f(\vec{y}) \quad = A(\mathbf{x}+\mathbf{y}) = A\mathbf{x} + A\mathbf{y}, f(c\vec{x}) = cf(\vec{x}) \quad = A(c\mathbf{x}) = cA\mathbf{x},$$

i.e. the transform of the sum of two vectors is the same as the sum of the transform of two vectors, and the transform of a scalar multiple of a vector is the same as the scalar multiple of the transform of a vector. This property is **linearity**, and matrices represent **linear maps** or **linear functions**.

> Anything which is linear is easy. Anything which isn't linear is hard.

2.2.2 Geometric intuition (cube -> parallelepiped)

An intuitive way of understanding matrix operations is to consider a matrix to transform a cube of vector space centered on the origin in one space to a **parallelotope** in another space, with the origin staying fixed. This is the *only* kind of transform a matrix can apply.

A parallelotope is the generalisation of a parallelogram to any finite dimensional vector space, which has parallel faces but edges which might not be at 90 degrees.

Transforms and projections

- A linear map is any function $f\ R^m \to R^n$ which satisfies the linearity requirements.
- If the map represented by the matrix is $n \times n$ then it maps from a vector space onto the *same* vector space (e.g. from $\mathbb{R}^n \to \mathbb{R}^n$), and it is called a **linear transform**.
- If the map has the property $Ax = AAx$ or equivalently $f(x) = f(f(x))$ then the operation is called a **linear projection**; for example, projecting 3D points onto a plane; applying this transform to a set of vectors twice is the same as applying it once.

2.2.3 Keeping it real

We will only consider **real matrices** in this course, although the abstract definitions above apply to linear maps across any vector space (e.g complex numbers, finite fields, polynomials).

Linear maps are representable as matrices *Every linear map of real vectors can be written as a real matrix.* In other words, if there is a function $f(\vec{x})$ that satisfies the linearity conditions above, it can be expressed as a matrix A.

2.3 Examples

It's easiest to see the effect of matrix operations in low-dimensional vector spaces. Let's visualise some examples of linear transforms (linear maps $A \in \mathbb{R}^{2\times2}, \mathbb{R}^2 \to \mathbb{R}^2$), on the 2D plane.

We will take collections of vectors $\vec{x_1}, \vec{x_2}, \ldots$ and then apply various matrices to them. We forms the product $A\vec{x}$, which "applies" the matrix to the vector x.

[ ]:
```
show_matrix_effect(np.array(
        [[1,0],
         [0,1]]), suptitle="Identity")
```

[ ]:
```
# uniform scaling
show_matrix_effect(np.array(
        [[0.5, 0],
         [0,   0.5]]), suptitle="Uniform scale")
```

[ ]:
```
# non-uniform scaling
show_matrix_effect(np.array(
        [[0.5, 0],
         [0,   1.0]]), suptitle="Non-uniform scale")
```

[ ]:
```
# rotation by 90 degrees
show_matrix_effect(np.array(
        [[0, 1],
         [-1, 0]]), suptitle="Rotate 90")
```

[ ]:
```
# rotation by 30 degrees
# don't worry about how this matrix is constructed just yet
# but observe its effect
d30 = np.radians(30)
cs = np.cos(d30)
ss = np.sin(d30)

show_matrix_effect(np.array(
        [[cs, ss],
         [-ss, cs]]), suptitle="Rotate 30")
```

[ ]:
```
# rotation by 45 degrees, scale by 0.5
d30 = np.radians(45)
cs = np.cos(d30) * 0.5
ss = np.sin(d30) * 0.5

show_matrix_effect(np.array([[cs, ss],
                             [-ss, cs]]), "Rotate 45, Scale 0.5")
```

[ ]:
```
# flip x
show_matrix_effect(np.array([[-1, 0],
                             [0, 1]]), "Flip x")
```

[ ]:
```
# shear
show_matrix_effect(np.array([[0.15, 0.75],
                             [0.5, 0.8]]), "Shear")
```

