Unit 1 Arrarys:

Basic datatypes (examples):
Lists, strings, dictionaries, classes/structures and trees
Why use arrays:
images, sounds, videos - 2D, 1D, 3D arrays
scientific data - tables as numerical arrays
3D graphics - manipulating geometry eg. triangles
abstraction and elegance - extend operations done to one number to a whole array at once
mathematical power - linear algebra
efficiency - compact and computationally efficient
deep learning - eg. speech recognition
vectorization - one operation, many data (SIMD) GPUs
spreadsheet-like computation

Practical array manipulation
NumPy - Python package which provides a very comprehensive set of array types and operations
Shape and dtype
order - rows → columns → depth/planes
indexing and axes - 0 → n-1
dtype - data type of each element, commonly: float64 (double precision float), float32 (single precision float), int32, uint8
Images and sounds
image array's shape = image resolution
length of a sound
= shape of array ÷ sampling rate
Tabular data
many datasets can be thought of as arrays
spreadsheet-like arrangement:
row = observation, column = variable

Blank arrays
create new array filled with a value:
np.zeros(shape)
np.ones(shape)
np.full(shape, value)
create blank arrays with same shape and dtype as an existing array using _like variants:
y = np.zeros_like(x)
Random arrays
np.random.randint(a, b, shape) - uniform random integers between a and b-1
np.random.uniform(a, b, shape) - uniform random floats between a and b
np.random.normal(mean, std, shape) - normally distributed floats with given mean and std

Typing and shapes of arrays: vector, matrix, tensor
ndarrays can have different dimensions aka ranks - rank-3 tensor = 3D array
vector - 1D array, matrix - 2D array, tensor - nD array
typically no more than 6 dimensions - too big amounts of memory to store
Axes
axes - specific dimensions for a matrix - rows (axis 0) and columns (axis 1) axis indices start from 0 to n-1
Vectors and matrices
geometry of vectors:
have length and direction
can be added, subtracted and scaled
write a vector as a bold lowercase symbol: x
Signal arrays
1D arrays
sequence of measurements over time
include images, sounds and other time series
can be scaled, mixed, chopped up and rearranged, filtered etc
Algebra of matrices
matrix represents a linear map - a particular kind of function which operates on vectors and the operation function depends completely on the elements of the matrix
matrix written as an upper case symbol A
Array operations:
transformations
arithmetic - eg. multiplying every element by 2
indexing and slicing - eg. selecting the top half of the matrix
generation - zero arrays, identity arrays, meshgrids, random arrays
rearranging - reshaping, unraveling, sticking together, repeated, rotated and flipped
order operations - arrays can be sorted, shuffled and reindexed
aggregate functions - operations which work on groups of numbers (eg. min, max, mean)
Mathematical operations:
vector operations - dot product, cross product, norm, etc.
matrix operations - multiplication, transpose, inverse, matrix exponentials, decompositions
signal processing operations - convolution, Fourier transform, numerical gradients, cumulative summation

Statically typed, rectangular arrays - ndarrays
properties: fixed, predefined size and type (all elements have the same type), hold numbers (ints or floats), must be rectangular - no jagged edges
size cannot be changed after creation, extending one requires creating a new array with the right size and copying in (a portion of) the old elements
elements within the array can be changed after creation
elements are not Python values but "raw numbers"
ndarrays are a fairly thin wrapper around raw blocks of memory
operations can be performed extremely quickly

Creating arrays
Converting and copying
from another sequence type using np.array()
created blank or filled with a value
filled with random values
loaded from disk
Mutability and copying
if multiple variables refer to the same array, changing one affects the others
use .copy() to explicitly copy:
x = np.array([1,2,3])
z = x.copy()
Ragged arrays
array with rows of different lengths
turns into an object array (list of lists), not a true ndarray

Ranges
np.arange(end) - 0 to end-1
np.arange(start, end) - start to end-1
np.arange(start, end, step) - start to end-1 in steps (can be negative/fractional)
np.linspace(start, end, num) - generates evenly spaced values between start and end (inclusive)
Loading and saving arrays
text files:
np.loadtxt(fname)
np.savetxt(fname)
binary formats for storing arrays: .mat, .npz
specialised formats: HDF5 (scientific data), images, sounds, 3D geometry

Slicing and indexing arrays
arrays can have multidimensional indices - tuples of values
format: [start:end:step]
x[start] - specific index
x[start:end] - range (start and end optional)
x[start:end:step] - range with a step (all elements optional)
negative indices - count from the end
can specify an axis as an index (not a range), e.g. x[0, :]
Slicing vs. indexing
slicing - does not change the rank; selects a rectangular subset with the same number of dimensions
indexing - reduces the rank; selects a rectangular subset where one dimension is singular and removed
Sound manipulation examples
snd[4000:10000] - extracts a shorter segment (0:00 duration)
snd[::-1] - reverses the sound (0:05 duration)
snd[::2] - doubles playback speed (0:02 duration), but causes poor quality due to interpolation and aliasing

Boolean tests
can apply any logical test on arrays
results in a Boolean array with the same shape as the original array
Rearranging arrays
keeps the same elements but changes their arrangement, e.g. [1, 2, 3, 4, 5, 6] → [6, 5, 4, 3, 2, 1]
Transposition
exchanges rows and columns
not the same as a 90° rotation
No effect on 1D arrays
Reverse the order of all dimensions for arrays, with more than 2 dimensions, e.g
(10, 5, 60, 2) → .T → shape (2, 60, 5, 10)
Very fast operation O(1)

$$x = \begin{bmatrix} 1 & 2 & 3 \\ 0 & 0 & 0 \\ 4 & 5 & 6 \end{bmatrix} \qquad x^T = \begin{bmatrix} 1 & 0 & 4 \\ 2 & 0 & 5 \\ 3 & 0 & 6 \end{bmatrix}$$

$$y = \begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix} \qquad y^T = \begin{bmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{bmatrix}$$

Flip, rotate
flip left to right: np.fliplr(x)
flip up to down: np.flipud(x)
rotate 90° (equivalent to transpose + flipud): np.rot90(x)
symmetric arrays - have one half flipped left-right and concatenated to the original half

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

# Stack along a new axis 0
np.stack((a, b), axis=0)
# Output:
# array([[[1, 2],
#         [3, 4]],
#        [[5, 6],
#         [7, 8]]])  # Shape: (2, 2, 2)
```

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

# Concatenate along axis 0 (row-wise)
np.concatenate((a, b), axis=0)
# Output:
# array([[1, 2],
#        [3, 4],
#        [5, 6],
#        [7, 8]])  # Shape: (4, 2)

# Concatenate along axis 1 (column-wise)
np.concatenate((a, b), axis=1)
# Output:
# array([[1, 2, 5, 6],
#        [3, 4, 7, 8]])  # Shape: (2, 4)
```

boolean indexing - can index an array directly with boolean arrays (if they match the broadcasting rule)

```
x = [1,2,3]
bool = [True, False, True]
x[bool] = [1,3]
```

Selection and masking
can compare arrays (e.g. x > y) to select or mask data
where - selects a if condition is true, else b:
np.where(bool, a, b) — a and b must have the same shape or be broadcastable
nonzero - converts a boolean array into an array of indices:
np.nonzero(bool)
Fancy indexing
allows selection of irregular parts of an array and operations on them
Index arrays - arrays of integer indices that can be used directly as indices

$$x = [10 \quad 20 \quad 30 \quad 40]$$

$$y = [0 \quad 1 \quad 1 \quad 2 \quad 2 \quad 1]$$

$$x[y] = [10 \quad 20 \quad 20 \quad 30 \quad 30 \quad 20]$$

14. Concatenation and stacking
concatenate - joins arrays along an existing dimension
stack - stacks arrays along a new dimension
multidimensional concatenate - specify the axis to join on
2D stacking
vertically: np.vstack()
horizontally: np.hstack()
“depthwise” (stack along third axis): np.dstack()
Piecing together images
e.g. cut one image in half and place another image between the halves
Tiling
repeats arrays: np.tile(A, tiles) — A: array, tiles: shape
can be used to stretch sounds

Map: arithmetic on arrays
basic arithmetic is computed element-wise on arrays
Single argument — e.g. np.tan(x), -x
Two arguments — e.g. x + y, x - 1, np.maximum(x, y)
Other cases — e.g. np.where(bool, a, b)
Map — applying a function to each element of a sequence
Rules for mapping
single-argument operations apply to each element
multi-argument operations require compatible shapes
same shape → operation applies element-wise
not the same shape → broadcasting will tile one array if possible

f operands have the same number of dimensions, their shapes must match
Broadcasting rules
Equal size ☑
One is 1 ☑
Anything else ✗
pads shape from right to left — e.g. shape (8,) becomes (1, 1, 8)
Acceptable examples
(2,2) × (2,)
(2,3,4) × (3,4)
(2,3,4) × (4,)
Unacceptable examples
(2,3,4) × (2,4)
(2,3,4) × (2,)
(2,3,4) × (8,)
Transposing in broadcast
e.g. adding a vector to every column — transpose, broadcast, then transpose back

Reduction
applying a binary operator repeatedly to a sequence
e.g. 1+2+3+4 = 10, 1×2×3×4 = 24
Aggregate operations return a single summary value
np.any — logical OR (returns True if any non-zero)
np.all — logical AND (returns True if all non-zero)
np.min, np.max
np.sum, np.prod
np.mean, np.std
aggregate functions reduce over the last axis by default (can specify axis)

Accumulation
Cumulative sum — result of summing elements up to each point
e.g. [1,2,3,4] → [1,3,6,10]
np.cumsum(x, axis=0) — cumulative sum
np.cumprod(x, axis=0) — cumulative product
np.diff(x, axis=0) — differences between elements (1 less output)
np.gradient(x) — central differences; returns a list for each axis
Finding
find indices that satisfy a condition
np.argmax(x) — index of largest element
np.argmin(x) — index of smallest element
np.argsort(x) — indices that would sort the array
np.nonzero(x) — indices of non-zero elements (or True in Boolean arrays)

Unit 2: Numerical Issues and Tensors

Algebra: a loss of structure
algebraic properties of operations on real numbers are not preserved for computations
$ab \neq ba$
$a + b \neq b + a$
$a(b + c) \neq ab + ac$
$a(bc) \neq (ab)c$

Number types
Integers — whole numbers, usually stored as binary (2's complement or unsigned)
overflow — exceeding bounds may lead to undefined behaviour (usually wraparound)
Floating point — numbers with fractional parts
compact way to represent a wide range of values
represented as a number between 1.0 and 2.0 with a scaling factor (powers of 2)
precision is variable — very precise near 0, less precise further away

Binary representations of floats
Value = (−1)^sign × 2^(exponent − bias) × 1.mantissa

64 bit bias is 1023, 32 bit bias is 127. (2^(number of exponent bits) - 1)

Integers in floats
For float64 every integer from -2^{53} to 2^{53} is precisely representable, otherwise not so precise.

1.0000000000000000000 / 1.00e+00

Raw	0 | 01111111111 | 0000000000000000000000000000000000000000000000000000		
Hex	00 00 00 00 00 00 F0 3F		
	Sign	Exp	Mantissa
Binary	0	01111111111	0000000000000000000000000000000000000000000000000000
Integer	1	0	0
Power	1	2^0	1.0
=	1	1.00e+00	1.0
Float	1.0000000000000000000		1.00e+00

Sign, exponent, mantissa
floating point represented by three parts
binary floating point uses base 2
Sign — 1 bit (positive or negative)
Exponent — 11 bits (signed int for how much to shift the mantissa)
Mantissa — 52 bits (unsigned int for fractional part)
representation:
sign × (1.[mantissa]) × (2^exponent)
the leading one — normalised so numbers are between 1.0 and 2.0
e.g. $0011010110101101110000111_2 \rightarrow 1.1011010111011100001111_2$

IEEE 754
specifies both a representation for floating point numbers and operations defined upon them along with a set of conventions for "special" characters

Floats and doubles
floats — single precision: float32
doubles — double precision: float64
GPUs are generally faster using float32
most desktop CPUs have specialised float64 hardware

Exotic floating point numbers
float16 — very fast operations but less precision
float128 / float256 — very rare, mainly used in astronomical simulations

Type	Min	Max
int8	-128	127
uint8	0	255
int16	-32,768	32,767
uint16	0	65,535
int32	-2,147,483,648	2,147,483,647
uint32	0	4,294,967,295
int64	$-9{,}223 \times 10^{18}$	$9{,}223 \times 10^{18}$
uint64	0	$1{,}844 \times 10^{19}$

Name	Common name	Base	Digits	Decimal digits	Exponent bits	Decimal E max	Exponent bias	E min	E max
binary16	Half precision	2	11	3.31	5	4.51	$2^4-1 = 15$	−14	+15
binary32	Single precision	2	24	7.22	8	38.23	$2^7-1 = 127$	−126	+127
binary64	Double precision	2	53	15.95	11	307.95	$2^{10}-1 = 1023$	−1022	+1023
binary128	Quadruple precision	2	113	34.02	15	4931.77	$2^{14}-1 = 16383$	−16382	+16383
binary256	Octuple precision	2	237	71.34	19	78913.2	$2^{18}-1 = 262143$	−262142	+262143

Special features of floats
Float exceptions occur at the hardware level.
Invalid operation – without a defined real number result, e.g. 0.0/0.0, sqrt(-1.0)
Division by zero
Overflow – floating point number exceeds float64 limits
Underflow – floating point number is below float64 limits
Inexact – occurs if a computation will produce an inexact result due to rounding
Each exception can be trapped or untrapped:
Trapped – indicates the operation is problematic; it can halt or take another action, usually for invalid operations
Untrapped – does not halt execution and instead does a default operation, usually for inexact and underflow
Can be either – overflow, division by zero

Special numbers
Zero: +0.0 and -0.0
These work exactly the same in all operations, except in sign-propagating cases:
0.0 == -0.0 → True, -0.0 < 0.0 → False
2 × -0.0 = -0.0, 2 × 0.0 = 0.0
Infinity: +∞ and -∞
Explicitly encode infinities
All ones for exponent
All zeros for mantissa
Examples:
np.inf × 0 = NaN
np.inf / np.inf = NaN
np.inf × -1 = -np.inf
NaN: Not a Number
Represents invalid values, e.g.:
0 / 0
∞ / ∞
∞ - ∞ or -∞ + ∞
∞ × 0 or 0 × -∞
sqrt(x) when x < 0
log(x) when x < 0
NaN propagates – any operation involving NaN outputs NaN
Any comparison to NaN evaluates to False, even NaN == NaN
Can be used in arrays as placeholders for missing data
np.isnan(a) tests if an array has a NaN
All ones exponent but non-zero mantissa

Roundoff and precision

Roundoff error is introduced when operations must quantise the results of computations, e.g., when there's a big difference in magnitude of numbers:

$1 \times 10^9 + 1 \times 10^{-16} = 1 \times 10^9$

Financial operations – never use floating point for financial calculations. Use decimal formats with precisely defined roundoff rules

Repeated tiny offsets to enormous values can have no effect – ordering of operations can be important:

$(1 \times 10^{30} + 1) - 1 \times 10^{30} = 0$

$(1 \times 10^{30} - 1 \times 10^{30}) + 1 = 1$

Some basic rules:

Magnitude error – x + y will have large error if x and y have very different magnitudes

Division error – x / y will have large error if y is very small

Cancellation error – x - y will have large error if $x \approx y$

All close

When comparing floating point numbers, do not test for precise equality

Instead, check if the absolute difference is less than a threshold: $|x - y| < \varepsilon$

To test if all elements of two arrays are close: np.allclose(x, y)

Relative error

the absolute difference between a floating point number and its real counterpart, normalised by the magnitude of the real number.

IEEE 754 guarantees that this error is always less than

$$\epsilon = \frac{|\mathrm{float}(x) - x|}{|x|},$$

machine precision float compared to its real counterpart

In IEEE754 where t-bits for mantissa

$$\epsilon \leq \frac{1}{2}2^{-t}, \epsilon \leq 2^{-t-1},$$

Types of operations by rank effect

Rank-preserving – operations like maps (e.g., x + 1), slices (e.g., x[:2, :2]), and transpose. These preserve the number of dimensions.

Rank-reducing – operations like indexing (e.g., x[0]), reductions (e.g., np.sum(x)), and ravel(). These reduce the number of dimensions.

Rank-promoting – operations that add new dimensions to an array. These include broadcasting (which implicitly promotes tensors when shapes can be repeated to match) and np.tile (which can repeat along any axis).

Advanced reshaping: squeezing and adding dimensions

Multiplying a row vector by a column vector results in a matrix.

To make a vector into a column vector, promote it to a matrix by adding a singleton dimension, e.g., (3,) → (3,1)

How arrays are laid out in memory

Efficiency is achieved by packing the numbers in one long float sequence, one after another, even if the array is more than one-dimensional. You can see the flat sequence using np.ravel() – this returns the array as a 1D vector.

Strides and shape:

Striding is used to implement multidimensional indexing.

There is one stride per dimension – each tells the system how many bytes forward to move to find the next element for each dimension.

Dope vector – striding information is held separately from the data.

Iliffe vector – uses nested pointers to refer to multidimensional arrays. Can store ragged arrays trivially but is much less efficient than dope vectors.

Transpose – to transpose an array, you only need to exchange the shape elements and the strides in the header. The elements themselves are not touched; operations simply read the memory in a different way after the transpose.

