Revision

General guidance

- Out of 60
- 2 questions from 3 [30 marks each]
- 90 minutes + 30 minutes
- Submit a PDF; use Word/Jupyter/Latex/whatever.
- Open book. Keep the lecture notes open in tabs!
- You don't ever need to have working code. Don't spend time trying to get things perfect.
- Fine to have a notebook or IDE open and test out ideas, but watch your time.

> Would there be any coding in the questions? If so, would it be tested with any program?

Yes, there might well be -- **but I am are not looking for working code** -- just that you have the right approach (e.g. as you would do in a whiteboard interview question). Syntax and precise details aren't important.

> Any advice on choosing which questions to answer?

- I'd read through and just aim for the ones that you think you are most confident in. There aren't easy or hard questions in general. Some questions will have more or less coding, which might be easier or harder depending on your strengths.

> Do you recommend typing or handwriting the exam?

- **DO NOT HANDWRITE THE EXAM!** Type it. If you need to draw something, take a photo and insert it. That's only likely if you need to insert an equation.

> Was 2019 harder than other exams?

- Yes, because students had a full 24 hours to answer, open book. Note that marks were slightly *higher* that year than in the past.

> Which exam is most indicative of the style of questions?

- 2020 probably the closest similar paper.

> Will we have to draw something?

No, you won't in DF(H).

> What happens if I answer all three questions?

We mark the first two we encounter.

Numpy

- What are IEEE754 special numbers?
    - -0.0 and 0.0 (-0.0 == 0.0)
    - -inf and inf (-inf + x = -inf, inf + x = inf, etc. BUT inf-inf or inf+-inf = NaN)
    - NaN (0/0, inf-inf, sqrt(-1), etc.)

- When might IEEE exceptions occur?
    - Overflow (big number * big number or big+big) -> inf / -inf
    - Underflow (small number * small number) -> 0.0
    - Invalid operation (0/0, inf-inf, etc.) -> NaN
    - Div. by zero (x/0, x!=0) -> inf / -inf
    - Inexact (don't worry about this one)

- What is the format of an IEEE754 float64
    - 1 bit sign, 11 bits exponent, 52 bits mantissa = 64 bits

- What is the mantissa?
    - A number between 1.0 and 2.0, represented as 1.xxxxxx in binary
    - The leading 1 isn't stored, because it's always the same

- What is the exponent?
    - A power of 2 (like 2, 8, 256) that the mantissa is treated as being scaled by.

- How do I do a computation like what byte is element [2,3] of a [9, 13] `float64` array:
    - float64 = 8 bytes (float32 = 4 bytes)
    - strides are therefore [8*13, 8] = [104, 8]
        - (note: last dimension will have smallest stride if C/row-major ordered)
    - first byte of element [2, 3] = 2 * 104 + 3 * 8 = 232 bytes after the start of the array.

- What is row-major ordering?
    - It's the default ordering of arrays in memory for NumPy, where the last indexing dimension "changes first".
    - The opposite is Fortan/column-major, where the first indexing dimension changes first.
    - It's just a convention, but it affects how data is ordered in memory.

- Do I need to know `einsum`?
    - Yes, but you're not likely to have a hard question about in an exam (too hard to test).

- How do you add something to every row of an array? Or in a higher-d tensor?

    - Example: I have a x, a [16, 10, 5] tensor. I want to add 1 to [:, 0, :], 2 to [:, 1, :], etc.
    - **Broadcasting only works on the last dimensions**
    - Switch the ordering, broadcast, switch back.
    - `np.arange(1, 11)` = [1,2,3,4,5,6,7,8,10]

    ```
    a = np.einsum('ijk -> ikj', x) # swap
    a = a + np.arange(1, 11) # broadcast
    a = np.einsum('ikj -> ijk', a) # swap back
    ```

Visualisation

Criticism questions

- If you're asked to criticise and fix; *make sure you actually suggest both the problem and the solution!*.

- Common issues in bad graphs:
    - Missing labels
    - Indistinguishable geoms (no colour)
    - Missing legends
    - Axes switched
    - **No uncertainty where there ought to be**
    - Line when should be step or scatter (and vice versa)
    - Layered when should be faceted
    - Wacky colour schemes or bad glyph shapes/sizes

Themes

- Do I need to know violin plots and KDEs?
    - **NO.**

- Do I need to know Box plots?
    - Yes.

- Do I need to know the layered grammar of graphics?
    - Yes. Know the words and be able to link them to plots.

Lin. alg.