[ ]:
```
# random!
show_matrix_effect(np.random.uniform(-1, 1, (2, 2)), "Random")
```

2.4 Matrix operations

There is an **algebra** of matrices; this is **linear algebra**. In particular, there is a concept of addition of matrices of *equal size*, which is simple elementwise addition:

$$A + B = \begin{bmatrix} a_{1,1} + b_{1,1} & a_{1,2} + b_{1,2} & \dots & a_{1,m} + b_{1,m} \\ a_{2,1} + b_{2,1} & a_{2,2} + b_{2,2} & \dots & a_{2,m} + b_{2,m} \\ \dots & & & \\ a_{n,1} + b_{n,1} & a_{n,2} + b_{n,2} & \dots & a_{n,m} + b_{n,m} \end{bmatrix}$$

along with scalar multiplication cA, which multiplies each element by c.

$$cA = \begin{bmatrix} ca_{1,1} & ca_{1,2} & \dots & ca_{1,m} \\ ca_{2,1} & ca_{2,2} & \dots & ca_{2,m} \\ \dots & & & \\ ca_{n,1} & ca_{n,2} & \dots & ca_{n,m} \end{bmatrix}$$

These correspond exactly to addition and scalar multiplication in NumPy.

[ ]:
```
a = np.arange(9).reshape(3, 3)
b = np.array([[1, 0, 1],
              [-1, -1, -1],
              [1, -1, 0]])

print_matrix("A", a)
print_matrix("B", b)
print_matrix("A+B", a + b)  # matrix addition
print_matrix("2A=A+A", a * 2)  # scalar multiplication
print_matrix("0.5A", a * 0.5)  # scalar multiplication
print_matrix("A-B = A+(-1)B", a - b)  # equal to (-1) * a + b
```

2.5 Application to vectors

We can apply a matrix to a vector. We write it as a product $A\vec{x}$, to mean the matrix A applied to the vector $\vec{x}$. This is equivalent to applying the function $f(\vec{x})$, where f is the corresponding function.

If A is $\mathbb{R}^{n\times m}$, and $\vec{x}$ is $\mathbb{R}^m$, then this will map from an m dimensional vector space to an n dimensional vector space.

All application of a matrix to a vector does is form a weighted sum of the elements of the vector. This is a linear combination (equivalent to a "weighted sum") of the components.

In particular, we take each element of $\vec{x}, x_1, x_2, \dots, x_m$, multiply it with the corresponding *column* of A, and sum these columns together.

- Set $\vec{y} = [0, 0, 0, \dots] = 0^n$ (the n-dimensional zero vector)
- For each column $1 \leq i \leq m$ in A
  - $\vec{y} = \vec{y} + x_i A_i$. Note that $x_i A_i$ is scalar times vector, and has n elements. A_i here means the ith column of A.

  1 2 1 3 4 2 5 6 7 8

  = 1 * [1,3,5,7] + 2 * [2,4,6,8] = [11+22, 13+25, 14+26, 17+28], = [5, 11, 17, 23]

  We can use @ to form products of vectors and matrices in Numpy:

[ ]:
```
A = np.array([[1, 2], [3, 4], [5, 6], [7, 8]])
x = np.array([1, 2])

print_matrix("A", A)
print_matrix("\\bf x", x)
print_matrix("A\\bf x", A @ x)
```

```
[ ]: # we'd never do this by hand
def apply_matrix_vector(A, x):
    y = np.zeros(A.shape[0])
    for i in range(A.shape[1]):
        y += x[i] * A[:, i]
    return y

print_matrix("A\\bf x", apply_matrix_vector(A, x))
```

```
[ ]: x = np.array([1, 2, 3])
A = np.array([[1, -1, 1], [1, 0, 0], [0, 1, 1]])  # 3x3
print_matrix("x", x)  # note: written horizontally, but interpreted vertically!
print_matrix("A", A)
print_matrix("Ax", A @ x)  # linear transform -- output dimension == input␣
 ↪dimension
```

```
[ ]: B = np.array([[2, -5, 5], [1, 0, 0]])  # 2x3 -- OK
print_matrix("B", B)
print_matrix("Bx", B @ x)
```

```
[ ]: ### shape error
C = np.array([[2, -5], [1, 0], [3, 3]])  # 3x2 -- not OK
print_matrix("C", C)
print_matrix("Cx", C @ x)
```