Rigid transformations like flipping and transposing only change the strides used to compute indexing operations:

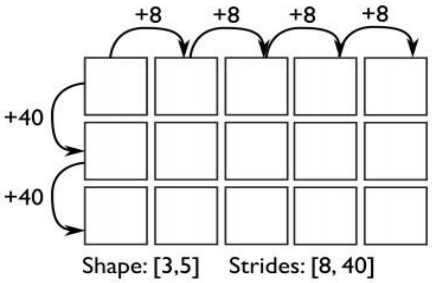
Original: shape (6, 5), strides (40, 8)

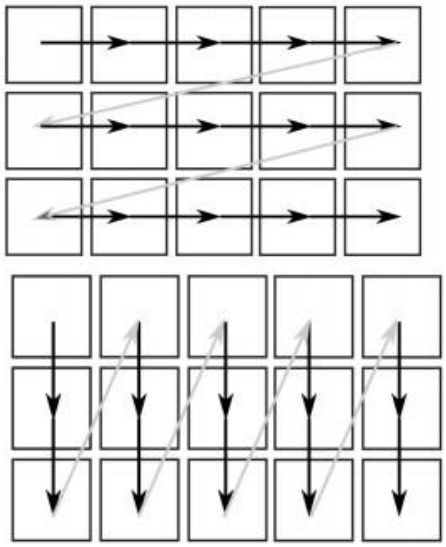
Transposed: shape (5, 6), strides (8, 40)

Flipped up/down: shape (6, 5), strides (-40, 8)

Flipped left/right: shape (6, 5), strides (40, -8)

Rotated 90°: shape (5, 6), strides (-8, 40)





C and Fortran order

NumPy default is that the last index changes fastest. C ordering (also known as row-major) is the NumPy default. See image

Fortran order (also known as column-major).

Tensor operations

1D and 2D matrices are easy to work with.

Being able to think about vectorised operations and then mold a tensor to the right shape is very important.

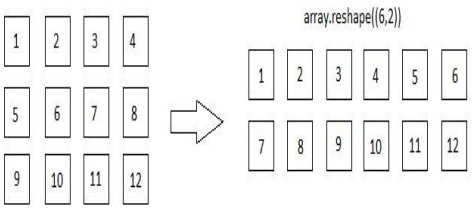
Reshape

You can change the shape of an array completely, but the size must remain the same. Rules:

Same size

Same order of elements

The last dimension changes fastest, the second last changes second fastest, etc.



Adding dimensions:

np.reshape(y, (3,1))

Index an array with an additional dimension to promote it to a higher rank:

x[:] → x[None, :] → shape (1, 3)

x[:] → x[:, np.newaxis] → shape (3, 1)

x[:] → x[None, None, :, None] → shape (1, 1, 3, 1)

```
2]: # einsum is really useful for higher order arrays
    y = np.zeros((5, 4, 10, 10))
    y_rearranged = np.einsum("ijkl -> jlik", y)
    print("Y shape", y.shape)
    print("Y strides", y.strides)
    print()
    print("Y rearranged", y_rearranged.shape)
    print("Y rearranged strides", y_rearranged.strides)

    Y shape (5, 4, 10, 10)
    Y strides (3200, 800, 80, 8)

    Y rearranged (4, 10, 5, 10)
    Y rearranged strides (800, 8, 3200, 80)
```

Squeezing

Simply index to remove dimensions:

x[0, 0, :, 0] → shape (3,)

np.squeeze(x) removes all singleton dimensions.

Elided arrays

When dimensions are repeated, indexing can use ellipsis (...) to simplify:

x[0, :, :, :, 4] → x[0, ..., 4]

Swapping and rearranging axes

You can swap axes to, for example, perform an operation on each column and then swap back: x.swapaxes(2, 3)

You can bypass the "last dimension pours first" rule by doing the following:

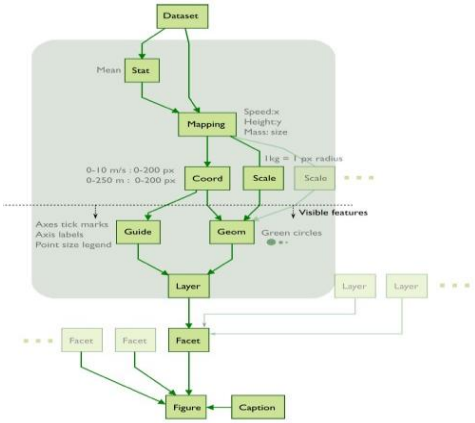
- Rearranging axes
- Reshaping the array
- (Optional) Rearranging axes again

Einstein summation notation

Allows you to swap axes easily, e.g., np.einsum('ijkl -> jlik', x)

Unit 3: Scientific Visualisation

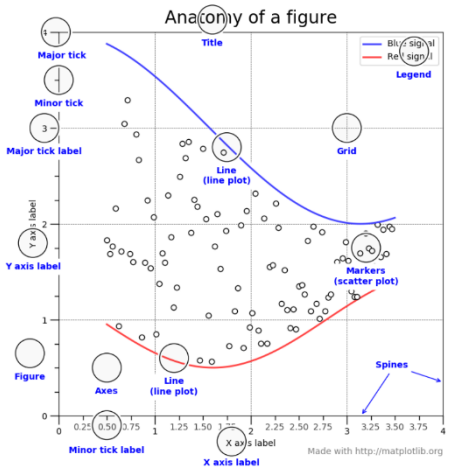
What is visualisation?
Visualisation is the representation of data for human visual perception.
Purposes include:
To build intuition about relationships and structure in data.
To summarise large quantities of data.
To help decision-makers make quick judgments on specific questions.
Scientific visualisation refers specifically to numerical data, where the focus is on precision, clarity, and accuracy in the visualisation.
Basic categories include:
Plots of single 1D arrays, e.g. histogram.
Plots of pairs of 1D arrays (x, y), e.g. scatterplot.
Plots of single 2D arrays, e.g. contour plots.
Grammar of graphics
The creation of scientific visualisations can be specified in terms of a graphical language – how to turn data into images and how the reader should interpret them.
Layered Grammar of Graphics was developed by Hadley Wickham.



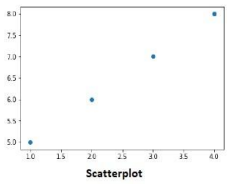
Plotting data well
Units – if an axis represents real-world units (e.g. mm, s), use units appropriate to the scale. Only truly dimensionless values (bare numbers) should have no visible units.
Avoid rescaled units like 1e8 or 10^8.
Axis – you can change the axis limits to suit your needs.
Faceted – if needed, split a graph into multiple graphs to reveal different aspects of the data.
One dataset, many views
Visualisation must communicate meaning to a reader.
Visualisation is not defined solely by the dataset.
Some visualisation types are better suited for certain kinds of datasets, but the choice depends on the purpose of the visualisation.
It is often helpful to think about the caption you would want to write before creating the visualisation.

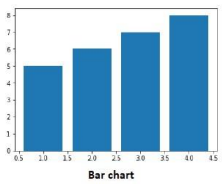
Stat
Statistic computed from data, e.g. mean.
Mapping
Transformation of data attributes to visual values.
Scale
Specifies the transformation of the units.
Coord
Coordinate system connecting mapped data onto points on the plane.
Guide
Visual reference indicating the meaning of a mapping, including the scale and the attribute being mapped. Includes tick marks, labels, colour scales, and legends.
Geom
Geometric representation of data after it has been mapped.
Layer
One set of geoms with one mapping on one coordinate system makes up one layer. Multiple layers can be overlaid on a single coordinate system.
Facet
Different view on the same dataset on a separate coordinate system.
Figure
A set of one or more facets.
Caption
Explains the visualisation to the reader.
Anatomy of a figure
Figure and caption
Many figures are single graphs (facets) but some may have multiple facets. Every figure needs to have a clear caption which explains to the reader what they should see from the graph.
Visual representation of plots: guides, geoms and coords
Guides
Axes – labelled with any applicable units, tick marks show subdivisions.
Legend – explains what markers and lines mean, guides to identify different layers in the plot.
Title – explains what the plot is.
The plot may also have a grid and annotations to point out relevant features.
Geoms
Lines or curves – for continuous functions.
Markers – for discrete points.
Patches – for shapes with areas.

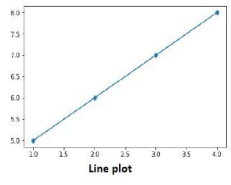
Stats
Common examples of stats include aggregate summary statistics like central tendency (mean, median) and deviation (standard deviation, max/min, interquartile range).
Binning operations categorise data into a number of discrete bins and count the data points falling into these bins.
Smoothing and regression involve finding approximating functions to datasets, such as linear regression.

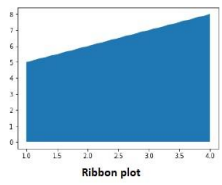


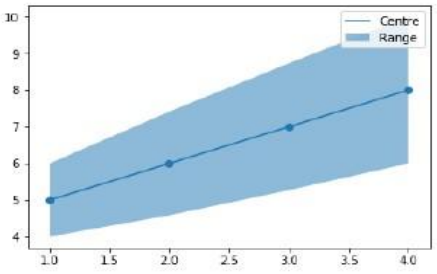
Matplotlib
A sophisticated plotting package that integrates with NumPy and the scientific Python stack.
It is the standard for producing publication-quality figures.
Basic 2D plots
Dependent and independent variables: y = f(x)
x is the independent variable, which "causes" the relationship.
y is the dependent variable, the "effect" of the relationship.
Common 2D plot types
Ribbon plots are usually combined with layering geoms.
Line geoms show the trend of data.
Point geoms notate the actual measurements
area geoms - represent the measurement uncertainty







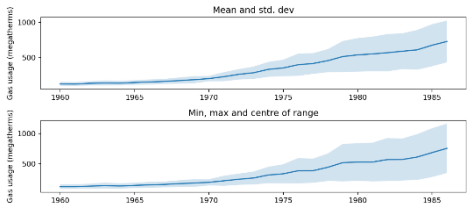


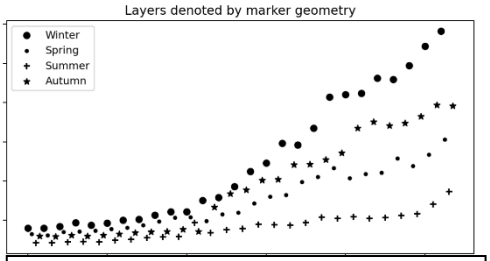


Binning operations
Divides data into a number of discrete groups (bins) which help summarise data, especially for continuous variables.
A histogram is a combination of a binning operation with a standard 2D bar chart.
Too many bins can result in a sparse and "gappy" output.
Too few bins can overly smooth the data, losing detail.
Bins placed poorly may fail to capture meaningful parts of the data.
Binning can work in any dimension.

Ranking operations
An alternative view of a 1D vector is a sorted chart or rank plot, which shows the value against its rank in the array.
This idea is sometimes generalised into a Q-Q plot (quantile-quantile plot), where the ranks are replaced with ranked values from a known statistical distribution.

Aggregate summaries
A common statistical transform is to represent data ranges using the minimum and maximum, standard deviation, or interquartile range.



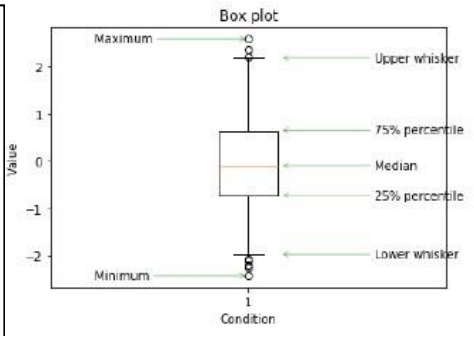


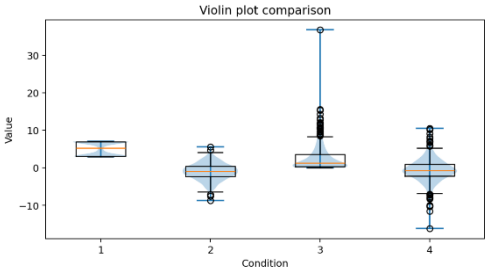
Lines
Lines are geoms that connect points together.
They should be used when it makes sense to ask what lies between two data points.
Line styles can include variable thickness, colour, and dash patterns.
If we know that the value cannot have changed between measurements (for example, in a coin toss), a staircase plot should be used.
A staircase connects points but keeps the value fixed until a new data point is observed.
If measurements are naturally discrete, a bar chart may be more suitable.

Alpha and transparency
Geoms can be rendered with varying transparency, controlled by the alpha value (opacity is the inverse of transparency).
Transparency is useful when many geoms overlap or when emphasising certain geoms.
Use transparency carefully — too much can make a graph hard to read.

Box plot
Computes multiple statistics of a dataset and shows them in a single geom which has multiple components.
Interquartile range (75%–25% percentiles of the dataset) is shown as a box.
The median value is shown as a horizontal line inside the box.
Extrema (often 2.5% and 98.5%) are drawn as whiskers – lines with a short crossbar at the end.
Outliers are usually marked as crosses or circles.

Violin plot
Extends the box plot to show the distribution of data more precisely using a smoothing technique called a kernel density estimate.



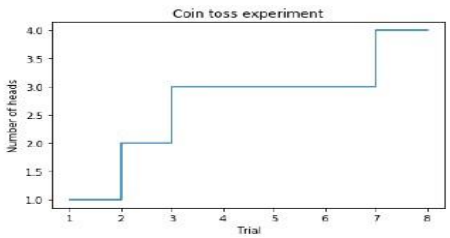


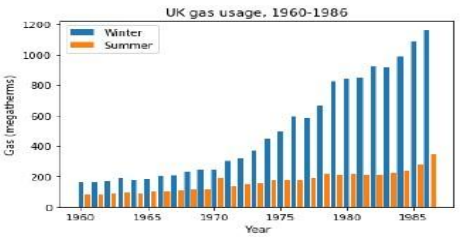
Regression and smoothing
Regression involves finding an approximate function to some data. The most familiar example is linear regression, which fits a line to the data.
Smoothing finds a simplified version of the data by removing fast changes in the values.

Geoms
Markers are geoms that represent bare points in a coordinate system.
Layer identification can be achieved by modifying both the shape and colouring of markers to distinguish different layers.

Colour choices
Colour is essential for comprehension.
Care should be taken if plots may be viewed in black and white printouts.
A significant portion of the population suffers from some form of colour blindness, so this should be considered when choosing colour schemes.

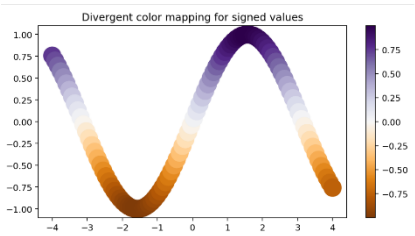
Coords
Axis limits specify a range in data units, which are mapped onto the figure space in visual units.
In matplotlib, visual units can be controlled using figsize when creating a figure.
Axis limits can be set with ax.set_xlim() and ax.set_ylim().
Aspect ratio can be adjusted with set_aspect() and is especially important for images to avoid distortion.
A coordinate system can include a 2D projection and transformations.
Log scales are used when data spans several orders of magnitude.
semilog x uses a log scale on the x-axis, semilog y on the y-axis, and log-log on both axes.
Log of a negative number is undefined and diverges to $-\infty$ at 0, so symmetric log scales may be used:
they exclude a region around 0 and plot log(abs(x)) × sign(x).
Polar coordinates map values to an angle θ and radius r.
Polar plots are typically used for angular measurements and only support positive radii.
There is no natural way to represent negative radii on a polar plot, so these should be avoided in such cases.





Markers: scalar attributes
Markers can also be used to display a third or fourth scalar attribute, visualising not just x, y but x, y, z or even x, y, z, w.
This can be done by modulating the scaling or colouring of each marker.

Colour maps
Colour maps map scalar values to colours and should always be presented with a colour bar showing the mapping of values to colours.
If the data is an unsigned scalar (positive only), use a colour map with monotonically varying brightness, where the colour gets consistently lighter or darker as the attribute increases.
If the data is a signed scalar (positive and negative values), use a colour map that diverges around 0 and is monotonic in brightness on each side of 0.



Facets and layers

Multiple geoms can be displayed either using distinct layers or separate facets.

Layering is appropriate when different views of the dataset are closely related and use the same coordinate system and units.

Legends are essential to distinguish different layers.

A double y-axis is an example of layering with shared x-axis but different y-axes.

This should be avoided where possible due to difficulty in interpretation.

Facets use separate coordinate systems to show different aspects of the dataset.

Facets don't need to share the same scale or range, but if they display the same attribute, they should use the same scale when possible for comparison.

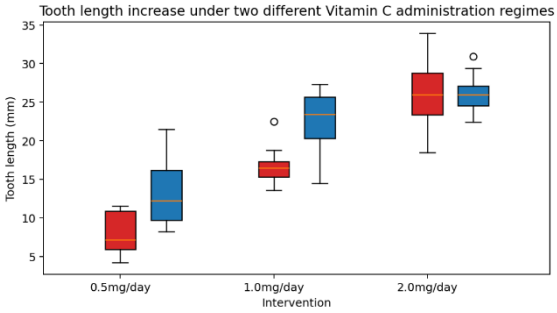
Facet layouts are usually arranged in a grid using fig.add_subplot(rows, columns, index).

Communicating uncertainty

Scientific visualisations need to be honest, so uncertainty should be shown appropriately. Data are often collected with observation error, and this needs to be communicated clearly in visualisations. Statistics like standard deviation and interquartile range can summarise the spread or variation of a dataset.

Error bars

Error bars can be used to represent uncertainty and can be chosen from various options such as standard deviation, standard error, confidence intervals (e.g., 95%), or nonparametric intervals (e.g., interquartile range).



Unit 4 Vector Spaced and Matrices

Example: Text and translation

There are comparison functions for strings, but they work only at the character level. Words alone aren't sufficient for meaningful text comparisons. Embedding is the process of embedding text fragments with additional mathematical structure by placing them in a vector space. This allows one to ask questions like:

What words are similar to…?

What word is equivalent to… but with the property… instead of…?

Vector spaces

Consider vectors as ordered tuples of real numbers. A vector has a fixed dimension, which is the length of the tuple. Each element of the vector represents a distance in a direction orthogonal to the other elements (orthogonal means independent or at 90° to each other).

For example, a vector [5,7,3] can be viewed as a weighted sum of three orthogonal unit vectors:

[5,7,3] = 5 × [1,0,0] + 7 × [0,1,0] + 3 × [0,0,1]

Geometric operations

Vectors are often used to represent 2D or 3D geometric data. Nearly all computations in modern computer games or 3D rendering engines involve low-dimensional vector operations performed at enormous scale. Standard transformations include:

Scaling, Rotation, Flipping (mirroring) and Translation (shifting)

Specialized operations include:

Colour space transformations

Estimating the surface normals of a triangle mesh (indicating their direction)

Graphical pipelines process everything as large arrays of vectors. Shader languages, like HLSL and GLSL, have special types and operations for working with low-dimensional

Machine learning applications

Vector representation is crucial in machine learning:

Data is often transformed into feature vectors.