- What's a norm? Like L1 norm or L2 norm?
    - It's just a particular kind of length -- lengths of vectors are thing we have to define.
    - x = [1, 2, -3]
    - L1 norm = sum of abs. components L1(x) = 1 + 2 + 3 = 6
    - L2 norm = square root of sum of squares. L2(x) = 1^2 + 2^2 + 3^2 = sqrt(14)
    - L_∞ norm = largest (abs.) element of x Linf(x) = 3
    - $L_{-\infty}$ norm = smallest (abs.) element of x Linf(x) = 1

- What's the relation between distance and norm?
    - The distance between two vectors a and b under a specific norm is just norm(a-b) (length of the distance).

- What is a matrix?
    - A compact way of writing certain functions as a block of numbers.
    - Specifically, every **linear function** on real vectors can be written as a matrix.
    - It's also a 2D array (that's the "encoding" of the function)

- What is a linear function?
    - Symbolically:
        - f(a) + f(b) = f(a+b)
        - f(c * a) = c * f(a) (c is a scalar)
    - Geometrically:
        - A linear function on real vectors is a rotation and/or a scaling *
        - E.g. scale all vectors by 2, rotate by 45 degrees, or combinations, which are *skews* (rotate-scale-rotate)
    - Every function which just scales and rotates is linear and vice versa. Any function that is not just scale and rotate is *not a linear function*.
    - Every real matrix defines a linear function on real vectors.

- There are lots of properties of matrices, but I forgot some of them...
    - Ax applies a matrix A to a vector x
    - AB is the multiplication of matrix A and B, and *composes* the effect of the functions.
    - A^T is the matrix transpose, and exchanges rows and columns.
    - A square matrix:
        - Has a determinant det(A) = product of eigenvalues. If det(A) is 0, the matrix collapses one or more dimensions, and *can't be inverted*.
        - Has a trace tr(A) = sum of eigenvalues = sum of diagonal.
        - Has eigenvalues and eigenvectors. An eigenvector is just a vector that only gets scaled as it goes through the function (direction does not change).
            - Eigenvalue is *how much* it gets scaled.
        - **May** have an inverse. A^{-1}, which exists if det(A)!=0 (or every dimension is preserved, or every eigenvalue is non-zero).
            - Inverse *undoes* the function A.
        - AA = A^2 = repeating the effect of the function twice
            - AAA = A^3, etc.
        - If it is diagonal (every entry is zero except the diagonal)
            - Then it is a pure scaling operation (scales each dimension by the corresponding diagonal)
            - Can be inverted by reversing the scaling (i.e. 1/each diagonal element)

        - The diagonal matrix with all ones is the *identity matrix* I and, since it justs scales each dimension by 1... does nothing at all.

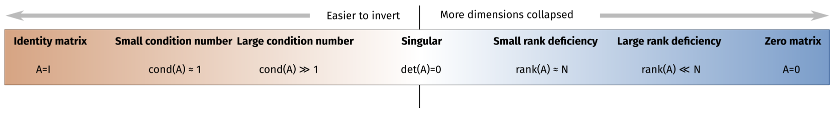
- How could I compute an eigenvector?
    - Just choose a random vector, put it through the matrix A, scale it to be length 1 ("normalise") and put through matrix A, scale it to be length 1, ...
    - until it stops changing. The resulting vector is the *leading eigenvector* and is the eigenvector associated with the largest eigenvalue.
    - (which you can compute by putting the eigenvector through the matrix and seeing how much bigger/smaller it gets).
    - You don't need to know how to compute more eigenvectors (other than numpy will compute them for you).

- Why would I want to?
    - Eigenvectors and eigenvalues are "characteristics" of a matrix (that's what "eigen" means) -- they characterise the effect of the matrix.
    - E.g. all eigenvalues = 1 matrix does no scaling
    - E.g. some eigenvalues = 0 matrix cannot be inverted
    - Eigenvectors of a covariance matrix are the *principal components*
        - These are the axes of the ellipse that "describes the data"
        - More properly, the axes of the multivariate normal that would best approximate the data.