2.5.1 Matrix multiplication

Multiplication is the interesting matrix operation. Matrix multiplication defines the product $C = AB$, where A, B, C are all matrices.

Matrix multiplication is defined such that if A represents linear transform $f(\vec{x})$ and B represents linear transform $g(\vec{x})$, then $BA\vec{x} = g(f(\vec{x}))$.

Multiplying two matrices is equivalent to composing the linear functions they represent, and it results in a matrix which has that affect.

*Note that the composition of linear maps is read right to left. To apply the transformation A, **then** B, we form the product BA, and so on.*

2.5.2 Multiplication algorithm

This gives rise to many important uses of matrices: for example, the product of a scaling matrix and a rotation matrix is a scale-and-rotate matrix. It also places some requirements on the matrices which form a valid product. Multiplication is *only* defined for two matrices A, B if: * A is $p \times q$ and * B is $q \times r$.

This follows from the definition of multiplication: A represents a map $\mathbb{R}^q \to \mathbb{R}^p$ and B represents a map $\mathbb{R}^r \to \mathbb{R}^q$. The output of A must match the dimension of the input of B, or the operation is undefined.

Matrix multiplication is defined in a slightly surprising way, which is easiest to see in the form of an algorithm:

If $C = AB$ then

$$C_{ij} = \sum_k a_{ik} b_{kj}$$

The element at C_{ij} is the sum of the elementwise product of the ith row and the jth column, which will be the same size by the requirement above.

[ ]:
```
def matmul(a, b):
    p, q_a = a.shape
    q_b, r = b.shape
    # we can only multiply two matrices if A is p x q and B in q x r
    assert q_a == q_b
    # the result is a matrix of size p x r
    c = np.zeros((p, r))
    for i in range(p):
        for j in range(r):
            # Note that this can be seen as a simple *weighted sum*
            # the sum of the ith row of A weighted by the jth column of B
            c[i, j] = np.sum(a[i, :] * b[:, j])
    return c
```

[ ]:
```
a = np.array([[1, 2, -3]])
b = np.array([[1, -1, 1],
              [2, -2, 2],
              [3, -3, 3]])

print_matrix("{\\bf A}", a)
print_matrix("{\\bf B}", b)
c = matmul(a,b)
print_matrix("{\\bf A}B", c)
```

Matrix multiplication is of course built in to NumPy, and much more efficient than this algorthim. Matrix multiplication is applied by `np.dot(a,b)` or by the syntax `a @ b`

We'll use `a @ b` as the standard syntax for matrix multiplication.

[ ]:
```
# verify that this is the same as the built in matrix multiply
c_numpy = np.dot(a, b)
print_matrix("C_{\\text numpy}", c_numpy)
print(np.allclose(c, c_numpy))

c_at = a @ b
print_matrix("C_{\\text a @ b}", a @ b)
print(np.allclose(c_at, c))
```

2.5.3 Time complexity of multiplication

Matrix multiplication has, in the general case, of time complexity $O(pqr)$, or for multiplying two square matrices $O(n^3)$. This is apparent from the three nested loops above. However, there are many special forms of matrices for which this complexity can be reduced, such as diagonal, triangular, sparse and banded matrices. We will we see these **special forms** later.

There are some accelerated algorithms for general multiplication. The time complexity of all of them is $> O(N^2)$ but $< O(N^3)$. Most accelerated algorithms are impractical for all but the largest matrices because they have enormous constant overhead.