A function is created to transform feature vectors into predictions.

Feature vectors encode the data in vector space, and feature transforms take raw data and output feature vectors.

Most machine learning processes can be seen as geometric operations. For example, one of the simplest effective algorithms is k-nearest neighbours. When classifying a new feature for prediction, the k-nearest vectors from the training set are computed, using a norm to measure distances. The output prediction is the class label that appears most frequently among these k neighbours, with k typically preset (often between 3 and 12 for many problems).

Points in space

Notation:

$\mathbb{R}$ = set of real numbers

$\mathbb{R}\geq 0$ = set of non-negative reals

$\mathbb{R}^n$ = set of tuples of exactly n real numbers

$\mathbb{R}^{n\times}{}_m$ = set of 2D arrays of real numbers with exactly n rows and m elements

$(\mathbb{R}^n, \mathbb{R}^n) \rightarrow \mathbb{R}$: means the operation defines a map from a pair of n-dimensional vectors to a real number

Vector spaces

Any vector of a given dimension n lies in a vector space called $\mathbb{R}^n$, which is a set of all possible vectors of length n with real elements.

The operations within a vector space include:

Scalar multiplication: For any scalar a, ax is defined as:

$ax = [ax_1, ax_2, \ldots, axn]$

Vector addition: For two vectors x and y of equal dimensions:

$x + y = [x_1 + y_1, x_2 + y_2, \ldots, xn + yn]$

Norm: The length of a vector is measured as $||x||$

Inner product: Comparing the angles of two vectors (equals 0 for orthogonal vectors)

$\langle \mathbf{x}|\mathbf{y}\rangle$ or $\mathbf{x} \bullet \mathbf{y}$

All operations between vectors can be defined within a vector space, but not between vectors of different dimensions.

Image compression

Images are typically represented as 2D arrays of brightness. On their own, these numbers have little meaning. One common approach to image compression involves splitting the image into patches, treating each as a vector. These patches are clustered to find a small set of representative vectors. These vectors are stored in a codebook, and the rest of the image is represented by the indices of the closest matching vector in the codebook. This process is known as vector quantisation.

Unit vectors and normalisation

The definition of a unit vector depends on the norm used.

To normalise a vector for the Euclidean norm, scale the vector x by $1/||x||_2$.

Unit vectors always have a length of 1.

Topological vector space

A topological vector space is one that is continuous, meaning it makes sense to talk about vectors being "close together" or having a neighborhood around them.

Vectors in different contexts

Vectors can be viewed as points in space, as arrows pointing from the origin, or as tuples of numbers.

Relation to arrays

Vectors can be represented by 1D floating-point arrays, but it's important to note that the representation and the mathematical concept of vectors are distinct.

Uses of vectors

Vectors serve as a 'lingua franca' for data, meaning they can be:

Composed (addition)

Compared (using norms/inner product)

Weighted (scaling)

Mapped onto efficient ndarray structures

Vector data

Datasets are commonly stored as 2D tables, similar to a list of vectors. Each row represents a vector (in $\mathbb{R}^n$) corresponding to an observation, and each column represents one element of the vector across many observations. The entire matrix is a sequence of vectors in the same vector space.

Basic vector operations

Addition and multiplication

Vectors can be combined through weighted sums: $\lambda_1 x_1 + \lambda_2 x_2 + \ldots + \lambda_n x_n$, where the vectors must all be of the same dimension.

Linear interpolation allows us to move smoothly between two vectors. As α goes from 0 to 1, the result moves in a straight line from x to y. The equation for linear interpolation is:

$\text{lerp}(x, y, \alpha) = \alpha\, y + (1 - \alpha)\, x$

How big is that vector?

The Euclidean length (or magnitude) of a vector x can be computed using np.linalg.norm():

$||x||_2 = \sqrt{(x_0^2 + x_1^2 + x_2^2 + \ldots + x_n^2)}$

Different norms:

The distance between two vectors is given by: $||x - y||_2$

Lp norms (also known as Minkowski norms) are another way to measure vector length or distance.

$$\|\mathbf{x}\|_p = \left(\sum_i x_i^p\right)^{\frac{1}{p}}$$

p	Notation	Common name	Effect	Uses	Geometric view
2	$\|x\|$ or $\|x\|_2$	Euclidean norm	Ordinary distance	Spatial distance measurement	Sphere just touching point
1	$\|x\|_1$	Taxicab norm; Manhattan norm	Sum of absolute values	Distances in high dimensions, or on grids	Axis-aligned steps to get to point
0	$\|x\|_0$	Zero pseudo-norm; non-zero sum	Count of non-zero values	Counting the number of "active elements"	Numbers of dimensions not touching axes
∞	$\|x\|_\infty$	Infinity norm; max norm	Maximum element	Capturing maximum "activation" or "excursion"	Smallest cube enclosing point
−∞	$\|x\|_{-\inf}$	Min norm	Minimum element	Capturing minimum excursion	Distance of point to closest axis

Inner products of vectors (dot product)

The dot product measures the angle between two real vectors.

The formula is:

$\cos\theta = (x \cdot y) / (||x||\ ||y||)$

For unit vectors (since $||x|| = 1$), this simplifies to: $\cos\theta = x \cdot y$

The dot product itself is: $x \cdot y = \sum x_i y_i$

The dot product is particularly useful for comparing vectors that may have very different magnitudes.

Basic vector statistics

The mean vector of a collection of N vectors (geometric centroid) is calculated as:

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

If the vectors are stacked in a matrix X (one vector per row), np.mean(x, axis=0) will calculate the mean vector.

To center a dataset, subtract the mean vector from every row.

The median is harder to calculate and doesn't have a simple algorithm for it.

Basic vector statistics

The mean vector of a collection of N vectors (geometric centroid) is calculated as:

$\text{mean}(x_1, \ldots, x_n) =$

If the vectors are stacked in a matrix X (one vector per row), np.mean(x, axis=0) will calculate the mean vector.

To center a dataset, subtract the mean vector from every row.

The median is harder to calculate and doesn't have a simple algorithm for it.

High-dimensional vector spaces

These spaces obey the same mathematical rules but not the same properties as lower-dimensional spaces. For high-dimensional spaces (D > 3), the following properties apply:

Volume increases exponentially.

There is a lot of empty space in high dimensions.

The curse of dimensionality: algorithms break down in higher dimensions.

For example, with 10 different measurements and 20 bins each, we would need a histogram with 20^{10} bins — over 10 trillion bins.

Paradoxes of high-dimensional vector spaces:

Even if we take two random points in a high-dimensional cube and draw a line between them, the points on the line still end up on the edge of the space.

Distances between two random high-dimensional vectors are almost the same as for

Matrices and linear operators

Uses of matrices:

Matrices represent operations that perform transformations on vectors in space (rigid transformations).

Operations with matrices:

Matrices can be added and subtracted:

$C = A + B$

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

Matrices can be scaled with a scalar:

$C = sA$

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

Matrices can be transposed:

$B = A^T$ (exchange rows and columns)

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

Matrices can be applied to vectors:

$y = Ax$

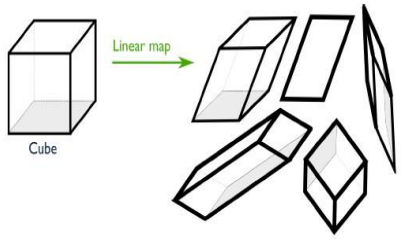
$(\mathbb{R}^{n\times m}, \mathbb{R}^m) \to \mathbb{R}^n$

Matrices can be multiplied together:

$C = AB$ (composes the effect of two matrices)

$(\mathbb{R}^{p\times q}, \mathbb{R}^{q\times r}) \to \mathbb{R}^{p\times r}$

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



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

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

$$x = \begin{bmatrix} 1 & 2 & 3 \end{bmatrix} \qquad A = \begin{bmatrix} 1 & -1 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 1 \end{bmatrix} \qquad B = \begin{bmatrix} 2 & -5 & 5 \\ 1 & 0 & 0 \end{bmatrix}$$

$$Ax = \begin{bmatrix} 2 & 1 & 5 \end{bmatrix} \qquad Bx = \begin{bmatrix} 7 & 1 \end{bmatrix}$$

Matrix Multiplication

$$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \\ a_{31} & a_{32} \end{bmatrix} \cdot \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix}$$

$$= \begin{bmatrix} a_{11}b_{11} + a_{12}b_{21} & a_{11}b_{12} + a_{12}b_{22} \\ a_{21}b_{11} + a_{22}b_{21} & a_{21}b_{12} + a_{22}b_{22} \\ a_{31}b_{11} + a_{32}b_{21} & a_{31}b_{12} + a_{32}b_{22} \end{bmatrix}$$

Intro to matrix notation

Matrices as maps

Matrices are applied to vectors by multiplying them.

Note that code indexes from 0, whereas mathematical notation indexes from 1!

An n × m matrix A takes m-dimensional vectors to n-dimensional vectors, such that all straight lines remain straight, all parallel lines remain parallel, and the origin does not move.

Linearity:

$f(x + y) = f(x) + f(y) = A(x + y) = Ax + Ay$

$f(cx) = cf(x) = A(cx) = cAx$

This means that the transformation of the sum of two vectors is the same as the sum of the transformations of the individual vectors, and the transformation of a scalar multiple of a vector is the same as the scalar multiple of the transformation of the vector. This property is called linearity, and matrices represent linear maps or linear functions.

Geometric intuition:

A cube can be transformed into a parallelepiped through matrix operations.

Linear Map

A linear map is any function such that $\mathbb{R}^m \to \mathbb{R}^n$ that satisfies linearity requirements.

If the map is n × n, then the vector is linearly transformed from a vector space to the same vector space.

Linear Projection:

$Ax = AAx$ or $f(x) = f(f(x))$

Every linear map of real vectors can be written as a real matrix.

Matrix Operations

Matrix Multiplication

Matrix multiplication defines the product C = AB, where all three are matrices.

If A = f(x) and B = g(x), then BAx = g(f(x)). Multiplying two matrices is equivalent to composing the linear functions they represent, and the result is a matrix that has that effect.

Matrix multiplication is only defined for two matrices A and B if:

- A is p × q
- B is q × r

If C = AB, then:

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

In NumPy, matrix multiplication can be applied using 'np.dot(a, b)' or 'a @ b'.

The time complexity of matrix multiplication is greater than $O(N^2)$ but less than $O(N^3)$.

Apply Matrices to Vectors

Some algorithms for multiplying two matrices also apply when multiplying a matrix by a vector.

The vector has to be represented as an m × 1 column vector.

For a vector m × 1, the matrix must be n × m for the operation to be defined.

If the matrix is m × m, then it is a linear transform, and the result is another vector of the same dimension.

Column and Row Vectors

The product of an M × 1 and 1 × N vector is an M × N matrix. This is the outer product of two vectors, which represents every possible combination of their elements.

The product of a 1 × N and N × 1 vector is a 1 × 1 matrix. This is exactly the inner product of two vectors.

Composed Maps

Multiplication is composition: BA = g(f(x))

Order is important! Perform A first, then B.

Concatenation of transforms allows multiple operations to be combined into a single matrix operation.

Multiplication is not commutative: BA ≠ AB

Transpose Order Switching:

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

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

Left-Multiply:

A[Bx + y] = ABx + Ay

Right-Multiply:

[Bx + y]A = BxA + yA

$$\mathbf{x} \otimes \mathbf{y} = \mathbf{x}^T\mathbf{y} \qquad \mathbf{x} \bullet \mathbf{y} = \mathbf{x}\mathbf{y}^T$$

$$\mathbf{x} = \begin{bmatrix} 1 & 2 & 3 \end{bmatrix} \qquad \mathbf{y} = \begin{bmatrix} 4 & 5 & 6 \end{bmatrix} \qquad \mathbf{x} \bullet \mathbf{y} = \begin{bmatrix} 32 \end{bmatrix}$$

$$\mathbf{x} \otimes \mathbf{y} = \begin{bmatrix} 4 & 5 & 6 \\ 8 & 10 & 12 \\ 12 & 15 & 18 \end{bmatrix}$$

Covariance Matrices

Variance measures the spread of a dataset: the sum of squared differences of each element from the mean of the vector.

Variance(σ2) formula below

Standard deviation σ is the square root of the variance and is more commonly used.

In a multidimensional case, the covariance of each dimension with every other dimension needs to be computed. This results in a 2D array ∑, where each element in position i,j represents the covariance between the i-th and j-th dimensions.

$\sum_{i,j} = \frac{1}{N-1}\sum_{i=0}^{N} \quad (X_{ki} - \mu_i)(X_{kj} - \mu_j)$

Covariance matrices can be computed using NumPy's np.cov(x).

Covariance Ellipses:

Covariance ellipses capture the spread of data, including any correlations between dimensions.

The ellipse represents the shape of the dataset, with the mean vector at the center of the ellipse.

The covariance matrix represents the shape and orientation of the ellipse.

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

Anatomy of a matrix

Diagonals of a matrix are important.

Diagonal matrices are zero everywhere except for a single diagonal and represent a transformation that is an independent scaling of each dimension.

np.diag(x) returns a diagonal matrix for a given vector x.

Antidiagonal matrices have non-zero elements on the reverse diagonal (from top-right to bottom-left).

$$\text{anti-diagonal} = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 2 & 0 \\ 3 & 0 & 0 \end{bmatrix}$$

Special matrix forms

Identity matrix: A square diagonal matrix for which all diagonal elements are 1, has no effect when multiplied by another matrix: IA = A = AI.

Any scalar multiple of an identity corresponds to a function that uniformly scales vectors.

Zero matrix: A matrix with all zeros, multiplying by it results in the original vector/matrix shape filled with zeros.

Square matrices: Have an n × n shape and possess:

- An inverse
- Determinants
- An eigendecomposition

Triangular matrices: Have non-zero elements only above (upper triangular) or below (lower triangular) the diagonal (including the diagonal).

Unit 5 Computational Linear Algebra

Graphs as matrices

An **adjacency matrix** is a square matrix of |V| × |V| size (where |V| = number of **vertices**). In this matrix, no edge from vertex **Vi** to vertex **Vj** means 0, and existing edges are represented by 1.

Out-degree is the sum across the rows of the adjacency matrix.

In-degree is the sum across the columns of the adjacency matrix.

A **symmetric matrix** indicates an **undirected graph**.

A **directed graph** can be turned into an **undirected graph** using the formula: **A' = A + AT**.

A non-zero **diagonal** means there are edges from a **vertex** to itself.

Edge weights represent edges with a weight different from 1.

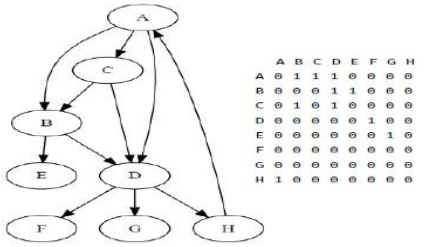
Static matrices are used in graphs to represent flow properties:

If the weight of the total flow out of a **vertex** is greater than 1, it represents a **source of weight** (e.g., a **manufacturer**).

If the weight is less than 1, it represents a **sink** (e.g., a **consumer**).

If the weight equals 1, it represents a **rerouting** of things, conserving **mass**.

A **stochastic matrix** is a square matrix with non-negative elements, where each **row** or **column** sums to 1.



Matrix powers

These are only defined for **square matrices** (otherwise, the shape could change).

It represents the repeated application of a **matrix**:

A^2 = AA, **A^3 = AAA**, etc.

Decompositions

This involves breaking down a mathematical object, e.g., **30 = 2 × 3 × 5**.

Similarly, you can decompose **matrices** into **vectors** or **special form matrices**.

Uses

- **Analyze datasets** represented as matrices.
- **Analyze the effect** of **linear maps** that matrices represent.
- Perform **matrix computations** efficiently.
- Compute interesting **matrix functions**, such as the **square root** of a matrix (**A^1/2**), **logarithm** (**log(A)**), or **inverse** (**A^{-1}**).

A fixed point is any point of a function **f(x)** such that **f(x) = x**.

A **stable fixed point** is one where applying **f(x)** to any point in its neighborhood moves it closer to the fixed point.

An **unstable fixed point** is one where applying **f(x)** to any point in its neighborhood moves it further away.

Types of stochastic matrices

A **right stochastic matrix** has each **row** summing to 1.

A **left stochastic matrix** has each **column** summing to 1.

A **doubly stochastic matrix** has both **rows** and **columns** summing to 1 (even if the graph is reversed, **mass** is conserved).

The degree matrix and the Laplacian

The **degree matrix** (**D**) is a **diagonal matrix** whose diagonal elements are the sum of the **weights** of the edges connected to each **vertex**. It is computed by summing the rows of the **adjacency matrix**.

$$D_{ii} = \sum_j A_{ij}$$

```
A = [0  2  3  0]   D = [5  0  0  0]
    [2  0  0  1]       [0  3  0  0]
    [3  0  0  5]       [0  0  8  0]
    [0  1  5  0]       [0  0  0  6]
```

The **Laplacian matrix** of a graph is defined as the difference between the **degree matrix** and the **adjacency matrix**. This is usually computed for **undirected graphs**, where the **adjacency matrix** is **symmetric**.

$$L = D - A$$

A **symmetric matrix** is one that is equal to its **transpose**, reflecting about its **transpose**. **Symmetric matrices** represent **undirected graphs**.

A **sparse matrix** is one where most of the elements are zero. **Sparse matrices** are common in practice, as most things are not connected to most other things. The opposite is a **dense matrix**.

A **stochastic matrix** is a square matrix with non-negative real numbers, with each **row** summing to 1.0. If the **columns** sum to one instead of the **rows**, **mass** is preserved in reverse.

From discrete to continuous

Analyze discrete problems (like connectivity of **graphs**) using tools from **continuous mathematics**.