- What's the SVD?
    - The **singular value decomposition** is the standard way to do many operations on matrices when implementing algorithms.
    - It takes *any* matrix A and returns U, Σ, V, such that

      $A = U\Sigma V^T$ *and*
        - U is a square, pure rotation matrix
        - \Sigma is diagonal, pure scaling matrix; the diag. elements are called the **singular values**
        - V^T a square, pure rotation matrix.
    - E.g. if U and V are identity matrices, then A was a scaling operation (no rotation can have happened)
    - If \Sigma is identity, then A performs no scaling
    - The **rank** of a matrix is the number of dimensions preserved = number of non-zero singular values = number of non-zero eigenvalues (for square matrices).
        - =number of rows/columns if A is invertible ("non-singular")
        - < number of rows/columns if not
        - The zero matrix has zero rank

- The **condition number** of a matrix is the ratio of biggest and smallest singular value.
    - Infinite = not invertible
    - Large = numerically unstable
    - Small = numerically stable.



```
* Explain what the rank of a matrix represents? L
ectures gave off multiple meanings which confused
 me.
    * In linear algebra: rank of *a matrix* = the
 number of (vector space) dimensions preserved by
 the transform.
    * In numerical computation: rank of *an array*
or a tensor = the number of indexing dimensions.
    * (yes this is confusing, and yes this is the
 standard terminology)
```

For 2020 3b could we have used np.linspace(from_actor, to_actor, k)

In [2]:
```
# Yes, you could.

from_actor = [1,2,3,4]
to_actor = [-1, 0, 3, -4]
np.linspace(from_actor, to_actor, 10)
```

executed in 7ms, finished 14:52:39 2022-05-02

```
array([[ 1.        ,  2.        ,  3.        ,  4.        ],
       [ 0.77777778,  1.77777778,  3.        ,  3.11111111],
       [ 0.55555556,  1.55555556,  3.        ,  2.22222222],
       [ 0.33333333,  1.33333333,  3.        ,  1.33333333],
       [ 0.11111111,  1.11111111,  3.        ,  0.44444444],
       [-0.11111111,  0.88888889,  3.        , -0.44444444],
       [-0.33333333,  0.66666667,  3.        , -1.33333333],
       [-0.55555556,  0.44444444,  3.        , -2.22222222],
       [-0.77777778,  0.22222222,  3.        , -3.11111111],
       [-1.        ,  0.        ,  3.        , -4.        ]])
```

```
In [4]:
a = np.array([1, 2, 3])
b = np.array([0, 5, 6])

t_0 = 50 # a
t_1 = 800 # b

t = 95 # c

alpha = (t - t_0) / (t_1 - t_0) # proportion
c = (1-alpha) * a + alpha * b #

c
```

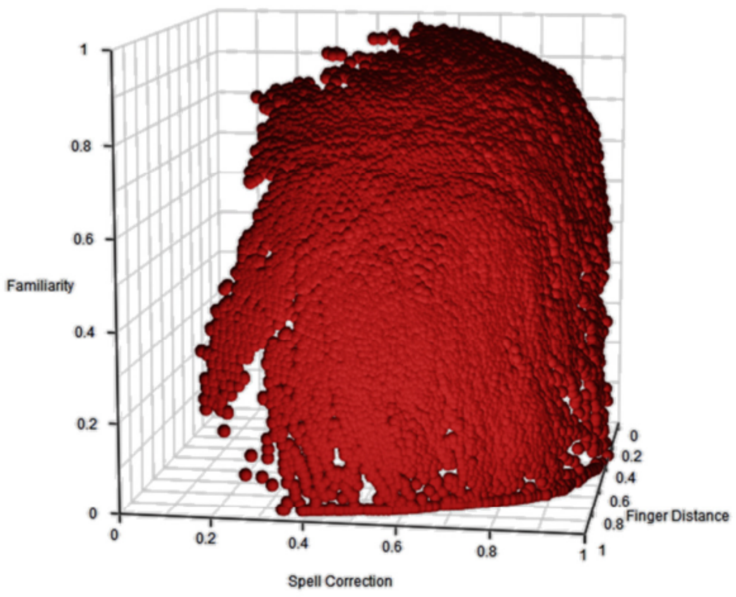
executed in 13ms, finished 12:44:21 2022-05-03

```
array([0.94, 2.18, 3.18])
```

Optimisation

- What do I need to look out for in an optimisation question?
    - What are the **parameters** -- what can we adjust to make things better/worse?
    - What's the **objective function** -- maps parameters to a single measure of "goodness"?
    - Are there any **constraints** -- parameter settings that are impossible/should be avoided?
    - What do we know about the implementation of the objective function?
        - Is it smooth/continuous? If not, we need something like grid/random search.
        - Is it differentiable? If so, we can use gradient descent; if not, something like genetic algorithms or hill-climbing.
        - Is it a sum of simpler functions? If so, and we're using gradient descent, we can use stochastic gradient descent.
        - Is it convex? If so, there are fast algorithms to solve it (you don't need to know more than this).