2.5.4 Apply matrices to vectors

The same algorithm for multiplying two matrices applies to multiplying a matrix by a vector **if we assume a m dimensional vector $\vec{x} \in \mathbb{R}^m$ is represented as a $m \times 1$ column vector**:

$$\begin{bmatrix} x_1 \\ x_2 \\ \cdots \\ x_m \end{bmatrix}$$

Then the product Ax is application of the linear map defined by A to vector $\vec{x}$. A must be of dimension $n \times m$ for this operation to be defined. If A is $m \times m$ then it is a **linear transform** (as we defined it above), and the result is another vector of the same dimension.

Note: this is a slight abuse of notation. $\vec{x}$ is a vector, not a matrix, and it is neither a column vector nor a row vector – it's just an element of a vector space. However, it's convenient to pretend it works like a $m \times 1$ matrix.

[ ]:
```
print_matrix("Ax", A @ x)
# force to a 2D array of m x 1 size
print_matrix("Ax", A @ x.reshape(-1, 1))
# note this will still be a 2D array in NumPy
# and will appear differently when printed. However, it
# has the same *semantics* as a 1D array
```

2.5.5 Transposition

The **transpose** of a matrix A is written A^T and has the same elements, but with the rows and columns exchanged. Many matrix algorithms use transpose in computations.

NumPy uses the `A.T` syntax to transpose any array by reversing its stride array, which corresponds to the mathematical transpose for matrices.

[ ]:
```
### Transpose
A = np.array([[2, -5], [1, 0], [3, 3]])
print_matrix("A", A)
print_matrix("A^T", A.T)
```

2.5.6 Column and row vectors

The transpose of a column vector

$$\vec{x} = \begin{bmatrix} x_1 \\ x_2 \\ \dots \\ x_n \end{bmatrix}$$

is a row vector

$$\vec{x}^T = \begin{bmatrix} x_1 & x_2 & \dots & x_n \end{bmatrix}$$

Note that from our definition of matrix multiplication, the product of a Mx1 with a 1xN vector is an $M \times N$ matrix. This is the **outer product** of two vectors, every possible combination of their elements:

$$\vec{x} \otimes \vec{y} = \vec{x}^T \vec{y}$$

and the product of a 1xN with an Nx1 vector is a 1x1 matrix; a scalar. This is exactly the **inner product** of two vectors:

$$\vec{x} \bullet \vec{y} = \vec{x}\vec{y^T},$$

and is only defined for vectors $\vec{x}, \vec{y}$ of the same length.

[again, we abuse notation to make it meaningful to "transpose" a vector; this is assuming we treat it as a column vector]

[ ]:
```
x = np.array([[1,2,3]])
y = np.array([[4,5,6]])

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

[ ]:
```
print_matrix("{\\bf x} \otimes {\\bf y}", np.outer(x,y))
print_matrix("{\\bf x}^T{\\bf y}", x.T @ y)

print_matrix("{\\bf x} \\bullet {\\bf y}", np.inner(x,y))
print_matrix("{\\bf x}{\\bf y}^T", x @ y.T)
```

2.6 Composed maps

There is a very important property of matrices. If A represents $f(x)$ and B represents $g(x)$, then the product BA represents $g(f(x))$. **Multiplication is composition.** Note carefully the order of operations. $BA\vec{x} = B(A\vec{x})$ means do A to $\vec{x}$, then do B to the result.

We can visually verify that composition of matrices by multiplication is the composition of their effects. For example, lets define a nonuniform scaling and a rotation matrix:

[ ]:
```
## Rotation
d30 = np.radians(30)
```

```
cs = np.cos(d30)
ss = np.sin(d30)
rot30 = np.array([[cs, ss], [-ss, cs]])