Flow analysis:

This may involve an initial distribution of quantities over **vectors**: a vector **xt=0**, and a set of **weighted edges** that give flow from one **vertex** to another.

New operations:

Exponentiate: C = **An** ($\mathbb{R}n\times n$, $\mathbb{R}$) -> $\mathbb{R}n\times n$

Invert: C = **A^{-1}** ($\mathbb{R}n\times n$ -> $\mathbb{R}n\times n$)

Factorise: This breaks any **matrix** into three parts with special forms ($\mathbb{R}n\times m$ -> $\mathbb{R}m\times m$, $\mathbb{R}n$, $\mathbb{R}n\times n$)

Find stable points: **Ax = λx**; find **vectors** which are only scaled by a factor **λ** by a particular **matrix A** $\mathbb{R}n\times n$ -> [($\mathbb{R}$, $\mathbb{R}n$), ($\mathbb{R}$, $\mathbb{R}n$),...]

Measure properties of A numerically, like **determinant**, **trace**, or **condition number**.

From discrete to continuous

Analyze discrete problems (like connectivity of **graphs**) using tools from **continuous mathematics**.

Flow analysis

This may involve an initial distribution of quantities over **vectors**: a vector **xt=0**, and a set of **weighted edges** that give flow from one **vertex** to another.

New operations

Exponentiate: C = **An** ($\mathbb{R}n\times n$, $\mathbb{R}$) -> $\mathbb{R}n\times n$

Invert: C = **A^{-1}** ($\mathbb{R}n\times n$ -> $\mathbb{R}n\times n$)

Eigenvalues: Identify specific vectors that xi that are only scaled by a factor (not rotated) when transformed by matrix A

$$A\vec{x_i} = \lambda_i \vec{x_i};$$

Factorise: $A = U\Sigma V^T$ This breaks any **matrix** into three parts with special forms ($\mathbb{R}n\times m$ -> $\mathbb{R}m\times m$, $\mathbb{R}n$, $\mathbb{R}n\times n$)

Find stable points: **Ax = λx**; find **vectors** which are only scaled by a factor **λ** by a particular **matrix A** $\mathbb{R}n\times n$ -> [($\mathbb{R}$, $\mathbb{R}n$), ($\mathbb{R}$, $\mathbb{R}n$),...]

Measure properties of A numerically, like **determinant**, **trace**, or **condition number**.

Eigenproblems

A **matrix** is just a **linear map**.

A **linear transform** is a linear map from a **vector space**.

Linear transforms have important points that are a close analogue of fixed points: **eigenvalues** and **eigenvectors**.

Eigen means **characteristic**.

An **eigenvector** of **A** is a vector **x** that is only scaled when **A** is applied to it, not rotated.

An **eigenvalue** of **A** is how much the **eigenvector x** is scaled by when **A** is applied to it.

Eigenproblems involve problems relating to fundamental characteristics of **matrices**.

Eigenvectors and **eigenvalues** allow us to analyse the effect of **matrices**.

Properties of eigenvalues

For any matrix, the **eigenvalues** are uniquely determined, but **eigenvectors** are not. There may be many eigenvectors corresponding to any given eigenvalue.

The **shape** of any given matrix can be understood from the **eigendecomposition**—this allows us to identify which dimensions are large or small.

Eigendecomposition of the Laplacian

The **eigendecomposition** of the **Laplacian matrix** is particularly important.

For example, the number of **connected components** in a graph is equal to the number of **zero eigenvalues** of the **Laplacian matrix**.

The second smallest eigenvalue of the **Laplacian matrix** is called the **Fiedler value**. It tells us how well connected the graph is. If the **Fiedler value** is small, the graph is well connected. If it is large, the graph is poorly connected. The corresponding **eigenvector** is called the **Fiedler vector** and can be used to partition the graph into two relatively disconnected parts.

The **Laplacian matrix** is also used in **spectral clustering**, a method for clustering data based on the **eigenvectors** of the **Laplacian matrix**.

A word of warning

np.linalg.eigh 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.eigh** is often sufficient for most purposes, but you should be careful how you use it.

If you are using **non-symmetric matrices**, then you need to use a general-purpose **eigenvector solver** such as **np.linalg.eig**. This is more prone to numerical instability, and you should be careful to check that the **eigenvectors** are orthogonal.

Power iteration method

Repeatedly apply the **linear transform** we are interested in to a **random initial value** and observe the result.

Normalize the resulting vector after each step and compute the value in an **iterative process** to get a proper result.

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

Regardless of the starting vector, the **power iteration** always approaches a **fixed point**—this holds for almost every **square matrix**.

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

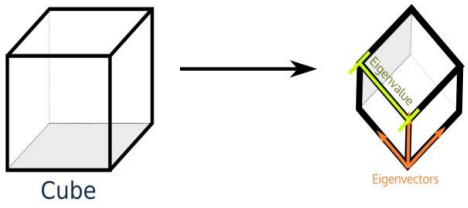
The vector obtained in this way is the **leading eigenvector**.

The **leading eigenvector** satisfies the property that the matrix maps the vector onto a pure scaling operation: **Ax = λx**.

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

A x scale factor =

λ is the **leading eigenvalue**.



Dimensionality reduction

We can 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.

Reconstruction

Take all of the **eigenvectors** $\mathbf{x}_i$, normalize them to be unit length, and stack them into a matrix **Q**.

Take the **eigenvalues λ** and put them into a diagonal matrix **Λ**.

Then, the original matrix **A** can be reconstructed as:

$\mathbf{A = Q\Lambda Q^T}$

You can approximate a matrix by truncating it—represent only the most important effects on a vector space and set the smallest **eigenvalues** to 0.

Formal definition of eigenvectors and eigenvalues

Consider a vector function **f(x)**. There may exist vectors such that **f(x) = λx**. These are the points where the function maps vectors to scaled versions of themselves, with no rotation or skewing applied.

Any given **square matrix A** represents a function **f(x)** and may have points like this, such that $\mathbf{A}\,\mathbf{x}_i = \lambda_i\,\mathbf{x}_i$.

Each vector **x** satisfying this equation is known as an **eigenvector**, and each corresponding factor **λ** is known as an **eigenvalue**.

Multiple eigenvectors

In **NumPy**, you can use **np.linalg.eig()** to calculate the eigenvalues and eigenvectors.

Eigenspectrum

The **eigenspectrum** is the sequence of absolute **eigenvalues**, ordered by absolute value:

$|\lambda_1| > |\lambda_2| > \ldots > |\lambda_n|$

Eigenvectors of the covariance matrix (aka principal components)

The **axes of the parallelotope** are the **eigenvectors**.

The **magnitude of the axes** is given by the **eigenvalues**.

This tells us in which direction the **data** varies most.

The direction of principle component i is given by the eigenvector xi, and the length of the component given by $\sqrt{\lambda_i}$.

Which way is up

Eigendecomposition can be used anywhere there is a system modeled as a **linear transform** ($\mathbb{R}^n \to \mathbb{R}^n$). It allows us to predict behavior over different time scales.

Uses of **eigendecomposition** include:

- Finding **modes** or **resonances** in a system
- Predicting the behavior of **feedback control systems**
- Partitioning **graphs** and **clustering data**
- Predicting **flows on graphs**
- **Exploratory analysis** or **data compression**

Matrix properties from the eigendecomposition

Trace is the sum of a matrix's diagonal values, and it is equal to the sum of the **eigenvalues** of **A**:

$\mathbf{Tr(A) = a_{11} + a_{22} + \ldots + a_{nn} = sum\ of\ the\ eigenvalues}$

This is almost the perimeter of the **parallelotope** of a **unit cube** transformed by the matrix:

Perimeter(A) = 2^{n-1} Tr(A), where the constant of proportionality is the same.

Determinant measures how much the space expands or contracts after the **linear transform**. The determinant of **A** is the product of its eigenvalues:

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

If any eigenvalue **λ** is 0, then **det(A) = 0**, and the transformation cannot be reversed.

Definite and semidefinite matrices

Positive definite matrices have all eigenvalues $\lambda_i > 0$.

Positive semidefinite matrices have all eigenvalues $\lambda_i \geq 0$.

Negative definite matrices have all eigenvalues $\lambda_i < 0$, and **negative semidefinite** matrices have eigenvalues $\lambda_i \leq 0$.

For a **positive definite matrix**, vectors are never "flipped around" if $\mathbf{x}^T\mathbf{Ax} > 0$ for all nonzero vectors **x**. This means the dot product of **x** with **Ax** must be positive, which can only happen if the angle between them is less than $\pi/2$.

Real and complex eigenvalues

All eigenvalues of a matrix will be real if the matrix is both real and symmetric.

For real, non-symmetric matrices, the eigenvalues can be complex numbers of the form **a + bi** (and they will also have a matching conjugate eigenvalue **a - bi**).

Complex eigenvalues are related to oscillatory behavior in linear systems, while **real eigenvalues** correspond to scaling behavior.

If there are only complex eigenvalues, the system exhibits oscillating flows without any real eigenvalues.

Inversion

Key Properties

$A^{-1}(Ax) = x$

$A^{-1}A = I$

$(A^{-1})^{-1} = A$

$(AB)^{-1} = B^{-1}A^{-1}$

Conditions for Inversion

Only defined for certain kinds of matrices

Only for square matrices: This is equivalent to saying that the determinant of the matrix must be non-zero: $\det(A) \neq 0$

Singular – $\det(A) = 0$ has no inverse

Non-singular – $\det(A) \neq 0$ is invertible

Computational Tools

NumPy can compute an inverse using np.linalg.inv(x)

Purpose of Inversion

Inversion can undo: inversion of a matrix creates a new linear operator which reverses the original operation

Performance Considerations

Numerical stability – algorithms converge to the right answer reliably

Time complexity – $O(n^3)$

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 are only 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, for example, by the **power iteration algorithm** to find 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 the list of **absolute eigenvalues** of a matrix, ordered by magnitude, from largest to smallest.

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 **x**, the vector will be stretched more and more along the largest eigenvectors.

Special Matrix Forms

Orthogonal matrices (rows and columns are all orthogonal unit vectors): $O(1) \rightarrow A^{-1} = A^T$

Diagonal matrices: $O(n) \rightarrow A^{-1} = 1/A$

Positive-definite matrices: $O(n^2)$

Triangular matrices: $O(n^2)$, trivially invertible by elimination algorithms

Applications

Solving problems with inversion:

If A^n allows us to "see" into the future

A^{-n} allows us to look back

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

ıber of steps:

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

But roundoff error will lead to distorted results

Linear Systems

One way to look at matrices is as a collection of weightings of components of a vector, e.g.

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

each component is a weighted sum of the components of the input vectors

If we have A and y, what x is needed to have Ax = y

Things we can tell from eigenvectors/values

If a matrix has one or more zero eigenvalues, the transform it performs collapses one or more dimensions in vector space. This type of operation is irreversible, and this tells us that **A** is singular (non-invertible).

Eigenvectors corresponding to larger (absolute) eigenvalues are more "important"; they represent directions in which data will get stretched the most.

If the **eigenspectrum** is nearly flat (all eigenvalues 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 many 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.

Solving Linear Systems

Square Matrices

If A is square: $Ax = y$ then $x = A^{-1}y$

Numerical Issues

Direct inversion is almost never used to solve linear systems; instead, use iterations based on optimisation

Singular Value Decomposition (SVD)

Definition

SVD splits any matrix into three matrices:

$A = U\Sigma V^T$, where a unitary matrix is one whose conjugate transpose is equal to its inverse

Matrix Components

A – m × n matrix

U – square orthogonal m × m matrix (left singular vectors)

V^T – square orthogonal n × n matrix (right singular vectors)

Σ – diagonal m × n matrix (diagonal contains the singular values)

Properties

An orthogonal matrix is one where $U^{-1} = U^T$, with rows and columns all having unit norm

Σ's diagonal contains singular values – always positive real numbers (like eigenvalues but not exactly the same)

Computation

np.linalg.svd(x) computes the SVD

Relation to Eigendecomposition

Eigenvectors of A^TA

Taking the eigenvectors of A^TA gives U

Taking the square root of the absolute value of the eigenvalues λ_i of A^TA to get $\Sigma_i = \sqrt{|\lambda_i|}$

Eigenvectors of AA^T

Taking the eigenvectors of AA^T gives V^T

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.

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

Using the SVD

Fractional Powers (for Symmetric Matrices)

"Part do" an operation – raise matrix to fractional power, e.g., A

"Undo" their operation – invert matrices A^{-1}

Approximately "undo" an operation even if A isn't square – pseudo-invert matrices A^T

To do any of these operations, ignore U and V and apply the function to the singular values elementwise

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 = VΣ^nU^T

For a symmetric matrix, this is the same as:

A^n = UΣ^nU^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) = UΣ^(1/2)V^T.

Inversion

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

Pseudo-Inverse

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 generalizes inversion to (some) non-square matrices.

We can find approximate solutions for x in the equation:

Ax = 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.

Fitting Lines/Planes

Nearest solution to AX = Y, $A = X^+Y$

Singular Values

Rank of a Matrix

Number of non-zero singular values

Full rank matrix – rank is equal to the size of the matrix (no zeros)

Deficient rank matrix – not full rank, meaning it is singular

Low rank – if the number of non-zero singular values is much less than the size

Condition Number of Singular Values (only defined for full rank matrices)

Ratio of the largest singular value to the smallest

Well-conditioned – small condition number (unlikely to cause numerical issues)

Ill-conditioned – numerical issues are likely to be significant

Applying Decompositions

Whitening

Removes all linear correlations within a dataset

Given a dataset matrix X:

$$X^{\mathbf{w}} = (X - \vec{\mu})\Sigma^{-1/2}$$

Where μ is the mean vector

Σ is the covariance matrix

Properties of Whitening

Centers the data around its mean, so it is zero mean

"Squashes" the data so that it is spherical and has unit covariance

Steps to Achieve Whitening

To get Σ^{-2}:

- Compute the inverse square root in one step (take the reciprocal of the square root of the diagonal components from the SVD, then reconstruct)
- Remove all linear relationships (important when applying machine

Low-Rank Approximations

Sums of Stereotypes

For a "movie watching" data example:

- Every user can be written as a linear sum of stereotypes (e.g., romantic comedy fan)
- Every movie can also be written as a linear sum of these stereotypes

Low-Rank Approximation

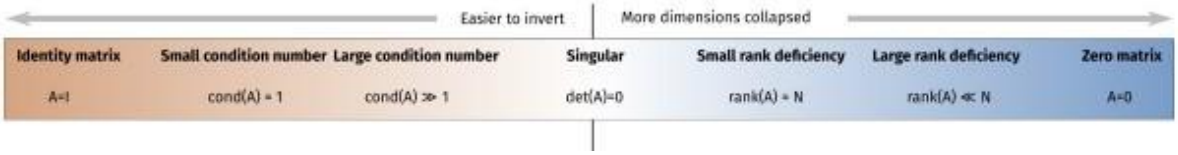
Use SVD to find a low-rank approximation to a matrix

It is "low rank" because the resulting matrix only has a few different "directions" represented – the stereotype directions

Truncate the SVD by keeping only the leading k singular values, the first k columns of U, and the first k rows of V

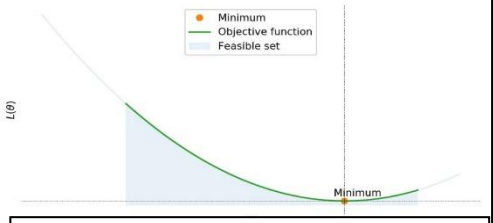
$X \approx U_k\Sigma_k V_k$

This is a form of dimensional reduction



Unit 6: Introduction to Optimisation

Optimisation
Process of adjusting things to make them better
In computer science, optimisation can be thought of as a search, and optimisation algorithms search for efficiency using the mathematical structure of the problem space
Formulate the problems so that generic optimisation algorithms can solve them



Types of Optimisation
Discrete vs. Continuous
Continuous space → continuous, discrete parameters → discrete
Continuous Optimisation
Continuous optimisation is usually easier due to smoothness and continuity
Example: Optimising an airfoil for maximum lift or minimum drag
Parameters are continuous (e.g., the angle of attack or the curvature of the camber line)
Measure the lift/drag ratio
Adjust airfoil parameters
Discrete Optimisation
Example: Bin packing
Goal: Find an assignment of elements ei with different sizes to best occupy a set of bins bj with defined capacity cj
Parameters: Assignment of items to bins
Objective function: Quality of packing (e.g., number of overfull or underfull bins)
Discrete optimisation often requires splitting the optimisation into subproblems, to which some sort of backtracking search can be applied

Parameters and Objective Function
Parts of an Optimisation Problem
There are two parts to an optimisation problem: parameters and the objective function
Parameters (θ)
Things we can adjust: scalar, vector, or other array of values. They exist in a parameter space θ – the set of all possible configurations of parameters.
Parameter space can be a vector space, in which case the parameters are parameter vectors.
Objective Function L(θ)
A function that maps the parameters onto a single numerical measure of how good the configuration is. The output is a single scalar. Also known as loss function, cost function, fitness function, utility function, or energy surface.

$$\theta^* = \arg\min_{\theta \in \Theta} L(\theta)$$

Where θ* is the configuration for which the objective function is lowest.
Θ is the set of all possible configurations that θ could take on
Constraints
Limitation on the parameters (also called feasible set/region)
Think of an objective function as a cost that is minimised (or maximised)

Every Optimisation Problem Has Two Parts
Parameters: The things that can be adjusted.
Objective Function: Measures how good a particular set of parameters are.
Focus: Continuous optimisation in real vector spaces
Iterative Optimisation Algorithms
Generate successively better approximations to a solution
Direct Optimisation
Involves solving a system of equations exactly in one step

Minimising Differences
Often we have to express problems in a form where the objective function is a distance between an output and a reference, measured as:
L(θ) = ||y' - y|| = ||f(x; θ) - y||, where f(x; θ) produces an output from x by a set of parameters
Evaluating the Objective Function
Evaluating the objective function might be expensive to compute:
Computation might take a long time
Might require a real-world experiment to be performed
Might be dangerous
Might require data that must be bought and paid for
A good optimisation algorithm will, using a mathematical structure, find the optimal configuration of parameters with few queries
Computational Modelling
If a computational model that authentically simulates how something will behave can be made and run quickly/cheaply, numerical optimisation becomes more feasible