- Do I need to know Pareto optimisation?
    - No, it's not in the course any more.



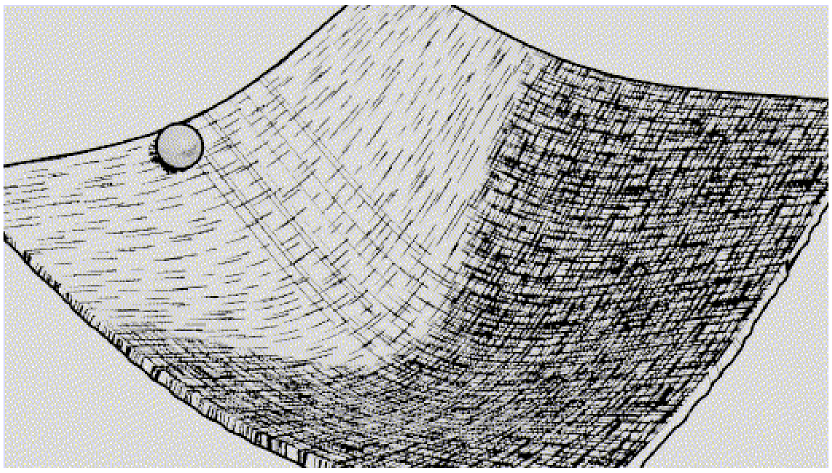
- How can you tell if an objective function can be split into sub-objective functions? As well as this, how can you tell data can be split.
  - If you've been told it is possible or impossible! For example, if you've been told $L(x) = L_1(x) + L_2(x) + L_3(x)$ this is clearly a sum of sub-objective functions.
  - If you've been told the data can't be split into batches (e.g. because it represents a signal that can't logically be split) then it is not possible.
  - You don't need to deduce these properties; they will be stated in some form.

- What's the "approximation" form of an objective function?
  - $L(\theta) = ||f(x;\theta) - y||$
    - the distance
    - between a function applied to x (which is a given input)
    - with parameters θ (which we can change)
    - and some some given output y
  - we tweak θ until $f(x;\theta) \approx y$.

- How do I use constraints in an optimisation?
  - Generally, the answer to this will be "add a penalty term to the objective function" -- that is a value that gets larger as the solution gets worse

- What is gradient descent?
  - An algorithm that takes the *local* gradient of an objective function at a specific parameter setting (point) and makes a *small* step in that direction.
  - It requires a way of computing the gradient at any given parameter setting (point)

  - Usually, this is performed by automatic differentiation and gives exact values everywhere
  - Numerical approximations are impractically slow and subject to numerical issues...



- Do I need to know the equation for gradient descent?
  - Yes, but if you forget it, just look it up.

$$\theta_{t+1} = \theta_t - \delta L'(\theta_t)$$

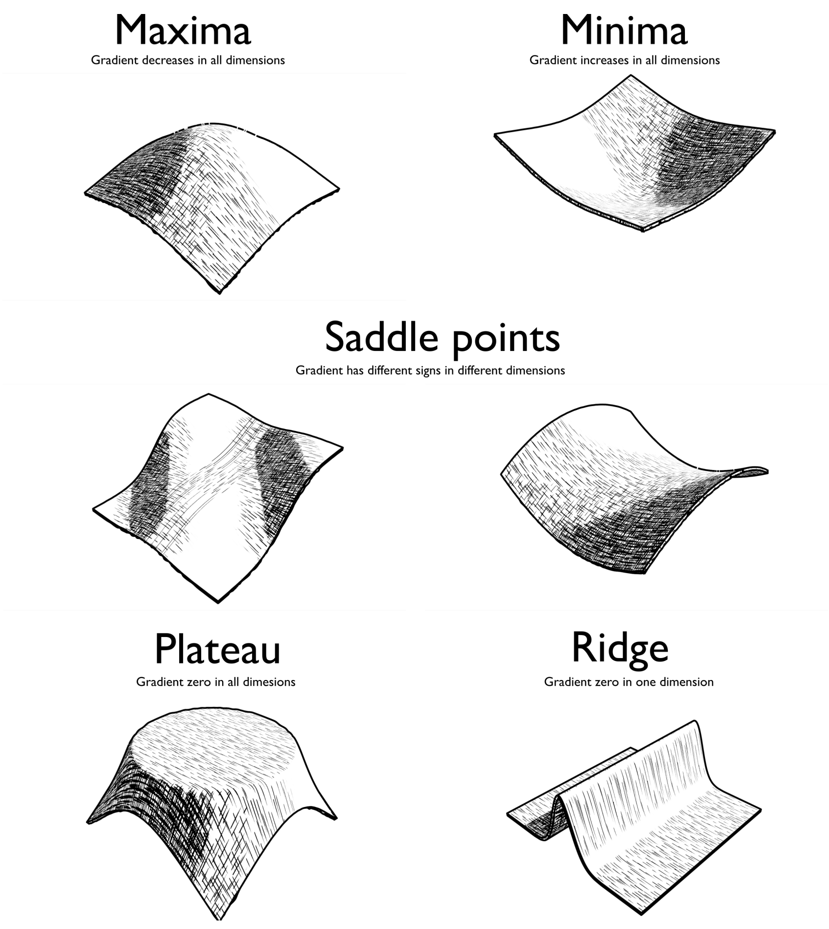
- When does gradient descent stop?
  - When the gradient (or rather, norm of the gradient) gets too small (drops below some threshold).
- Why would I *not* use gradient descent?
  - Your objective function is not C^1 continuous
  - You cannot differentiate your objective function for some reason
  - You have an even more efficient approach that already works (linear program, second-order methods) -- I wouldn't expect you to know that.