## Scaling
scale_x = np.array([[1,0], [0, 0.5]])
```

[ ]:
```
show_matrix_effect(rot30)
```

[ ]:
```
show_matrix_effect(scale_x)
```

[ ]:
```
A = scale_x @ rot30 # product of scale and rotate
print(A)
show_matrix_effect(A, "Rotate then scale")   # rotate, then scale
```

[ ]:
```
B = rot30 @ scale_x
print(B)
show_matrix_effect(B, "Scale then rotate")  # scale, then rotate
# note: Not the same!
```

2.6.1 Concatenation of transforms

Many software operations take advantage of the definition of matrix multiplication as the composition of linear maps. In a graphics processing pipeline, for example, all of the operations to position, scale and orient visible objects are represented as matrix transforms. Multiple operations can be combined into *one single matrix operation*.

The desktop UI environment you are uses linear transforms to represent the transformation from data coordinates to screen coordinates. Because multiplication composes transforms, only a single matrix for each object needs to be kept around. (actually for 3D graphics, at least two matrices are kept: one to map 3D -> 3D (the *modelview matrix*) and one to map 3D -> 2D (the *projection matrix*)).

Rotating an object by 90 degrees computes product of the current view matrix with a 90 degree rotation matrix, which is then stored in place of the previous view matrix. This means that all rendering just needs to apply to relevant matrix to the geometry data to get the pixel coordinates to perform the rendering.

2.6.2 Commutativity

The order of multiplication is important. Matrix multiplication does **not** commute; in general:

$$AB \neq BA$$

This should be obvious from the fact that multiplication is only defined for matrices of dimension $p \times q$ and $q \times r$; unless $p = q = r$ then the multiplication is not even *defined* if the operands are switched, since it would involve a $q \times r$ matrix by a $p \times q$ one!

Even if two matrices are compatible in dimension when permuted (i.e. if they are square matrices, so $p = q = r$), multiplication still does not generally commute and it matters which order operations are applied in.

Transpose order switching There is a very important identity which is used frequently in rearranging expressions to make computation feasible. That is:

$$(AB)^T = B^T A^T$$

Remember that matrix multiplication doesn't commute, so $AB \neq BA$ in general (though it can be true in some special cases), so this is the only way to algebraically reorder general matrix multiplication expressions (side note: inversion has the same effect, but only works on non-singular matrices). This lets us rearrange the order of matrix multiplies to "put matrices in the right place".

It is also true that

$$(A + B)^T = A^T + B^T$$

but this is less often useful.

2.6.3 Left-multiply and right-multiply

Because of the noncommutativity of multiplication of matrices, there are actually two different matrix multiplication operations: **left multiplication** and **right multiplication**.

B left-multiply A is AB; B right-multiply A is BA. This becomes important if we have to multiply out a longer expression:

$B\vec{x} + \vec{y}$left multiply by A $= A[B\vec{x} + \vec{y}] = AB\vec{x} + A\vec{y}$right multiply by A $= [B\vec{x} + \vec{y}]A = B\vec{x}A + \vec{y}A$

2.7 An example matrix for measuring spread: covariance matrices

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

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

This is a measure of how "spread out" a vector of values $\vec{x}$ 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 $\vec{x}$.

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:

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

As we will discuss shortly, this is a *special form* of matrix: it is square, symmetric and positive semi-definite.

[ ]:
```
x = np.random.normal(0,1,(500, 5))