Geometric Median: Optimisation in $\mathbb{R}^2$
Problem: Find the median of a >1D dataset
Optimisation: Find a point that minimises the distance to a collection of other points (with respect to some norm)
Parameters: θ = [x, y, …] (a position in 2D)
Objective Function: Sum of distances between a point and a collection of target points xi:
L(θ) = $\sum i \; ||\theta - xi||^2$
An Example of Optimisation in $\mathbb{R}^n$
Problem: Find a layout of points in such a way that they are evenly spaced
Optimisation: Optimise a whole collection of points by rolling them all up into a single parameter vector
Parameters: θ = [x_1, y_1, x_2, y_2, …] (an array of positions of points in 2D – a whole configuration of points is a single point in a vector space)
Loss Function: Sum of squares of differences between the Euclidean pairwise distances between points and some target distance:
$\sum i \sum j \; (\alpha - ||x_i - x_j||^2)^2$, where α is the distance between points

Constrained Optimisation
Constrained Optimisation
Constrained optimisation occurs if a problem has constraints on the parameters beyond purely minimising the objective function
Equality Constraint
Constraining the parameters to a volume to represent bounds on the values
Common Constraint Types
Box constraints – all θ lie in a box inside $\mathbb{R}^n$
Convex constraints – the constraint is a collection of inequalities on a convex sum of the parameters θ, the feasible set is limited by the interaction of many planes/hyperplanes
Unconstrained optimisation – no constraints, which may provide unhelpful results
Constraints and Penalties
Constrained optimisation – an optimisation algorithm that supports hard constraints inherently
Pros of Constrained Optimisation
a. Guarantees that the solution will satisfy constraints
b. May be able to use constraints to speed up optimisation

Cons of Constrained Optimisation
a. May be less efficient than unconstrained optimisation.
b. Fewer algorithms available for optimisation.
c. May be hard to specify feasible region with the parameters available in the optimiser.
oft Constraints
Apply penalties to the objective function to "discourage" solutions that violate the constraints
L(θ') = L(θ) + λ(θ), where λ(θ) is the penalty function with an increasing volume as the constraints are more shockingly violated
Pros of Soft Constraints
a. Any optimiser can be used
b. Can deal with soft constraints sensibly
Cons of Soft Constraints
a. May not represent important constraints, particularly if they are sharp
b. Can be hard to formulate constraints as penalties
c. Cannot take advantage of efficient search in constrained regions of space

A constrained optimisation might be written in terms of an equality constraint:

$$\theta^* = \arg\min_{\theta \in \Theta} L(\theta) \text{ subject to } c(\theta) = 0,$$

or an inequality:

$$\theta^* = \arg\min_{\theta \in \Theta} L(\theta) \text{ subject to } c(\theta) \leq 0,$$

where $c(\theta)$ is a function that represents the constraints.

Relaxation of Objective Functions
Relaxation is a technique used when facing difficult optimization problems.
Original hard problems (discrete or constrained optimization) are converted into similar but easier problems (continuous or unconstrained).
The simplified "relaxed" version is solved instead of the original problem.
Example: Constraints can be absorbed into the objective function to transform a constrained problem into an unconstrained one.
This approach makes complex optimization problems more computationally tractable while still providing useful solutions.

Algorithms
Direct Convex Optimisation: Least Squares
Sometimes a solution to an optimisation problem can be found in one step.
Linear least squares: Finds x closest to Ax = y.

$$\arg\min_{x} L(\mathbf{x}) = \|A\mathbf{x} - \mathbf{y}\|_2^2,$$

This function is convex - only has zero/min minimum as it doesn't have powers greater than 2 (no x^3 etc.).
Line fitting:
Find gradient m and offset c for y = mx + c such that squared distance to a set of observed (x, y) data points is minimized.
Parameters: θ = [m, c].
Objective function: $L(\theta) = \Sigma i(y - mxi + c)^2$.
Can be solved using pseudo-inverse via the SVD.

Iterative Optimisation
Involves making a series of steps in parameter space.
Current parameter vector (or collection of them) is adjusted at each iteration, hopefully decreasing the objective function.
Optimization terminates when some termination criteria have been met.

1. Choose a starting point x_0.
2. While the objective function is changing:
   A. Adjust parameters.
   B. Evaluate the objective function.
   C. If a better solution is found than any so far, record it.
3. Return the best parameter set found.

Bogosort - Joke sorting algorithm
Randomize the order of the sequence.
Check if it's sorted and if not, then randomize again.

Penalisation
Penalisation is a technique that modifies an objective function by adding terms that discourage certain solution properties.
Adds penalty terms to the objective function to approximate constrained optimization.
Common in approximation problems to find solutions that "generalize well."
Prevents overfitting by discouraging solutions that match training data too closely.
Transforms problems with hard constraints into simpler problems with modified objective functions.
Lagrange multipliers are a specific example of relaxing hard constraints into penalty terms.
This approach allows standard optimization methods to handle problems that would otherwise require specialized constraint-handling algorithms.
Penalty Function
A penalty function is just a term added to an objective function which will disfavour "bad solutions."
Penalty function approach: Modify the objective function to make excessive strength throws unattractive.

Regular Search: Grid Search
Straightforward but inefficient algorithm for multidimensional problems.
Parameter space is equally divided in each dimension, usually with a fixed number of divisions per dimension.
Objective function is evaluated at each θ on this grid and the lowest loss θ found so far is tracked.
Not feasible for higher dimensions.
Density: If the objective function isn't very smooth, then a much denser grid is needed to catch the minima.
Pros:

- Works for any continuous parameter space.
- Requires no knowledge of the objective function.
- Trivial to implement.

Cons:

- Incredibly inefficient (exponential search).
- Must specify search space bounds in advance.
- Highly biased to finding things near the "early corners" of the space.
- Depends heavily on the number of divisions chosen.
- Hard to tune so that minima are not missed easily.

properties of the Objective Function
Convexity, Global and Local Minima
Local minima: Any point where the objective function increases in every direction around that point.
Convex: Single, global minimum; local minima can still exist but cannot be the smallest possible values.
Convexity: Implies that finding any minimum is equivalent to finding the global minimum, either find the minimum or prove there is none (there is only one minimum).
Convex optimisation:
Constraints and objective function are linear (linear programming).
Quadratic objective function and linear constraints (quadratic programming).
Specialised cases (semi-quadratic programming, quadratically constrained quadratic programme).
Non-convex Problems
Require the use of iterative methods.
Continuity
For some very small adjustment to θ, there is an arbitrarily small change in L(θ), no sudden jumps in values.
Discontinuous Functions
Are not guaranteed to converge to a solution.

Hyperparameters
Hyperparameters control how optimization algorithms work (not what is being optimized).
Examples include search ranges and grid divisions in grid search.
They affect optimization results but aren't parameters of the objective function itself.
Ideal optimizers would have no hyperparameters (solution shouldn't depend on the search method).
In practice, all useful optimizers require some hyperparameters.
Fewer hyperparameters is generally better (easier to tune and use).
Performance of optimization algorithms depends on proper hyperparameter tuning.

Simple Stochastic: Random Search
Algorithm:
Gives a random parameter θ.
Check the objective function L(θ).
If L(θ) < L(θ*) (previous best), set θ* = θ.
Pros:

- Cannot get trapped in a local minima.
- Requires no knowledge of the structure of the objective function.
- Very simple to implement.
- Better than grid search almost always.

Cons:

- Extremely inefficient, use only if there is no mathematical alternative.
- Must be possible to randomly sample from the parameter space.
- Results do not necessarily get better over time; the best may be found on the first run or the 1000th run

Metaheuristics (Used to Improve Random Search)

Locality

Metaheuristics leverage the fact that the objective function is likely to have similar values for similar parameter configurations, assuming the objective function is continuous.

The method makes incremental changes to a solution, which can help improve it. However, it can also become trapped in a local minima.

Hill Climbing

Instead of drawing samples randomly from the parameter space, hill climbing randomly samples configurations near the current best parameter vector.

It makes incremental adjustments, keeping transitions to neighboring states only if they improve the loss.

Simple hill climbing adjusts one of the parameter vectors at a time, examining each direction in turn.

Stochastic hill climbing makes a random adjustment to the parameter vector and then accepts it if it improves the solution, rejecting it otherwise.

One issue is that hill climbing can get trapped behind ridges.

Pros

It is not much more complicated than random search, yet it can be much faster than random search.

Cons

It can be hard to choose how much of an adjustment to make and is susceptible to getting stuck in local minima.

It struggles with objective function regions that are relatively flat and requires that the objective function be (approximately) continuous.

Quality of optimisation

Convergance – optimisation algorithm is said to converge to a solution.

Convex – means global min was found.

Non-convex – means a local min was found which cannot be escaped.

Guarantees of convergence – most algorithms are not guaranteed to converge even if a solution exists.

Tuning optimisation – use the right algorithm.

What can go wrong?

Slow progress. Noisy and diverging performance. Getting stuck. Plateaus. Local minima. Saddle points. Very steep or discontinuous objective functions.

Common Problems

Slow progress: typically from steps that are too small (tiny δ or neighborhood)

Noisy/diverging performance: steps too large causing bouncing or falling into "the abyss"

Divergence may require constraints to limit the feasible set

Adaptive Local Search

The size of the neighborhood can be adapted. If there is no improvement after a number of iterations, the size of random steps can be increased.

Multiple Restarts

To avoid getting stuck in a bad local minima, several random initial guesses can be used.

Temperature

Temperature refers to how the rate of movement in the parameter space changes over the course of an optimization.

Population

Involves tracking multiple simultaneous parameter configurations and selecting or mixing them. This approach uses mutation (random variation), natural selection (solution selection), and breeding (interchanging solutions).

A solution's genotype refers to its parameter set. Each iteration slightly moves solutions by random mutation, culls the weakest solutions, and copies the remaining "fittest" solutions to keep the population size constant.

Breeding involves creating combinations of the fittest solutions rather than simply copying them. This approach works well when the parameter can be partitioned into distinct components, allowing offspring to inherit good qualities from both parents.

Pros

It is easy to understand and applies to many problems. It requires only weak knowledge of the objective function and can be applied to both discrete and continuous components.

The method offers some robustness against local minima, although it can be hard to control.

It has great flexibility in parameterisation, including mutation schemes, crossover schemes, fitness functions, and selection functions.

Cons

There are many hyperparameters to tune, which radically affect the performance of the optimization. There is no guarantee of convergence, and it is ad hoc. The algorithm is often slow compared to using stronger knowledge of the objective function, and many evaluations of the objective function are required, one per population member per iteration.

Algorithm Selection

For least-squares problems: use specialized solvers or pseudo-inverse

For convex problems: use convex solvers (much more efficient)

If derivatives are available: use first-order or second-order methods

If none of the above: use zeroth-order solvers (simulated annealing, genetic algorithms)

Simulated Annealing

Simulated annealing extends hill climbing by sometimes going uphill instead of always moving downhill. It uses a temperature schedule, allowing more uphill steps at the start of the optimization and fewer later in the process to avoid getting stuck in a local minima.

It accepts that sometimes it is necessary to go uphill. A proportion of jumps that temporarily increase the loss are randomly accepted, and this proportion is gradually reduced over time.

Pros

It is much less sensitive to getting trapped in minima than hill climbing. Simulated annealing is easy to implement and empirically very effective.

Cons

It depends on a good choice for the temperature schedule and neighborhood function, which are extra free parameters to worry about. There are no guarantees of convergence, and it can be slow if uphill steps are not needed.

Memory

Memory refers to recording good or bad steps in the past and revisiting or avoiding them. Specifically, it involves remembering good steps in the solution space.

When combined with a population, memory leads to stigmergy, where a trace left in the environment by an action stimulates the performance of a subset of actions.

Pros

It can be very effective in spaces where good solutions are separated by large, narrow valleys. It can use fewer evaluations of the objective function than a generic algorithm if pheromones (a form of memory) are effective. When it works, it really works.

Cons

It is a moderately complex algorithm to implement. There is no guarantee of convergence, and it is ad hoc. It has even more hyperparameters to tune than genetic algorithms.

Convergence in Optimization

Convergence means finding a solution (global minimum in convex problems, local minimum in non-convex)

Good algorithms converge quickly with steep drops in the objective function

Bad algorithms may wander forever or diverge to infinity

Convergence often depends on initial conditions

Some algorithms guarantee convergence if a solution exists

Heuristic methods (like random search) offer no convergence guarantees

Plotting objective function value vs. iteration helps diagnose convergence issues

Ideal convergence shows rapid decrease in loss function value

Tuning Optimization

Optimization converts specific problems into ones solvable by general algorithms

Optimizers have hyperparameters that affect search behavior and require tuning

Unit 7: Numerical nonlinear optimisation

Example: deep neural networks (aka deep learning)
Uses:
speech recognition
machine translation
image classification
image synthesis
Problem: finding an approximating function
In a simple model:
$\theta^* = \text{argmin}_\theta \sum i \, ||f(xi; \theta) - yi||$
Where:
x – observations y - outputs
θ* - distance between f and yi (expected output)
Backpropagation:
Traditional neural network consists of a series of layers each of which is a linear map followed by a simple, fixed nonlinear function (rotate, stretch (linear map), fold (simple nonlinear folding))
Weight map - matrix specifying the linear map in each layer, all of the entries of these matrices can be seen as parameter vector θ
Nonlinear function G(x) is fixed for every layer and cannot vary, usually it "squashes" the output in some way
yi = G(Wixi + bi)
Where: W - linear map x - input of previous layer
Backpropagation - algorithm for automatic differentiation
Advantage - we can tell how much of an effect each weight will have on the prediction for every weight, in the whole network, in one go

Optimisation with derivatives
If we know (or can compute) the gradient, we can get:
The direction of fastest increase
The steepness of that slope
First-order optimisation
Differentiability
Smooth function – has continuous derivatives up to some order, usually easier to do iterative optimisation on
Cn continuous function – nth derivative is continuous
First-order optimisation uses the (first) derivatives of the objective function with respect to the parameters. Can only be applied if the function is:
C1 continuous
Differentiable – gradient is defined almost everywhere

Why not heuristic search?
Can be very slow
No guarantee of convergence or progress
Huge number of hyperparameters to change
Rolling a ball: physical optimisation
Imagine a ball rolling on a (smooth) surface which represents the value of the objective function across a 2D domain
The ball will eventually settle in a configuration where there is a balanced set of forces applied to it
Attractors: flowing towards a solution
The trajectory of the search is parallel to the "flow field" of the objective function
Can find the minima by moving towards the "basins" by following these flow lines
Notation:
f(x) -> function of x
f'(x) -> first derivative of f
f''(x) -> second derivative
We can collect this derivative information into a matrix called the **Jacobian** matrix, which characterises the slope at a specific point x. If the input $x \in \mathbb{R}^n$ and the output $y \in \mathbb{R}^m$, then we have an m × n matrix.

$$f'(\mathbf{x}) = \mathbf{J} = \begin{bmatrix} \frac{\partial y_0}{\partial x_0} & \frac{\partial y_0}{\partial x_1} & \cdots & \frac{\partial y_0}{\partial x_n} \\ \frac{\partial y_1}{\partial x_0} & \frac{\partial y_1}{\partial x_1} & \cdots & \frac{\partial y_1}{\partial x_n} \\ \cdots & & & \\ \frac{\partial y_m}{\partial x_0} & \frac{\partial y_m}{\partial x_1} & \cdots & \frac{\partial y_m}{\partial x_n} \end{bmatrix}$$

Lipschitz continuity (no ankle breaking)
For $\mathbb{R}^n \rightarrow \mathbb{R}$: gradient is bounded and the function cannot change more quickly than some constant
There is a maximum steepness:
K for all i and some fixed K

$$\frac{\partial L(\theta)}{\partial t} \leq K$$

Lipschitz constant
Imagine running a cone of particular steepness across a surface; we can check if it ever touches the surface
Lipschitz constant K is a measure of how wide this cone that only touches the function once is,

$$K = \sup \left[\frac{|f(x) - f(y)|}{|x - y|} \right],$$

where sup is the supremum (smallest value that is larger than every value of the function)
Smaller K means a smoother function
K = 0 is totally flat

Gradient and Hessian
∇f(x) → gradient vector of a scalar function of a vector (first derivative for vector functions), has one (partial) derivative per component of x. One row of the Jacobian

$$\nabla L(\theta) = \left[\frac{\partial L(\theta)}{\partial \theta_1}, \frac{\partial L(\theta)}{\partial \theta_2}, \ldots, \frac{\partial L(\theta)}{\partial \theta_n}, \right]$$

∇²f(x) → Hessian matrix of a (scalar) function of a vector, the equivalent of the second derivative for vector functions: Jacobian of the gradient vector

$$H(L) = \nabla\nabla L(\theta) = \nabla^2 L(\theta) = \begin{bmatrix} \frac{\partial^2 L(\theta)}{\partial \theta_1^2} & \frac{\partial^2 L(\theta)}{\partial \theta_1 \partial \theta_2} & \frac{\partial^2 L(\theta)}{\partial \theta_1 \partial \theta_3} & \cdots & \frac{\partial^2 L(\theta)}{\partial \theta_1 \partial \theta_n} \\ \frac{\partial^2 L(\theta)}{\partial \theta_2 \partial \theta_1} & \frac{\partial^2 L(\theta)}{\partial \theta_2^2} & \frac{\partial^2 L(\theta)}{\partial \theta_2 \partial \theta_3} & \cdots & \frac{\partial^2 L(\theta)}{\partial \theta_2 \partial \theta_n} \\ \cdots & & & & \\ \frac{\partial^2 L(\theta)}{\partial \theta_n \partial \theta_1} & \frac{\partial^2 L(\theta)}{\partial \theta_n \partial \theta_2} & \frac{\partial^2 L(\theta)}{\partial \theta_n \partial \theta_3} & \cdots & \frac{\partial^2 L(\theta)}{\partial \theta_n^2} \end{bmatrix}$$

Difference in objective functions
For some objective functions, we can compute the exact derivatives with respect to the parameters θ
Example: L(θ) = θ² → L'(θ) = 2θ
If we know the derivatives, we can use this to move in "good directions" – down the slope of the objective function towards a minimum
More complicated if there is a gradient vector instead of a simple scalar derivative
Orders:
Zeroth order – only requires evaluation of L(θ)
First order – evaluate L(θ) and its derivative ∇L(θ)
Second order – evaluate L(θ), ∇L(θ), and ∇∇L(θ); these methods include quasi-Newtonian optimisation

Gradient: A derivative vector

$$\nabla L(\theta) = \left[\frac{\partial L(\theta)}{\partial \theta_1}, \frac{\partial L(\theta)}{\partial \theta_2}, \ldots, \frac{\partial L(\theta)}{\partial \theta_n}, \right]$$

At any given point, the gradient of a function points in the direction where the function increases fastest
Magnitude is the rate at which the function is changing ("the steepness")
Gradient descent
Basic first-order algorithm
$\theta^{(i+1)} = \theta^i - \partial \nabla L(\theta i)$
Where ∂ is the step size (scaling parameter), which can be fixed or chosen adaptively
Steps:
Start somewhere θ_0
Repeat:
Check how steep the ground is in each direction $v = \nabla L(\theta^i)$
Move a little step ∂ in the steepest direction v to find θ^{i+1}
Downhill is not always the shortest route
However, it is generally much faster than blindly bouncing around hoping to get to the bottom

Why Step Size Matters

Too small step size (δ) leads to very small steps and slow convergence

Too large step size causes unpredictable optimisation behaviour

Unpredictability occurs when the gradient function changes significantly within a step

Example: gradient changing sign across a step

Finding optimal step size is crucial for efficient optimisation

Step size δ is a critical hyperparameter that determines convergence versus divergence

Optimal δ is directly related to the Lipschitz constant K of the objective function

K is rarely known precisely in real-world optimisation problems

Gradient Descent in 2D

This technique extends to any number of dimensions, as long as we can get the gradient vector at any point in the parameter space

We just have a gradient vector ∇L(θ) instead of a simple 1D derivative

There is no change to the code

How Do We Get Derivatives?