- What's the Jacobian?
  - Imagine a function operating on vectors and outputting vectors.
  - It might have n-dim inputs and m-dim outputs.
  - The Jacobian *at a specific input value of that function* tells you how much each output changes as each input is varied.
  - It's stored/manipulated as a matrix.
  - We could estimate it like this:

```
def approx_jacobian(f, x, eps=0.001):
    n = len(x)
    y = f(x)
    m = len(y)
    J = np.zeros(n, m)
    for i in range(n):
        d = np.zeros(n)
        d[i] = eps
        J[i, :] = (f(x + d) - f(x - d)) / (2*d)
    return J
```

- Then what about the Hessian?

    - For functions **that take vectors and return scalars** (i.e. the Jacobian is just one row), then the Hessian matrix *at a specific input value* gives the *second derivatives* for all possible pairs of inputs
    - This gives the curvature of the function at that point
    - It's useful for classifying "critical points" (where J(x)=0), e.g. to distinguish a maxima, minima, saddle point.


Maxima
Gradient decreases in all dimensions
Minima
Gradient increases in all dimensions
Saddle points
Gradient has different signs in different dimensions
Plateau
Gradient zero in all dimesions
Ridge
Gradient zero in one dimension

- When would random search be a good choice?
    - When your objective function has no sense of locality -- it makes no sense to talk about a "arbitrarily small change"
    - Which pair of shoes should I wear? I might measure their suitability and measure how good they were are resisting cold, for example, but I can't choose a shoe half way between a "leather boot" and a "plastic trainer".
- What is stochastic relaxation?
    - The property by which adding some kind of noise into an optimisation process can effectively smooth out the objective function
    - This can improve convergence if the objective function is very "bumpy"

Probability

- What does independence mean?
    - Independence of two random variables X and Y means that knowing X tells you nothing about Y and vice versa.
    - P(X|Y) = P(X) and P(Y|X) = P(Y) in this case; P(X=x)P(Y=y) = P(X=x) * P(Y=y) in this case, as well.
- What is the difference between a PMF and a PDF?

    - A PMF assigns discrete possibilities to probabilities (i.e. we can enumerate each possibility)
    - A PDF assigns continuous possibilities to densities (i.e. we can associate dense subsets/ranges with probabilities)

- When do I sum and when do I integrate?
    - Sum a PMF (discrete outcomes)
    - Integrate a PDF (continuous outcomes)

- Calculating PDF is taking samples of multiple multivariate uniform to get an approximation to the normal distribution?
    - **NO!**. I'm not sure what you mean, but you don't calculate PDFs.
    - They are functions which you could estimate; one way to do so would be to *assume* that the distribution was multivariate Gaussian
    - Then estimate the mean vector and covariance matrix
    - Then these give you a parametric form for the PDF (though you don't need to know the equation)

- What does the axiom of unitarity imply?
    - sum P(outcome) for all outcomes = 1
    - so if there are binary outcomes P(cat=good) and P(cat=bad), and cats can only be good or bad, then
        - P(cat=good) = 1 - P(cat=bad) and P(cat=bad) = 1 - P(cat=good)

- What does P(A=a|B=b) mean?
    - The conditional probability of A given that we know B is true.
    - We can compute this from a PMF, for example: P(A|B) = f_X(A, B) / f_X(B) = P(A, B) / P(B)