mu = np.mean(x, axis=0)
sigma_cross = ((x - mu).T @ (x - mu)) / (x.shape[0]-1)
np.set_printoptions(suppress=True, precision=2)
print_matrix("\Sigma_{\\text{cross}}", sigma_cross)
```

It is also directly provided by NumPy as `np.cov(x)`.

[ ]:
```
# verify it is close to the function provided by NumPy
sigma_np = np.cov(x, rowvar=False)
print_matrix("\Sigma_{\\text{numpy}}",
             sigma_np)
```

2.7.1 Covariance ellipses

This matrix captures the spread of data, including any **correlations** between dimensions. It can be seen as capturing an **ellipse** that represents a dataset. The **mean vector** represents the centre of the ellipse, and the **covariance matrix** represent the shape of the ellipse. This ellipse is often called the **error ellipse** and is a very useful summary of high-dimensional data.

The covariance matrix represents a (inverse) transform of a unit sphere to an ellipse covering the data. Sphere->ellipse is equivalent to square->parallelotope and so can be precisely represented as a matrix transform.

[ ]:
```
#import importlib; importlib.reload(utils.ellipse)
from jhwutils import ellipse as ellipse
```

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

x =  np.random.normal(0,1,(200,2)) @ np.array([[0.1, 0.5], [-0.9, 1.0]])

ax.scatter(x[:,0], x[:,1], c='C0', label="Original", s=10)
ellipse.cov_ellipse(ax, x[:,0:2], 1, facecolor='none', edgecolor='k')
ellipse.cov_ellipse(ax, x[:,0:2], 2, facecolor='none', edgecolor='k')
ellipse.cov_ellipse(ax, x[:,0:2], 3, facecolor='none', edgecolor='k')

ax.set_xlim(-4,4)
ax.set_ylim(-4,4)
ax.axhline(0)
ax.axvline(0)
```

```
ax.set_frame_on(False)
ax.set_aspect(1.0)
ax.legend()
```

The mean vector and covariance matrix capture the idea of "centre" and "spread" of a collection of points in a vector space, the way the mean and the standard deviation do for real numbers.

3 Anatomy of a matrix

The **diagonal** entries of a matrix (A_{ii}) are important "landmarks" in the structure of a matrix. Matrix elements are often referred to as being "diagonal" or "off-diagonal" terms.

Matrices which are all zero except for a single diagonal entry are called… *diagonal* matrices. They represent a transformation that is an independent scaling of each dimension. Such matrices transform cubes to cuboids (i.e. all angles remain unchanged, and no rotation occurs).

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 3 \end{bmatrix}$$

`np.diag(x)` will return a diagonal matrix for a given vector `x`

[ ]:
```
print_matrix("\\text{diag}(x)", np.diag([1,2,3,3,2,1]))
```

The **anti-diagonal** is the set of elements $A_{i[N-i]}$ for an NxN matrix, for example a 3x3 binary anti-diagonal matrix:

$$\begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix}$$

[ ]:
```
# an anti-diagonal array is the diagonal of the flipped array
print_matrix("\\text{anti-diagonal}", np.fliplr(np.diag([1,2,3])))
```

3.1 Special matrix forms

There are *many* different special types of matrices, with different properties (for example, matrices that permute dimensions, or matrices that represent *invertible* transformations).

Wikipedia's enormous list of matrices gives a fairly comprehensive overview. We will only deal with a few special kinds of matrix in DF(H). In particular, we will deal with **real matrices** only, and primarily with **real square matrices**.

3.1.1 Identity

The identity matrix is denoted I and is a n square matrix, where all values are zero except 1 along the diagonal:

$$\begin{bmatrix} 1 & 0 & 0 & \dots & 0 \\ 0 & 1 & 0 & \dots & 0 \\ 0 & 0 & 1 & \dots & 0 \\ \dots & & & & \\ 0 & 0 & 0 & \dots & 1 \end{bmatrix}$$

The identity matrx *has no effect* when multiplied by another matrix or vector. (Obviously, it must be dimension compatible to be multiplied at all)

$IA = A = AI$ and $I\mathbf{x} = \mathbf{x}$.

It is generated by `np.eye(n)`

[ ]:
```
print_matrix("I", np.eye(3))
```

[ ]:
```
# your identity never changes anything
print_matrix("Ix", np.eye(3) @ np.array([4,5,6]) )
```

[ ]:
```
print_matrix("AI", np.array([[1,2,3],[4,5,6],[7,8,9]]) @
             np.eye(3))
print_matrix("IA", np.eye(3) @ np.array([[1,2,3],[4,5,6],[7,8,9]]) )
```

Any scalar multiple of the identity corresponds to a function which uniformly scales vectors:

$$(cI)\mathbf{x} = c\mathbf{x}$$

[ ]:
```
a = 0.5
x = np.array([4,5,6])
aI = np.eye(3) * a
print("c=",a)
print_matrix("cI", aI)
print_matrix("(cI){\\bf x}\n", aI @ x)

# the same thing:
print_matrix("c{\\bf x}\n",a*x)
```

3.1.2 Zero

The zero matrix is all zeros, and is defined for any matrix size $m \times n$. It is written as 0. Multiplying any vector or matrix by the zero matrix results in a result consisting of all zeros. The 0 matrix maps all vectors onto the zero vector (the origin).