First-order optimisation requires available derivatives of the objective function

Not applicable to pure empirical optimisation without computational models

Building models enables gradient-based optimisation approaches

Three ways to obtain gradients:

Symbolic/analytical – using computer algebra software. Less flexible and awkward in practice

Numerical (finite differences) – easy but extremely slow for high-dimensional parameters and numerically unstable

Automatic Differentiation

Most flexible and efficient method

Allows gradient computation at any point "for free"

Preferred method in practice

If we have analytical derivatives (i.e. we know the derivative of the function in closed form; we can write down the maths directly), you will probably remember the "high school" process for mathematical optimisation

Why Not Use Numerical Differences?

$$\frac{d}{dx} f(x) = \lim_{h \to 0} \frac{f(x+h) - f(x-h)}{2h}$$

Given this definition, why do we need to know the true derivative ∇L(θ) if we can evaluate L(θ) anywhere? Why not just evaluate L(θ + h) and L(θ − h) for some small h?

This approach is called numerical differentiation, and these are finite differences

This works fine for reasonably smooth one-dimensional functions

Numerical Problems

Choosing appropriate step size h is challenging

Finite differences method violates key rules for accurate floating point calculations:

Magnitude error – adds small h to potentially much larger x

Cancellation error – subtracts very similar values f(x + h) and f(x − h)

Error magnification – divides result by very small value 2h

These combined numerical problems make finite differences particularly unstable

Methods for selecting step size

Fixed value approach – often determined through trial and error

In machine learning, called "learning rate" and found via grid/random search

Scheduled step sizes that decrease over iterations (annealing):

Exponential decay: $\delta = \delta_0 \cdot \exp(-\lambda \cdot i)$

Inverse decay: $\delta = \delta_0 / (1 + \lambda \cdot i)$

Some schedules occasionally increase step size to "reheat" optimisation

All methods introduce additional hyperparameters to tune

There are various adaptive schemes which adjust the step size algorithmically. They can adjust the step size globally (across all dimensions) or across each dimension independently.

Examples include:

Line search – choose the step size by starting with a big step and shrinking it if the gradient we get does not match the actual drop in the objective function

Adagrad – adapt the step size for each dimension based on the history of the gradient in that dimension

RMSProp – a variant of Adagrad that scales the step size by the root mean square of the gradient

There are many other adaptive schemes, but the basic idea is to adjust the step size based on the behaviour of the gradient

Revenge of the Curse of Dimensionality

Curse of dimensionality strikes again with numerical differentiation

For each gradient evaluation at point x:

Must compute differences in each dimension

For 1 million dimensions: 2 million evaluations of L(θ)

Time complexity increases by factor of O(n) over function evaluation

Typically requires $O(n^2)$ computations for gradient

This enormous computational overhead makes numerical differentiation impractical for high-dimensional problems

Even with infinite precision arithmetic, computational cost would be prohibitive

Automatic Differentiation

Automatic differentiation (autodiff) can derive exact derivatives for functions written in a programming language subset

Makes first-order optimisation feasible for functions of any dimension

Key Advantages

Creates gradient functions (grad_L(θ)) that compute exact derivatives at any point

No numerical issues like those in finite differences

Scales efficiently to billions of parameters

Time complexity is only a constant factor of evaluating the function itself

Line Search Algorithm

Set the step size δ to some relatively large value

Compute the gradient vector ∇L(θ)

Separate this gradient vector into a direction and length:

Normalise this vector to produce the (unit) direction of steepest descent:

$d = \nabla L(\theta) / \|\nabla L(\theta)\|_2$

Compute the length of the gradient:

$l = \|\nabla L(\theta)\|_2$

Test the Armijo–Goldstein condition, which checks if the actual decrease in the objective function is approximately what we'd expect given the local gradient:

$L(\theta - \delta \cdot d) \leq L(\theta) + \delta \cdot c \cdot l$

If it passes, we stop

Otherwise, we reduce δ by a factor τ and repeat the test

τ and c are parameters chosen to make the line search work well (hyperparameters). They are typically values like 0.5, 0.5; the exact values chosen usually don't make a huge difference

Revenge of the Curse of Dimensionality

Curse of dimensionality strikes again with numerical differentiation

For each gradient evaluation at point x:

Must compute differences in each dimension

For 1 million dimensions: 2 million evaluations of L(θ)

Time complexity increases by factor of O(n) over function evaluation

Typically requires $O(n^2)$ computations for gradient

This enormous computational overhead makes numerical differentiation impractical for high-dimensional problems

Even with infinite precision arithmetic, computational cost would be prohibitive

Automatic Differentiation

Automatic differentiation (autodiff) can derive exact derivatives for functions written in a programming language subset

Makes first-order optimisation feasible for functions of any dimension

Key Advantages

Creates gradient functions (grad_L(θ)) that compute exact derivatives at any point

No numerical issues like those in finite differences

Scales efficiently to billions of parameters

Time complexity is only a constant factor of evaluating the function itself

6.1.1. Example

This is how we might find the minimum of

$$f(x) = x^4 - 40x^2 - 100x.$$

The derivative is:

$$f'(x) = 4x^3 - 80x - 100$$

and the second derivative is

$$f''(x) = 12x^3 - 80.$$

We can solve for:

$$f'(x) = 4x^3 - 20x - 100 = 0,$$

and check the sign of

$$f''(x) = 12x^2 - 20$$

to find if we have a minimum.

Two Main Types

Forward mode – gives one column of Jacobian (useful for few inputs, many outputs)

Uses dual numbers concept (similar to complex numbers with derivative part)

Referred to as jvp mode (Jacobian-vector product)

Backward mode – gives one row of Jacobian (useful for many inputs, few outputs)

Used in deep learning software for gradient vectors

Requires storing intermediate computation results

Uses chain rule for derivatives

Referred to as vjp mode (vector-Jacobian product)

Implementation Approaches

Most packages (autograd, JAX) trace execution and store intermediates

Some (like Google Tangent) use source-to-source translation

Programming Advances Powering Data Science

Vectorised programming (NumPy, Eigen) – operations over tensors with GPU acceleration

Differentiable programming (autograd, JAX, PyTorch) – automatic differentiation of tensor algorithms

Probabilistic programming (PyMC, Stan) – handles uncertain values and random variables

Stochastic Relaxation

Sometimes optimisation involves binary-valued functions, such as "maths gets eaten or not"

While each specific case is binary and discontinuous, we can average over many random instances where the surrounding conditions are slightly different

This averaging is called stochastic relaxation

It smooths out a jagged, impossibly steep gradient

The result is an approximately Lipschitz continuous function

This is achieved by integrating across many randomised conditions

Stochastic Gradient Descent (SGD)

Objective function is broken down into small parts, and the optimizer can do gradient descent on randomly selected parts independently. Works if the objective function can be written as a sum: $L(\theta) = \sum i\ Li(\theta)$

i.e., the objective function is composed of the sum of many simple sub-objective functions L1(θ), L2(θ), …, Ln(θ). This occurs when matching parameters to observations (approximation problems). Many training examples xi with matching outputs yi. Want to find a parameter vector θ such that the difference between expected output and model output is minimized, summed over all results.

$$L(\theta) = \sum_i ||f(\mathbf{x_i}; \theta) - y_i||$$

Can interchange summation, scalar multiplication, and differentiation:

$$\nabla \sum_i ||f(\mathbf{x_i}; \theta) - y_i|| = \sum_i \nabla ||f(\mathbf{x_i}; \theta) - y_i||$$

Take any subset of training samples and outputs. Take the gradient for each sample. Move according to the computed gradient of the subset (called minibatch). Over time, the random subset selection will (hopefully) average out.

AutoGrad

autograd provides automatic differentiation for virtually any NumPy code. The example below is from the autograd documentation. It is a drop-in replacement which just "magically" estimates derivatives (although only some operations are supported).

Autograd has now evolved into **Google JAX**, probably the most promising machine learning library. JAX supports GPU and TPU computation with automatic differentiation.

We're not using it here due to installation complexity.

```
import autograd.numpy as np
from autograd import grad, elementwise_grad

# just plain numpy
def tanh(x):  # Define a function
    y = np.exp(-x)
    return (1.0 - y) / (1.0 + y)


# compute gradient
grad_tanh = grad(tanh)  # Obtain its gradient function
print(grad_tanh(1.0))  # Evaluate the gradient at x = 1.0
```

Using AutoGrad in Optimisation

Automatic differentiation provides derivatives "for free" from straightforward objective function computations

It makes first-order optimisation extremely efficient

It is at the core of modern machine learning libraries

It allows writing vectorised, differentiable code that can run on GPU/TPU hardware

A fundamental advantage is that it separates the description of the objective function from the optimisation process

Example: Line-fitting optimisation

This involves finding parameters *m* (slope) and *c* (intercept) for the line of best fit

It minimises the square difference between the line and the data points

It uses automatic differentiation to compute gradients efficiently

It eliminates the need for manual derivative calculation

It is significantly simpler than approaches requiring explicit derivatives

Heuristic enhancement

Random partitioning in each minibatch adds noise, which seems like a problem, but in fact, can help jump over a maxima.

Using SGD

No guarantee that SGD will even move in the right direction. Can be very efficient for many real-world problems.

Linear Regression with SGD

Can do a problem with 10,000 test points as the data is divided into lots of sums of smaller problems.

A nightmare function

Narrow valleys everywhere. Multiple local minima. A huge ridge through the middle. Gradient descent gets stuck in a valley and never even approaches a local minima.

Limits of Automatic Differentiation

Differentiation only works for differentiable functions

First-order gradients (Jacobian) are computationally efficient

First-order derivatives never require more than about three times the computation of the original function

Second-order derivatives (Hessian) can require up to the square of the number of dimensions in computation time

This scaling explains why second-order methods are rarely used in practice

High-dimensional problems make second-order methods particularly expensive

Improving Gradient Descent

Gradient descent is more efficient than zeroth-order methods, but it has limitations

Local minima problems

Gradient descent can get stuck in local minima

This happens in non-convex functions or with poor step sizes

Possible solutions include random restarts and momentum-based optimisation

Limited to smooth functions

Gradient descent only works if the objective function is smooth and differentiable

A solution is stochastic relaxation, which introduces randomness to help with steep or discontinuous functions

Slow evaluation

Gradient descent can be slow if the objective function or its gradient is expensive to compute

Stochastic Gradient Descent (SGD) improves speed by breaking optimisation into smaller subproblems

Random Restart

Run gradient descent until it gets stuck, then randomly restart with different initial conditions and try again. Repeat a few times, hopefully one of them ends up in the global minima.

Simple Memory: Momentum Terms

Rather than having ant-colony style paths everywhere, the optimizer just remembers one simple path — the way it is currently going. Premise: if you are going the right way now, keep going that way even if the gradient is not always quite downhill. Produce a velocity and have the optimizer move in that direction and gradually adjust v to align with the derivative:

$v = \propto v + \partial \nabla L(\theta)$

$\theta(i+1) = \theta i - v$

Types of Critical Points

Points where the gradient vector is the zero vector:

Maxima: Gradient decreases in all dimensions.

Minima.

Saddle Points: Gradient has different signs in different dimensions.

Plateau: Gradient 0 in all dimensions.

Ridge: Gradient 0 in one dimension.

Second-order Derivatives
First-order derivatives express the “slope” of a function. Second-order derivatives show the “curvature” of a function. For every parameter component θi, the Hessian stores how the steepness of every other θj changes.
Eigenvalues of the Hessian
If all are positive, the matrix is positive definite and the point is a minimum.
If all are negative, the matrix is negative definite and the point is a maximum.
Mixed, the point is a saddle point.
Positive/negative but with zeros, the matrix is semidefinite and the point is a plateau/ridge.
Second-order Optimisation
Jump to the bottom of each local quadratic approximation in a single step.
Curse of dimensionality: too much memory is used to store high dimensions.

$$H(L) = \nabla\nabla L(\theta) = \nabla^2 L(\theta) = \begin{bmatrix} \frac{\partial^2 L(\theta)}{\partial \theta_1^2} & \frac{\partial^2 L(\theta)}{\partial \theta_1 \partial \theta_2} & \frac{\partial^2 L(\theta)}{\partial \theta_1 \partial \theta_3} & \cdots & \frac{\partial^2 L(\theta)}{\partial \theta_1 \partial \theta_n} \\ \frac{\partial^2 L(\theta)}{\partial \theta_2 \partial \theta_1} & \frac{\partial^2 L(\theta)}{\partial \theta_2^2} & \frac{\partial^2 L(\theta)}{\partial \theta_2 \partial \theta_3} & \cdots & \frac{\partial^2 L(\theta)}{\partial \theta_2 \partial \theta_n} \\ \cdots & & & & \\ \frac{\partial^2 L(\theta)}{\partial \theta_n \partial \theta_1} & \frac{\partial^2 L(\theta)}{\partial \theta_n \partial \theta_2} & \frac{\partial^2 L(\theta)}{\partial \theta_n \partial \theta_3} & \cdots & \frac{\partial^2 L(\theta)}{\partial \theta_n^2} \end{bmatrix}$$

8.4. What calculus do I need to know?

If you don't know this in detail then "Matrix calculus explained" from the DF(H) Resources on Moodle is a good explanation: https://explained.ai/matrix-calculus/index.html.

- The definition of a derivative

$$\frac{df(x)}{dx} = \lim_{h \to 0} \frac{f(x+h) - f(x-h)}{2h}$$

- How to write partial derivatives

$$\frac{\partial f(\mathbf{x})}{dx_0}$$

is the derivative of a function taking a *vector* $\mathbf{x}$ with respect to the x_0 component.

$$\nabla f(X) = \left[\frac{\partial f(\mathbf{X})}{dx_0}, \frac{\partial f(\mathbf{X})}{dx_1}, \ldots \right]$$

is all of the partial derivatives combined into a **gradient vector**

- The linearity of derivatives

$$\frac{d[f(x) + g(x)]}{dx} = \frac{df(x)}{dx} + \frac{dg(x)}{dx}.$$

and

$$\frac{d[\lambda f(x)]}{dx} = \lambda \frac{df(x)}{dx}$$

and so:

$$\frac{d\left[\sum_i \lambda_i f_i(x)\right]}{dx} = \sum_i \lambda_i \frac{df_i(x)}{dx}.$$

(i.e. the weighted sum of derivatives is the derivative of the weighted sum)

Unit 8 Probability and Random Variables

Probability
Probability theory is a simple, consistent, and effective way to manipulate uncertainty.

Experiment (or trial)
An occurrence with an uncertain outcome. For example, losing a submarine — the location of the submarine is now unknown.

Outcome
The result of an experiment; one particular state of the world. For example, the submarine is in ocean grid square [2,3].

Sample Space
The set of all possible outcomes for an experiment. For example, ocean grid squares {[0,0], [0,1], [0,2], [0,3], ..., [8,7], [8,8], [8,9], [9,9]}.

Event
A subset of possible outcomes with some common property. For example, the grid squares which are south of the Equator.

Probability
The probability of an event with respect to a sample space is the number of outcomes from the sample space that are in the event, divided by the total number of outcomes in the sample space. Since it is a ratio, probability will always be a real number between 0 (representing an impossible event) and 1 (representing a certain event). For example, the probability of the submarine being below the equator, or the probability of the submarine being in grid square [0,0] (in this case the event is just a single outcome).

Formal Basis for Probability Theory

Axioms of Probability

Boundness — All possible outcomes A satisfy $0 \leq P(A) \leq 1$.

Unitarity — For the complete set of possible outcomes $A \in \sigma$ in a sample space σ, something always happens: $\sum P(A) = 1$.

Sum Rule — The probability of either outcome A or B happening: $P(A \vee B) = P(A) + P(B) - P(A \wedge B)$.

Conditional Probability — The probability that outcome A will happen given that we know B has happened:
$P(A|B) = P(A \wedge B) / P(B)$

Random Variables and Distributions
A random variable can take on different values, but we don't know what value it has — only the possible states the variable could take on.
Distributions define how likely different states of a random variable are.
$P(X = x)$ represents the probability of random variable X taking on value x.