- What's the difference between an event and an outcome? How does it relate to probabilities?
    - Every probability distribution maps outcomes (possibilities) to probabilities (numbers from 0-1)
    - An outcome is one possibility
    - An event is a set of possibilities (possibly one, zero or many).
    - P(hour=6) might be an outcome; P(hour<6) is an event.
    - Probabilities of both outcomes and events are in the range 0-1
    - The sum of all the probability of outcomes = 1
    - The sum of any collection of events that cover all possibilities exactly once = 1
        - P(hour=1) + P(hour=2) + ... + P(hour=24) = 1
        - P(hour<6) + P(hour>=6) = 1

- What is the empirical PMF? How do I compute it? Is it the same as the true PMF?
    - The empirical PMF is a PMF we estimate from data.

    - Just the number of counts of each outcome observed divided by the total number of outcomes.
    - It's not the true PMF unless you sample all possible data -- it's an empirical (experimental) approximation.
    - Imagine throwing a 6 faced die 60 times -- would you really expect to get *exactly* 10 of each face?

- What is entropy? When would I use it?
    - It is a measure of surprise/disorder/uncertainty. Usually measured in *bits*.
    - You can compute from a PMF for discrete variables.
    - $H(X) = -\sum_x P(X=x)\log_2(P(X=x))$
    - The bigger it is, the more uncertain we are about an outcome under than distribution.

- When do I use expectation?
    - When you have values and associated probabilities.
        - A bad cat is worth £-5
        - A good cat is worth £100
        - You reach into a bag with 8 good cats and 2 bad cats.
        - What is the expected resale value of the cat you pick out?
        - E(cat) = -5 * (2/10) + 100 * (8/10) = £70

- How do I use expectation?
    - Expectation is *just* a weighted sum, where the weights are the probabilities, and each outcomes has a value as well as a probability.
    - It tells you the *average outcome* of a probability distribution
    - It only makes sense when outcomes have values that can be averaged.

```
def expectation(probs, values):
    expectation = 0
    for key, p in probs.items():
        expectation += p * values[key]
    return expectation
```

- When do I use Bayes' rule?
    - When you have P(B|A) and P(A) and you want P(A|B)
    - P(A|B) = P(B|A) P(A) / P(B)

- What is the probability that a cat is bad given that it screams?
    - P(bad|screams) = P(screams|bad) P(bad) / P(screams)
    - Note: this is useful, because "screams" is observable, whereas "bad" is unobservable.
    - We might write:
        - P(good=0|screams=1) = P(screams=1|good=0) * P(good=0) / P(screams=1)

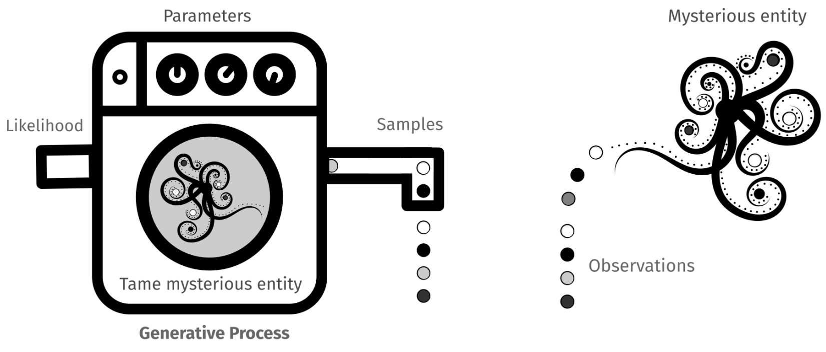
- Wait, how do I deal with computing P(B)?

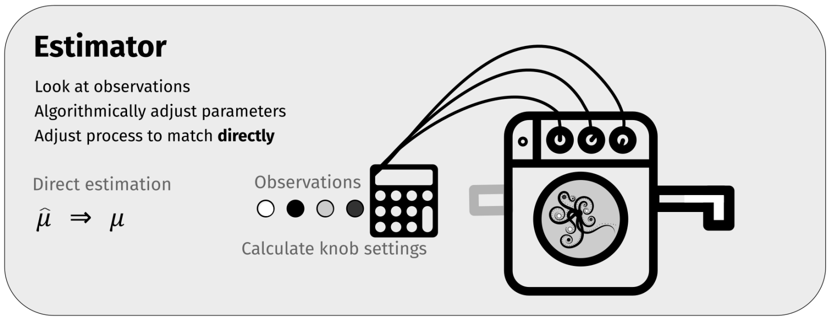
    - P(B) = sum(P(B|A) P(A)) over all A
    - So if cats can be good or bad, then
    - P(screams) = P(screams|bad) * P(bad) + P(screams|good) * P(good)