[ ]:
```
z = np.zeros((4,4))
print(z)
```

```
[ ]: x = np.array([1,2,3,4])
     y = np.array([[1,-1,1], [1,1,-1], [1,1,1], [-1,-1,-1]])
     print_matrix("x", x)
     print_matrix("y", y)
     print_matrix("Ox", z @ x)
     print()
     print_matrix("Oy", z @ y)
```

3.2 Square

A matrix is square if it has size $n \times n$. Square matrices are important, because they apply transformations *within* a vector space; a mapping from n dimensional space to n dimensional space; a map from $\mathbb{R}^n \to \mathbb{R}^n$.

They represent functions mapping one domain to itself. Square matrices are the only ones that: * have an inverse * have determinants * have an eigendecomposition

which are all ideas we will see in the following unit.

3.3 Triangular

A square matrix is triangular if it has non-zero elements only above (**upper triangular**) or below the diagonal (**lower triangular**), *inclusive of the diagonal*.

Upper triangular

$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 0 & 5 & 6 & 7 \\ 0 & 0 & 8 & 9 \\ 0 & 0 & 0 & 10 \end{bmatrix}$$

Lower triangular

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 2 & 3 & 0 & 0 \\ 4 & 5 & 6 & 0 \\ 7 & 8 & 9 & 10 \end{bmatrix}$$

These represent particularly simple to solve sets of simultaneous equations. For example, the lower triangular matrix above can be seen as the system of equations:

$$x_1 = y_1 2x_1 + 3x_2 = y_2 4x_1 + 5x_2 + 6x_3 = y_3 7x_1 + 8x_2 + 9x_3 + 10x_4 = y_4$$

which, for a given y_1, y_2, y_3, y_4 is trivial to solve by substitution.

```
[ ]: # tri generates an all ones lower triangular matrix
     upper = np.tri(4)
     print_matrix("T_u", upper)

     # transpose changes a lower triangular to an upper triangular
     lower = np.tri(4).T
```

```
print_matrix("T_l", lower)
```

3.4 Resources

- **3blue1brown Linear Algebra series** (**strongly recommended**)
- **Introduction to applied linear algebra** *by S. Boyd and L. Vandenberghe*

3.5 Beyond this course

- **Linear Algebra Done Right** *by Sheldon Axler* (excellent introduction to the "pure math" side of linear algebra) ISBN-13: 978-0387982588
- **Coding the Matrix: Linear Algebra through Applications to Computer Science** *by Philip N Klein* (top quality textbook on how linear algebra is implemented, all in Python) ISBN-13: 978-0615880990
- **Linear Algebra and Learning from Data** *Gilbert Strang*, ISBN-13: 978-069219638-0, explains many detailed aspects of linear algebra and how they relate to data science.

3.6 Way beyond this course

- **The Matrix Cookbook** *by Kaare Brandt Petersen and Michael Syskind Pedersen*. If you need to do a tricky calculation with matrices, this book will probably tell you how to do it.

4 Graphs as matrices

4.1 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:

$$A \to BA \to CA \to DB \to DB \to EC \to BC \to DD \to FD \to GD \to HH \to A$$

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).

4.2 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).

[ ]:
```
# 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)))
```

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

[ ]:
```
# 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)
```

[ ]:
```
# any self transitions?
print(np.all(np.diag(adj)==0))
```

4.3 Resources

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