Distribution of Discrete Variables
Described using a probability mass function (PMF): $P(X = x) = fX(x)$

Distribution of Continuous Variables
Described using a probability density function (PDF), which spreads the probability over outcomes as a continuous function. The PDF must integrate to unity (i.e. total probability sums to 1).

Probability Distribution
A mapping of outcomes to probabilities that sum to 1. This is because an outcome must happen from a trial (with probability 1) so the sum of all possible outcomes together will be 1. A random variable has a probability distribution which maps each outcome to a probability. For example $P(X = x)$, the probability that the submarine is in a specific grid square x.

Random Variable
A variable representing an unknown value, whose probability distribution we do know. The variable is associated with outcomes of a trial. For example, X is a random variable representing the location of the submarine.

Probability Density/Mass Function
A function that defines a probability distribution by mapping each outcome to a probability $fX(x)$, $x \to \mathbb{R}^+$. This could be a continuous function over x (density) or discrete function over x (mass). For example, $fX(x)$ would be a probability mass function for the submarine, which maps each grid square to a real number representing its probability.

Observation
An outcome that we have directly observed, i.e. data. For example, a submarine was found in grid square [0,5].

Likelihood
How likely an observation is given a probability distribution. For example, the likelihood of finding a submarine in grid square [0,5] if the submarine was distributed according to some pattern. Note: this coincides with probability for discrete distributions, but it is a different concept and is different for continuous distributions.

Expectation
The expected value of a random variable X is denoted $\mathbb{E}[X]$ and defined as:
$\mathbb{E}[X] = \int_x x\, fX(x)\, dx$ for continuous random variables.
For a discrete random variable with PMF $P(X = x) = fX(x)$, and finite possibilities:
$\mathbb{E}[X] = P(X = x_1)x_1 + P(X = x_2)x_2 + \ldots + P(X = xn)xn$

Expectation and Means
The expected value of a random variable is the true average of the value of all outcomes — the population mean.
The central tendency is given by the mean of a random variable X: $\mathbb{E}[X]$
The spread is described by the variance of a random variable X: $var(X) = \mathbb{E}[(X - \mathbb{E}[X])^2]$

Expectation of a Function g(X)
For a continuous random variable X, the expected value of any function g(X) is:
$\mathbb{E}[g(X)] = \int x\, fX(x)\, g(x)\, dx$
For discrete variables, it becomes a weighted sum over values of g(x) weighted by $P(X = x)$

$$\mathbb{E}[g(X)] = \sum_x f_X(x) g(x) dx$$

$$E[f(X)] \neq f(E[X])$$

Sample
An outcome that we have simulated according to a probability distribution. We say we have drawn a sample from a distribution. For example, if we believe that the submarine was distributed according to some pattern, generate possible concrete grid positions that follow this pattern.

Expectation / Expected Value
The "average" value of a random variable. The submarine was on average in grid square [3.46, 2.19]. A probability distribution is defined by a probability density/mass function $fX(x)$ which assigns probabilities to outcomes such that the sum of probabilities over all outcomes is 1, $\sum fX(x) = 1$.

Philosophy of Probability

Bayesian/Laplacian View on Probability
This is the calculus of belief, where probabilities measure degrees of belief.
Example with data:
Prior belief — it's Glasgow, so it's not likely to be sunny.
New evidence — there seems to be a bright reflection inside.
Update belief — calculate the posterior, our new probability that it is sunny outside.
This view explicitly quantifies our assumptions with probability distributions. It is considered subjective.

Frequentist View of Probability
Focuses only on the long-term behaviour of repeated events.
Makes statements about universal truths.
Does not include a calculus of belief and therefore cannot answer many questions.
It is considered objective.

Bayesian
Includes priors.
Probability is a degree of belief.
Parameters of the population are considered to be random variables, data is known.

Frequentist
No priors.
Probability is the long-term frequency of events.
Parameters of the population are assumed to be fixed, and data is considered random.
Probability is considered superior to other representations of uncertainty.

Generative Process
The idea that there is some unknown process going on, the results of which can be observed.
There are unobserved variables that are not known but can be inferred.
Forward probability refers to questions relating to the distribution of the observations.
Inverse probability refers to questions related to unobserved variables that govern the process that generated the observation.
Unknowable refers to questions that cannot be resolved with current observations.

Samples and Sampling
Samples are observed outcomes of an experiment (also known as observations).
Sampling is the process of simulating outcomes according to the probability distribution of those variables.

Empirical Distribution
An estimate of the probability mass function that might be generating observations. If nx is the number of times outcome x was observed, and N is the total number of trials, then the empirical probability of x is $P(X = x) = \frac{n_x}{N}$.

Random Sampling Procedures
Uniform Sampling — A uniformly distributed number has equal probability of taking on any value in its interval (a, b), and zero everywhere else: X ~ U(a, b)
Discrete Sampling — Sampling according to any arbitrary PMF by partitioning the unit interval:
Choose any arbitrary ordering for the outcomes.
Assign a "bin" (portion of the interval [0,1]) to each outcome so that the entire region is divided into consecutive non-overlapping sections.
Draw a uniform sample in the range [0,1].
Whichever "bin" the sample falls into determines the outcome to draw.

Writing and Manipulation of Probabilities
The odds of an event with probability p are given by the ratio p / (1 − p).

$$\text{odds}_{\text{against}} = \frac{1-p}{p} \quad \text{odds}_{\text{for}} = \frac{p}{1-p}$$

Log-odds (logit) are useful for very unlikely scenarios and are calculated as:
logit(p) = log(p / (1 − p))
You can convert logits l back to probabilities using the inverse logit function.

$$\text{prob}(g) = \frac{g}{1+e^g}$$

Log probabilities are often used when computing the probability of multiple independent random variables taking on a set of values, to avoid numerical underflow.

$$\log P(x_1, \ldots, x_n) = \sum_{i=1}^{n} \log P(x_i)$$

When talking about likelihood, we often write L(xi) to mean the likelihood of xi. The likelihood is not a probability; it is a function of the data.

$$\mathcal{L}(x_i) = f_X(x_i) \quad \mathcal{L}(c_1, c_2, \ldots c_n) = \prod_i P(c_i)$$

Bayes' Rule
Bayes' Rule is used to invert conditional distributions:
P(A|B) = (P(B|A) P(A)) / P(B)
Posterior is what we want to know: P(A|B)
Likelihood is how likely the event A is to produce the evidence B: P(B|A)
Prior is how likely A is, regardless of any new evidence: P(A)
Evidence is how likely B is, regardless of whether A happened: P(B)
This is often phrased in terms of a hypothesis H we want to evaluate given some data D, and we write Bayes' Rule as:
P(H|D) = (P(D|H) P(H)) / P(D)

The Wall and Dart Analogy
Think of a wall that's exactly 1 unit wide. You're going to divide this wall into sections, with each section representing a possible outcome. The width of each section is proportional to the probability of that outcome.
For example, if you have outcomes A, B, and C with probabilities 0.2, 0.5, and 0.3:
Outcome A gets the first 0.2 portion of the wall (from 0 to 0.2)
Outcome B gets the next 0.5 portion (from 0.2 to 0.7)
Outcome C gets the final 0.3 portion (from 0.7 to 1.0)
Now, to sample from this distribution, you throw a dart randomly at the wall (generate a random number between 0 and 1), and whichever section it lands in determines your sample.

Continuous Sampling
Continuous sampling involves writing the problem in terms of transformations of continuous values, as you can't pick discrete elements of the set.
If the inverse cumulative distribution function (ICDF) has an explicit form:
Uniformly sample from the unit interval Y ~ U(0, 1)
Use the inverse cumulative distribution function $cX^{-1}(x)$ to transform the result back to the original domain.

Integration Over the Evidence
To find the posterior, we need to compute the evidence.
For a discrete set of outcomes:
P(D) = ∑i P(D|Hi) P(Hi)
For continuous outcomes:
P(D) = ∫A P(D|H) P(H) dA

Binary Cases
P(H = 1|D) = (P(D|H = 1) P(H = 1)) / [P(D|H = 1) P(H = 1) + P(D|H = 0) P(H = 0)]

Natural Frequency
This involves imagining concrete populations of a fixed size and considering the proportions of the population as counts.

Bayes' Rule
Bayes' Rule is the correct way to combine prior belief and observation to update our beliefs.

Integration Over the Evidence
To find the posterior, we need to compute the evidence.
For a discrete set of outcomes:
P(D) = ∑i P(D|Hi) P(Hi)
For continuous outcomes:
P(D) = ∫A P(D|H) P(H) dA

Binary Cases

$$P(H = 1|D) = \frac{P(D|H=1)P(H=1)}{P(D|H=1)P(H=1) + P(D|H=0)P(H=0)}$$

Natural Frequency
This involves imagining concrete populations of a fixed size and considering the proportions of the population as counts.

Bayes' Rule
Bayes' Rule is the correct way to combine prior belief and observation to update our beliefs.

Joint Probability
The joint probability is the probability of two values appearing simultaneously:
P(X, Y) = P(X = x) ∧ P(Y = y)

Marginal Probability
Marginal probability is the derivation of P(X) from the joint probability P(X, Y) by integrating or summing over all possible outcomes of Y.

$$P(X) = \int_y P(X, Y)dy \text{ for a PDF.}$$

$$P(X) = \sum_y P(X, Y) \text{ for a PMF.}$$

Marginalisation refers to this process of integration over one or more variables from a joint distribution, effectively removing them.

Conditional Probability
Conditional probability expresses how likely the outcomes of X are if we know (or fix) the outcomes of Y:
P(X|Y) = P(X, Y) / P(Y)

Bigrams
A bigram model considers pairs of adjacent characters (ci−1, ci) as they appear in sequence within a text.

Components of a Character Bigram Model
Joint Distribution P(Ci = ci, Ci−1 = ci−1): This is the normalised count of each specific character pair occurrence.
Marginal Distribution P(Ci = ci): Found by summing the joint distribution over all possible preceding characters.
Marginal Distribution P(Ci−1 = ci−1): Found by summing the joint distribution over all possible following characters.
Conditional Distribution P(Ci = ci | Ci−1 = ci−1): This is the probability of observing character ci given the previous character was ci−1.

Calculation of Conditional Probability
P(Ci = ci | Ci−1 = ci−1) = P(Ci = ci, Ci−1 = ci−1) / P(Ci−1 = ci−1)
This represents the likelihood of a character following another specific character.

Applications
Bigram models are used to model character transitions in text, enabling the prediction of likely next characters in a sequence.

Generalisation
Bigram models are part of a broader family of n-gram models (such as unigram, trigram, etc.) that can be applied to characters or words.

Entropy
The entropy of a discrete distribution of a random variable can be computed as:

$$H(X) = \sum_x -P(X = x)\log_2(P(X = x))$$

This is just the expected value of the log-probability of a random variable – the "average" log-probability.

Unit 9: Sampling and Inference

Continuous Random Variables
Problems:
The probability of any specific value is zero, yet any value in the support of the distribution function (non-zero) is possible.
There is no direct way to sample from the PDF in the same way as PMF.
Bayes' Rule is easy to apply to discrete problems but not continuous.
Simple discrete distributions don't have a concept of dimension.

Probability Distribution Functions
The value of a PDF at any point is not a probability.
The probability of a continuous random variable X lying in range (a, b) is:

$$P(X \in (a,b)) = (a < X < b) = \int_a^b f_X(x)$$

Support
Domain the PDF maps from where the density is non-zero.

$$\text{supp}(x) = x \text{ such that } f_X(x) > 0$$

Inference
Process of determining parameters of the function generating data by looking at the aftermath (samples and observations).

Two Worldviews
Bayesian Inference - Parameters are random, data is fixed.
Frequentist Inference - Parameters are fixed, data is random.

Three Approaches
Direct Estimation - Define functions of observations that will estimate the values of parameters of distributions directly.
Maximum Likelihood Estimation - Use optimisation to find parameter settings that make the observations appear as likely as possible.
Bayesian, Probabilistic Approaches - Assume a distribution over the parameters themselves and consider the parameters to be random variables. Use observations to update some initial belief to hone the estimate of the parameters to a tighter distribution.

Direct Estimation
Estimators are computed via statistics (e.g., mean, variance).

Population Mean
"True" average.

Variance and Standard Deviation
Sample standard deviation:
Measures the spread of a collection of values.
Covariance matrix Σ generalises it to higher dimensions.

Compact Support Distributions
Have non-zero density only over a finite interval.
Examples include the uniform distribution.
Samples are strictly bounded within a specific range.
The probability of getting a value outside this range is exactly zero.
The PDF "cuts off" sharply at the boundaries.

Infinite Support Distributions
Have non-zero density across the entire real line (or some infinite subset).
Examples include the normal (Gaussian) distribution.
Samples can theoretically take any value within their domain.
Extreme values are possible but increasingly unlikely.
The PDF approaches but never actually reaches zero as you move away from the center.
For practical applications, this distinction matters when:
Setting bounds for algorithms.
Determining whether extreme values are possible.
Modeling physical quantities with natural limits.
Analyzing the behavior of statistical methods under different assumptions.

Linear Regression
Linear regression is the fitting of a line to observed data.
This passage describes the classic linear regression problem:
We're modeling data with a line equation y = mx + c (or y = Ax + b in higher dimensions).
The true relationship includes random noise: y = mx + c + ε.
We assume the noise follows a normal distribution: ε ~ N(0, σ²).
This makes our full model: y ~ N(mx + c, σ²).
The key elements are:
x values are known inputs (not random).
y values are random variables influenced by both the line and noise.
m, c, and σ are parameters we want to estimate.
We observe data points [(x_1, y_1), (x_2, y_2), ..., (x_n, y_n)].
The noise term serves two important purposes:

1. It accounts for variations that our simple line can't explain.
2. It acknowledges that real-world data is rarely perfectly linear.

This normal (Gaussian) noise assumption is popular because:
It makes mathematical analysis tractable.
It often matches real-world noise patterns.
It connects to the central limit theorem.
The goal is to estimate the best values for m, c, and possibly σ² from the observed data.

$$\hat{\mu} = \frac{1}{N}\sum_{i=1}^{N} x_i$$

Cumulative Distribution Functions
(always maps to the codomain [0, 1]) tells us how much probability mass there is that is less than or equal to x.

Distribution to Continuous Random Variables
Random variable X is distributed as Normal with mean μ and variance σ²: X ~ N(μ, σ²).
Location - (most dense around) μ, scale - (how spread out is it) σ².
Central Limit Theorem - if we form a sum of many random variables, then for almost any PDF that each may have, the total will be approximately normal.

Multivariate Distributions
Distributions over $\mathbb{R}^n$.
$\int fX(x) = 1, x \in \mathbb{R}^n$.

Multivariate Uniform
Assigns equal probability to some (axis-aligned) box in a vector space $\mathbb{R}^n$, such that:
$\int fX(x) = 1, x \in a\ box\ x$.

Transformed Uniform Distribution (Over Any Box)
Transform the vectors with a matrix A and shift by adding an offset b.

$$\mathbf{x}' = A\mathbf{x} + \mathbf{b}$$

Normal Distribution
Fully specified by a mean vector μ and a covariance matrix Σ.
Joint probability density - density over all dimensions.
Marginal probability density - density over some sub-selection of dimensions.
Matrix "covers" the data with an ellipsoidal shape.
For example, consider X ~ N(μ, Σ), X ∈ $\mathbb{R}^2$, a two dimensional ("bivariate") normal distribution. We can look at some examples of the PDF, showing:

- Joint P(X)
- Marginal P(X_1) and P(X_2)
- Conditionals P(X_1|X_2) and P(X_2|X_1)

Standard Estimators
Sample Mean - Good estimator of the true population mean:
Measures the central tendency of a collection of values.
The population mean is μ = E[X] for a random variable X.
The mean vector generalises it to higher dimensions.
The sample mean (μ^) is an unbiased estimator of the population mean (μ).
"Unbiased" means that the estimator doesn't systematically overestimate or underestimate the true parameter.
The sample mean is calculated as the arithmetic mean of all observed samples.
As sample size increases, the estimate μ^ becomes more accurate.
This illustrates the Law of Large Numbers - with more data, sample statistics converge to true population parameters.
The notation μ^ (with a "hat") indicates an estimate of the true parameter μ.
This principle is fundamental to statistical inference - using observable data to estimate unknown parameters.

Variance and Standard Deviation

Sample standard deviation:

Measures the spread of a collection of values.

Covariance matrix Σ generalises it to higher dimensions.

$$\hat{\Sigma} = \frac{1}{N}\sum_{i=1}^{N}(x_i - \hat{\mu})(x_i - \hat{\mu})^T$$

The sample variance is the squared difference of each value of a sequence from the mean of that sequence:

$\sigma^2 = (1/N)\ \Sigma(i=1 \text{ to } N)\ (x_i - \mu)^2$

It is an estimator of the population variance, $E[(X - E[X])^2]$

The sample standard deviation is just the square root of this value.

$\sigma = \sqrt{[(1/N)\ \Sigma(i=1 \text{ to } N)\ (x_i - \mu)^2]}$

Linear Regression via Direct Estimation

Linear regression parameters [m, c] can be estimated using the pseudo-inverse method from linear algebra.

This approach assumes that the noise (σ) follows a normal distribution.

The method is called "ordinary linear least-squares."

The problem is formulated as minimizing $L(\theta) = \|f(x; \theta) - y\|$.

Where:

θ = [m, c] are the parameters we want to estimate.

$f(x; \theta) = \theta_0 x + \theta_1$ is our linear model function.

y represents the observed data points.

When using squared Euclidean norm, we're minimizing the sum of squared differences between model predictions and actual observations.

This direct estimation method provides parameters in one computation, without iterative optimization.

The approach finds the line that minimizes the sum of squared vertical distances between the observed data points and the fitted line.

If we choose the squared Euclidean norm, then we have for the y = mx + c case:

$$L(\theta) = \|f(x;\theta) - y\|$$

$$L(\theta) = \|\theta_0 x + \theta_1 - y\|_2^2 = (\theta_0 x + \theta_1 - y)^2,$$

Fitting

Estimating the parameters of a normal distribution to a set of observations.