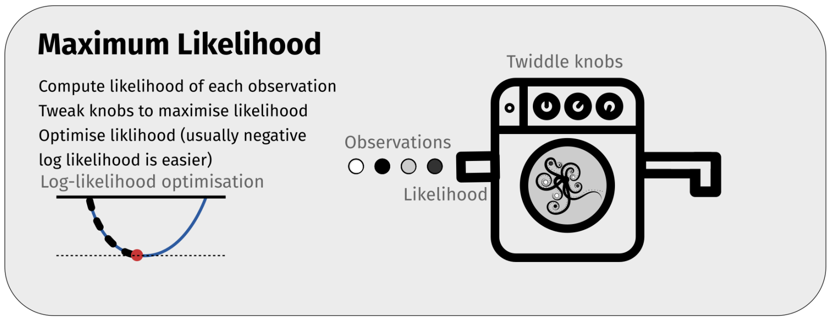
- That's just for one value, P(bad|screams) -- what if I want the whole distribution over possible cat behaviours?
    - Just compute P(x|screams) for each possible x (here good and bad)
    - This is a probability *distribution* over x, given that screaming was observed.

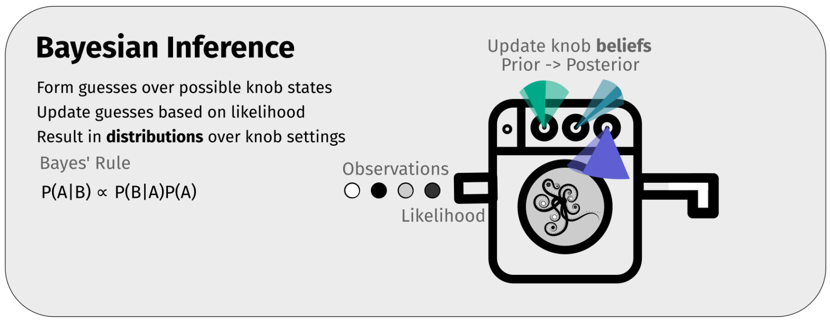
- What are parameters and observations?
    - Observations are collections of values we observe.
    - Parameters are values we *imagine* are governing a process that generates those observations.
    - Inference is trying to learn the values of parameters given observations.

- What's the difference between direct estimation, maximum likelihood and Bayesian inference?
    - **Direct estimation** compute a function of data (e.g. np.mean) and get a parameter estimate.
    - **Maximum likelihood** write down the likelihood of the observations as a function of parameters -- and then optimise this to find the parameters which make the data "most likely".
    - **Bayesian inference** we assume a distribution over the parameters (e.g. from experience, or a guess) "the prior", then we observe data, and use Bayes' rule to adjust the distribution to be compatible with the observation. Note: this is totally different from MLE, as it gives *distributions* over parameters, not single values.




Estimator
Look at observations
Algorithmically adjust parameters
Adjust process to match **directly**
Direct estimation
$\hat{\mu} \Rightarrow \mu$
Observations
Calculate knob settings


Maximum Likelihood
Compute likelihood of each observation
Tweak knobs to maximise likelihood
Optimise liklihood (usually negative
log likelihood is easier)
Log-likelihood optimisation
Twiddle knobs
Observations
Likelihood

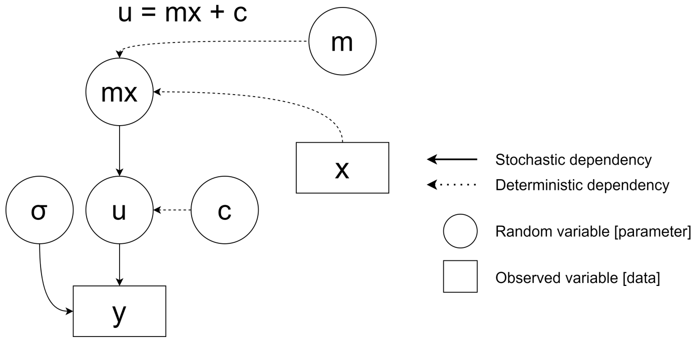

Bayesian Inference
Form guesses over possible knob states
Update guesses based on likelihood
Result in **distributions** over knob settings
Bayes' Rule
P(A|B) ∝ P(B|A)P(A)
Update knob **beliefs**
Prior -> Posterior
Observations
Likelihood