Maximum Likelihood Estimation: Estimation by Optimisation

If there is no fixed, closed-form function to estimate the parameters, use a likelihood function to apply optimization to work out a parameter setting under which the observed data was most likely.

Maximum Likelihood Estimation

Finds the best setting of parameters that would explain how the observations came to be.

If likelihood depends on some parameters of a distribution θ, then we write L(θ|x).

Define an objective function: $\theta^* = \text{argmin}_\theta\ L(\theta)$.

$$L(\theta) = -\log \mathcal{L}(\theta|x_1, \ldots, x_n) = -\sum_i \log f_X(x_i; \theta),$$

Bayesian Inference

Don't seek to find the most likely parameter setting but to infer a distribution over possible parameter settings compatible with the data.

Infer a posterior distribution, given prior belief (P(D)), evidence (P(θ)), and likelihood function P(D|θ):

$$P(\theta|D) = \frac{P(D|\theta)P(\theta)}{P(D)}$$

Inference

Conjugate Priors

Prior and posterior are of the same form.

Allows closed-form solution.

Rare but convenient (e.g., normal prior + normal likelihood = normal posterior).

Variational Inference

Approximates posterior by optimisation.

Finds the "closest" distribution to the true posterior.

Computationally expensive.

Requires specific forms for prior and likelihood.

Provides a true distribution, but it's an approximation.

Monte Carlo Methods

Draws samples from the posterior.

Most general and widely applicable.

Very computationally expensive.

We will use this method for posterior sampling.

P(θ|D)

P(θ|D) needs to be computed for a distribution over θ, not just some numbers.

$P(D) = \int_\theta P(D|\theta)P(\theta)$ which is likely intractable.

Ways to Make it Tractable

Samples will do.

Relative probability only:

$P(\theta|D) \propto P(D|\theta)P(\theta)$.

Applies to continuous and discrete distributions, but computations in closed form are much harder.

Linear Regression via Maximum Likelihood Estimation

The problem can be framed using the distribution of random variables.

Assume errors are normally distributed and corrupt a perfect relationship.

The model assumes:

The true relationship has a mean.

Errors have a standard deviation.

This leads to a maximum likelihood estimation (MLE) problem.

To avoid numerical underflow, use the log-likelihood instead of the likelihood.

For independent samples, the log-likelihood is used to describe the model.

The goal is to minimise the negative log-likelihood, which finds the most likely parameters.

When noise is normally distributed, MLE for linear regression is:

Equivalent to least-squares optimisation.

But also estimates the standard deviation of the noise, in addition to the regression parameters.

This method is called maximum likelihood linear regression.

To perform MLE, we must:

Compute the likelihood for each sample.

Take the product (or sum of log-likelihoods) across all samples.

This forms the objective function for optimisation.

By minimising the negative log-likelihood, we search for the parameter values that make the observed data most likely.

Efficient optimisation is key:

Heuristic methods exist but are often computationally expensive.

Gradient descent is much more efficient.

$$u(\theta) = \nabla_\theta \log \mathcal{L}(\theta|[x_1, x_2, \ldots\ x_n]).$$

The score function is:

The gradient of the log-likelihood.

Indicates how to adjust parameters to increase the likelihood.

We use the score function in gradient descent to update parameters iteratively using ordinary gradient descent.

In linear regression, the parameters are typically just β (regression coefficients).

While linear regression has analytical solutions, this gradient-based approach is general and works for any model with a likelihood function.

Markov Chain Monte Carlo

Walks through the space of parameter settings, proposing small random tweaks to the parameter settings, and accepting "jumps" if they make the estimate more likely.

Good - No need for complex approximations or limit models.

Bad - Choice of sampling strategy has a very large influence on the kind of sample runs that are practical to execute.

Metropolis Sampling

Uses proposal distribution Q(θ'|θ) (analogous to neighbourhood function in optimisation).

Metropolis-Hastings

Wanders around in the distribution space, accepting jumps to new positions using Q(θ'|θ) to randomly sample the space of P(θ|D).

Take current position θ and propose a new position θ' that is a random sample drawn from Q(θ'|θ).

$$P(\text{accept}) = \begin{cases} f_X(\theta')/f_X(\theta), & f_X(\theta) > f_X(\theta') \\ 1, & f_X(\theta) \leq f_X(\theta') \end{cases}$$

Metropolis-Hastings Algorithm

Initial Conditions

fX(θ) - PDF of the distribution to sample from.

q(θ) - Function that draws a random sample, distributed near θ.

θ_0 - First guess for θ.

p = $fX(\theta_0)$.

Algorithm

Draws a sample θ' from the prior distribution.

For n iterations:

a. Proposal: Compute θ' = $q(\theta_i)$.

b. Density of proposal: Compute p' = fX(θ').

c. Acceptance probability: Compute pr = p'/p.

d. Generate a uniform random number r in [0,1].

e. Accept/reject if r < pr: p = p' and θ_i = θ'.

Trace

History of accepted samples of MCMC.

Posterior Distribution of the Model Parameters

The values we expect the model parameters to take on given the data we observed and our prior.

Predictive Posterior

Distribution over observations we would expect to see.

For n repetitions:

Draw samples from the posterior distribution over parameters to give us a concrete distribution.

For m repetitions, draw samples from this concrete distribution.

Inference: A Review

Linear Regression

Fitting of a line to observed data (assumes the function generates data where one of the observed variables is a scaled and shifted version of another).

Scaling - Gradient m, shifting - offset c → y = mx + c.

Noise - Assume its normally distributed variations in our measurements:

$y = mx + c + N(0, \sigma^2) \rightarrow y \sim N(mx + c, \sigma^2)$.

y is random, x is known, and the rest we want to infer.

Linear Regression via Direct Optimisation

Function optimisation:

L(θ) = ||f(x; θ) - y||

$f(x; \theta) = \theta_0 x + \theta_1$.

For squared Euclidean form and y = mx + c case:

$L(\theta) = ||\theta_0 x + \theta_1 - y||^2_2 = (\theta_0 x + \theta_1 - y)^2$.

Linear Regression via Maximum Likelihood Estimation

Write down the problem in terms of the distribution of random variables.

Assume "errors" are normally distributed values, corrupting a perfect y = mx + c relationship:

$Y \sim N(mx + c, \sigma^2)$ where mx + c → mean, σ → standard deviation.

To avoid underflow, work with the log of the likelihood.

Bayesian Linear Regression

Derive a belief about the parameters as a probability distribution.

Time series and signals
Real world signals are continuous in both value and time/space: x(t)
They need to be sampled to make them amenable to digital signal processing (turn a representation of a function to a numerical array)
Quantise (force to a discrete set of values) the value and time/space
Sample the continuous signal regularly — make measurements with precisely fixed time intervals between them (time quantisation)
Each measurement is quantized to a set of values so that it can be represented as a fixed-length number in memory, e.g. int8 (amplitude quantization)

Sampled sequences: throwing away time
Sampled signal is stored as a single vector with the assumption that time starts at some origin and increases by a fixed amount for every successive element
Store the sampling rate fs (in Hz) — how frequently the data has been sampled
ΔT = 1/fs (spacing between samples in seconds)

Sampling theory
If we sample a signal frequently enough, we can perfectly reconstruct any given continuous signal
Digital signal processing: manipulating waves as ndarrays with confidence that the operations are meaningful in terms of the original signals
Nyquist limit — where fs is the sampling rate
We can recover an original time signal x(t) perfectly if it contains frequencies of at most the

Nyquist limit

$$f_n = \frac{f_s}{2}$$

For example, audio is often recorded with a sample rate of 44.1 KHz, because human hearing extends to about 20 KHz

Aliasing
Observing an artificial component in the sampled signal
Happens when the Nyquist limit is not followed: fq > fn
Artificial component = fn - (fq mod fn)
This creates phantom behaviour in the sampled signals, which doesn't correspond to any real-world changes in the continuous signal value
In video this shows up as the wagon wheel effect — objects seem to stop and reverse. This occurs precisely when the frequency of movement exceeds half the frame rate of the video/film

Why sample signals
Efficient representation: Approximates continuous signals as arrays, saving storage space
Enables computation: Allows digital processing of analogue signals (e.g. sound, images)
Supports array operations: Signal manipulations map to simple array operations:
Offset removal → subtract a value elementwise
Mixing signals → weighted elementwise sum
Correlation → elementwise multiplication
Region selection → array slicing
Smoothing/regression → apply on arrays
Compact & powerful: Enables fast and flexible processing of signal data

Noise
Signal has two components: y(t) — real value, ε(t) — random fluctuation
x(t) = y(t) + ε(t)
Signal to noise ratio
SNR = S/N, where S = amplitude of y(t), N = amplitude of ε(t)
Usually represented in decibels:
SNRdB = 10 log10(S/N)

Images
Before reducing image resolution, high-frequency details must be filtered out
If not, aliasing occurs — unwanted low-frequency patterns appear, making the image look distorted or messy

Audio
Aliasing in audio is clearly heard when signals are poorly sampled
In music synthesis, creating realistic sounds without aliasing is challenging and a key issue in digital sound design

Definition of linearity
f(x + y) = f(x) + f(y)
f(ax) = af(x)
f(0) = 0

Nonlinear filtering — any filter which is not a weighted sum

Median filtering
If most measurements are good, but a small minority may be arbitrarily corrupted
Like a moving average but uses a median to estimate the central tendency of a window

Order filters
Like median filters but use other order statistics such as maximum, minimum, and percentile filtering
This adds significant robustness to extreme values, as is always the benefit of the median over the mean
Median filters can be slow to compute, requiring sort operations or specialised median cascade algorithms

Generalising moving averages — convolution and linear filters

Linear filters — any filter where the output is a weighted sum of neighbouring values (and the original values)
Useful as computers often have a multiply-and-accumulate instruction, making implementing linear filters very easy
Example: f(x[t]) = 0.25x[t - 1] + 0.5x[t] + 0.25x[t + 1] in a 1D time series

Removing noise
ε(t) is random, so can't be subtracted from all values, but assumptions about y(t) can be made (e.g. no rapid changes)
For example, we might guess that wheat prices don't change very quickly — they stay fairly steady from year to year. So if we see sudden big changes in the signal, we can assume those fast changes are probably noise, not real changes in the price

Sampling: amplitude quantization
Makes f(t) discrete, by reducing it to a number of distinct values
Introduces noise to signals
Can plot the residual (difference) of a quantized signal from the original — shows the amount of error introduced
However, coarser quantisation requires less storage space, less precise circuitry, lower memory bandwidth and less computation time
The hardware which transforms analogue to digital signals (the ADC) will always have a limited quantisation capability; cheap hardware might quantise to 8 bits

Filtering and smoothing
Temporal structure — real signals cannot have arbitrary changes; at least some portion is predictable
Can average out the contribution of the noise by averaging over multiple time steps — this will also change the true signal slightly

Moving averages
Take a sliding window of samples and compute their mean
Break the signal into equally spaced chunks or windows of a common length K
Process these as an N × K data matrix — N windows of K samples
Uses a window of fixed length to smooth a signal
The output has fewer samples because there isn't enough data at the start and end to compute full averages
We can apply moving averages to a sound. This lowpass filters the sound and reduces high frequency content. The longer the moving average, the smoother the waveform and the more high-frequencies are suppressed
Moving averages are linear filters because their output is a weighted sum of previous inputs

Convolution — process of taking weighted sums of neighbouring values
x ∗ y for two vectors or matrices (also works for functions)

$$(x * y)[n] = \sum_{m=-M}^{M} x[n-m]y[m]$$

x is the signal to be transformed, y is the operation to be performed (convolution kernel)
y is typically much smaller than x

Algebraic properties of convolution
f * g = g * f
f * (g * h) = (f * g) * h
Because convolution is linear, convolution also distributes over addition (and thus also subtraction)

$$f * (g + h) = f * g + f * h$$

Simplest convolutions: Dirac delta functions

$$\int_{-\infty}^{\infty} \delta(x) = 1$$

$$\delta(x) = \begin{cases} 0, x \neq 0 \\ 1, x = 0 \end{cases}$$

Tool for analysing system responses
It is a function that is zero everywhere except at 0, where it is 1
Convolution of any function with the delta function does not change the original function
Impulse — an array of zeros with a single 1
By feeding a perfect impulse into a system we want to model, we can recover the convolution kernel — this is known as linear system identification
Reverberation — summation of lots of tiny echoes from the surroundings
Delta function

$$f(x) * \delta(x) = f(x)$$

The delta function is an identity element in convolution, meaning convolving any function with the delta function leaves the original function unchanged
Implication — stimulating a system with the delta function allows us to recover the system's convolution kernel, which represents its linear behaviour (this is called linear system identification)
In the discrete world, the delta function becomes an array with a single 1 (an impulse). Feeding this impulse into a system can help us model the system
Practical difficulties
Producing a perfect impulse (without distortion or blurring)
Dealing with noise in measurements, which complicates real-world system identification
In digital systems, these challenges are less of an issue since we can use noise-free impulses, but in physical systems, they are significant obstacles

Fast Fourier Transform (FFT)
The fast Fourier algorithm runs in O(N log N) if N is a power of 2
It is $O(N^2)$ for non-power-of-2 inputs. Variations of the FFT run in O(N log N) for vectors with length which is highly composite, but still $O(N^2)$ for prime-length vectors
Convolution theorem
The Fourier transform of the convolution of two signals is equal to the element-wise product of the Fourier transforms of those two signals
FT(f(x) ∗ g(x)) = FT(f(x)) FT(g(x)) where FT is the Fourier Transform
f(x) ∗ g(x) = IFT [FT(f(x)) FT(g(x))] where IFT is the Inverse Fourier Transform
Frequency domain effects of filtering
Filter types
Smoothing/lowpass filter reduces high frequencies
Highpass filter reduces low frequencies
Bandpass filter reduces frequencies outside of a certain band
Notch/bandstop filter reduces frequencies inside a certain band
Linear filter
A linear filter can change the amplitude of frequency components, but it can never introduce new frequencies
Proof: FT(f ∗ g) = FT(f) FT(g), so no matter what we make g, if any element of FT(f) is zero, it will always stay zero

Frequency domain
Signals can be seen as:
A sequence of amplitude measurements over time — time domain
A sum of oscillations with different frequencies — frequency domain
Pure oscillation is a sine wave
$x(t) = A \sin(2\pi\ \omega t + \theta)$
ω — frequency of oscillation
θ — phase (offset to time) of oscillation
A — magnitude of oscillation
A pure frequency is a sine wave with a specific period
Fourier transform
Any repeating (continuous) function can be decomposed into sine waves
Write every signal as a sum of sinusoidal functions
A sine wave has a single unique frequency, amplitude, and phase

$$x(t) = \sum_i A_i \sin(\omega_i 2\pi t + \theta_i)$$

Correlation between signals
This is the (unnormalised) correlation between two signals, or just the inner product of their representation as vectors
c = ∑ a[t]b[t] (summed over t)
If two signals are unrelated, c ≈ 0
If a[t] and b[t] are close, c will be large and positive
If a[t] and b[t] are inverses (negatives) of each other, c will be large and negative

Frequency
The frequency of the component is given by the k index and depends on the sampling rate
X_0 = 0 and $X_{n/2} = f_n$, where f_n is the Nyquist rate (half the original sampling rate)

freq = $f_N k/N$

Spectra
Magnitude spectrum: shows the amplitude of each frequency component
Phase spectrum: shows the phase of each frequency component
Logarithmic scale (dB): used for magnitude spectrum to better visualise relative amplitudes
Wrapped phase scale: used for phase spectrum to clearly show phase relationships
Main lobe: central band where most signal energy is concentrated
Side lobes: smaller energy bands at other frequencies
Side lobe level: ratio of side lobe energy to main lobe energy
Side lobes can be introduced by operations and are considered a form of distortion
Computational cost
DFT is expensive to compute: $O(N^2)$

Finding the phase of a sine wave
To find how shifted the sine wave should be to line up best, compare the sine wave and cosine wave and work out the phase
α[t] = sin(ωx)
α′[t] = cos(ωx) = sin(ωx + π/2)
The phase is defined as the angle of the vector formed of these two values
c(ω) represents the correlation value between the test signal b[t] and a sine wave of frequency ω (specifically sin(ωt))
c′(ω) represents the correlation value between the test signal b[t] and a cosine wave of the same frequency ω (specifically cos(ωt) or sin(ωt + π/2))

$$\theta = \tan^{-1}\left(\frac{c(\omega)}{c'(\omega)}\right)$$

Fourier transform: complex output

$$\hat{f}(\omega) = \int_{-\infty}^{\infty} f(x) e^{-2\pi i x \omega} dx$$

e^(2πiθ) = cos(2πθ) + i sin(2πθ)
For real signals, we can deal with only the real part of the complex output

$$\hat{f}(\omega)_{\text{real}} = \int_{-\infty}^{\infty} f(x)\cos(-2\pi x\omega)dx$$

$$\hat{f}(\omega)_{\text{imag}} = \int_{-\infty}^{\infty} f(x)\sin(-2\pi x\omega)dx$$

$$\hat{f}(\omega) = \int_{-\infty}^{\infty} f(x)\sin(-2\pi x\omega)dx + i\int_{-\infty}^{\infty} f(x)\cos(-2\pi x\omega)$$

Inversion — reconstructing the sinusoids into the original function

$$f(x) = \int_{-\infty}^{\infty} \hat{f}(\omega) e^{2\pi i x \omega} d\xi$$

The Fourier transform is just another way of looking at a function. Instead of considering a value which varies in amplitude over time, we consider a value which varies in amplitude and phase over frequencies
Discrete Fourier transform (DFT)
Works on discrete measurements

$$F[k] = \sum_{j=0}^{N-1} x[j] e^{-2\pi i \frac{j}{N}}$$

$$= \sum_{j=0}^{N-1} \left[x[j]\cos\left(-2\pi\frac{j}{N}\right) + ix[j]\sin\left(-2\pi\frac{j}{N}\right)\right]$$

For the k frequency components of x (k = 0, 1, ..., N − 1). The DFT has as many frequency components as x[t] has elements
Complex components: phase and magnitude
F[k] = a + bi
Any complex number can be written as a magnitude A and angle θ (phase)