- What is likelihood?
  - It's how likely data would be to be observed *given a particular parameter setting.*
  - We can compute it by just evaluating the appropriately parameterised PMF or PDF at the observation.
  - E.g. likelihood of a fair die coming up 5 = 1/6

- Why do we use log-likelihood?
  - The likelihood of a *collection* of *independent* observations is just the product of those likelihoods.
  - But a product of lots of small values will be very very very small -- roundoff error will be likely.
  - Better to use the *sum* of *log-likelihoods* which is more stable to compute. log(ab) = log(a) + log(b)

- What do those graphs of Bayesian models mean?
  - They show how parameters relate to each other and to observations.
  - Each node represents a variable, and each edge represents a relationship.
  - We draw square boxes around things we observe and round boxes around parameters.

$y \sim N(u, \sigma)$
$u = mx + c$



- Do I need to know how to do linear regression?
  - Not in detail, no.

Time series

- Why does sampling matter? How does it work? What is aliasing?
  - We can't work with a continuous (in time) signal, like a real sound wave, on a computer.
  - So we just measure it (instantaneously) at regular intervals and store the sequence of measurements.
  - If we do this "frequently enough" then we don't lose any of the information
  - Frequently enough means at least twice as fast as the highest frequency oscillation in the signal
  - Nyquist limit = half sampling rate = the highest frequency that can be accurately reproduced at that sampling rate.
  - Try to sample too infrequently? You get aliasing = nasty artifacts caused by "ghosts" of high-frequency signals that could not be represented.
- What's a sliding window?
  - Imagine I have a long sampled signal, represented as an array of 5000 values.
  - I can process this signal in various ways.
  - One way is to split it into the first 10 values, then the average.
    - Then the ten values from 1-11, then 2-12, and so on.
    - Each time "sliding" the "window" by one sample
  - This works for any signal -- we can take a sampled signal of arbitrary length and convert it to fixed length vectors via this sliding window, then apply some function to it.
  - This is the most common way to deal with variable length arrays.
- What is convolution?
  - The generalisation of a moving average: just a sum of weighted neighbouring samples in a sampled signals.
  - This corresponds to a sliding window, where each element of the window has its own fixed weight.

- It is a useful way of defining filters, because *every* linear filter can be written as a convolution.

```
def convolve(a, b):
    n, m = len(a), len(b)
    out = np.zeros(n-m)
    for i in range(n-m):
        for j in range(m):
            out[i] += b[j] * a[i + j]
        # or
        # out[i] = np.sum(b * a[i:i+j])
```

- Do I need to know the Fourier transform formula?
- Why do I need the Fourier transform?
    - If you want to analyse the frequencies that make up a signal, the FT is the way to do it.
    - Takes a signal as a function of time, and returns it as a function of frequency
    - That means it *decomposes* each function into a sum of pure frequencies = sinusoids.
    - For every possible frequency, the FT gives the amplitude/magnitude ("loudness") of that component, and the phase ("time offset")
    - $y = A\sin(ft + p)$ is a pure sinusoid/pure frequency
        - A = amplitude
        - f = frequency
        - p = phase
        - t = time
- Why does it use complex numbers?
    - In essence, because that lets us wrap up both amplitude and phase into one number.
- What is the convolution theorem?
    - It just says that $FT(A) * FT(B) = FT(A \otimes B)$; that is, the FT interchanges multiplication and convolution.
    - E.g. $A \otimes B = IFT(FT(A) * FT(B))$, which can be a fast way to compute convolutions.
    - Or if you wanted to know what effect convolving with B would have, you could see exactly how much each frequency would be scaled and shifted by taking FT(B).
    - Or we could prove that every linear filter cannot add new frequencies: because a linear filter is defined by a convolution, and multiplying FT(A) * FT(B) can't create a non-zero element in A if it was zero before...
- What is the difference between the FT, the DFT, the FFT, the IDFT, IRFFT, ...?
    - FT = Fourier transform *for continuous functions*
    - DFT = Discrete Fourier transform *for discrete (i.e. sampled) signals*, like we usually have on a computer
    - FFT = Fast Fourier Transform = a fast way to compute the DFT O(N log N) instead of O(N^2) for N = power of 2
    - IFT = *Inverse* Fourier transform, turns frequencies back into time
    - IDFT = Inverse Discrete Fourier transfrom, etc.

- (we haven't considered things like the RFFT, which is just an efficient way of computing the FFT if you know the input is real numbers -- you can throw away half of the computation), but IRFFT means "inverse real fast Fourier transform"

```
In [ ]:
```