Exercise sheet

April 17, 2026

The following are exercises on agent foundations. Each problem is broken into a sequence of lemmas leading to a main theorem. ***For each subquestion, try to prove the stated lemma before reading on.*** *If you get stuck, you may treat the lemma as given and proceed to the next part.*

Before diving into a formal derivation, try to build an intuition for ***why*** *the statement should be true. Even if you don't complete the proof, having a clear intuitive picture of what's going on is more valuable than a mechanical derivation you don't understand. Don't worry if some of the terminology is unfamiliar — the exercises are designed to be self-contained, and it should be possible to follow the questions from context.*

Formal systems and programs. A *formal system* is a precise set of rules for deriving mathematical statements from axioms. Fix a formal system L that is powerful enough to reason about programs (for instance, it can express statements about arithmetic, and any program can be encoded as a mathematical object that L can talk about). We write $L \vdash \varphi$ to mean that the statement φ is *provable* in L, i.e. there exists a finite sequence of steps, each justified by the rules of L, that derives φ.

We say L is *consistent* if it never proves a contradiction. We write $\bot$ for a fixed contradictory statement (such as $0 = 1$), so consistency means $L \nvdash \bot$. We assume throughout that L is consistent.

Programs. By a *program* we mean a mechanical procedure that follows a fixed list of instructions. A program may *halt* (finish and produce an output) or *run forever* (keep executing without ever stopping). Since L can reason about programs, it can express the statement "program M halts", which we write as $\mathsf{Halts}(M)$.

The bridge between L and programs. Formal systems and programs are intimately connected, and the key to these exercises is switching back and forth between the two perspectives:

- **From programs to L** (concrete outputs become proofs)**.** If a program concretely produces an output (e.g. it halts after some number of steps, or it finds a string with a certain property), then L can verify this by tracing through the execution step by step. In particular: if a program actually halts, then L can prove that it halts.
- **From L to programs** (proofs can be found by search)**.** Proofs in L are finite strings that can be checked mechanically. So for any statement P, we can write a program that searches through all possible strings, checks whether each one is a valid L-proof of P, and halts if it finds one:

  $$\texttt{ProofSeeker}(P) \;:=\; \text{"try every string; halt iff one is a valid } L\text{-proof of } P\text{."}$$

  This program halts if and only if P is provable in L.

Whenever you derive a fact about L (e.g. that some statement is or isn't provable), ask what that implies for the corresponding proof-search program, and vice versa.

The provability predicate $\Box P$. We write $\Box P$ for the statement, *expressed within L itself*, that "P is provable in L." This is a genuine mathematical statement that L can reason about, because it is equivalent to the claim that `ProofSeeker`(P) halts, and L can talk about programs.

The crucial distinction is between $L \vdash P$ and $L \vdash \Box P$:

- $L \vdash P$ means that P is provable: there *exists* a concrete proof of P in L. This is a fact about L that we observe from the outside.
- $L \vdash \Box P$ means that L has proved a statement *about itself*: namely, that a proof of P exists (equivalently, that `ProofSeeker`(P) halts). But this is a claim L is making about the `ProofSeeker`(P) program, not a direct certificate for P.

Intuition. To see why $L \vdash P$ and $L \vdash \Box P$ are conceptually distinct, consider the two different ways L might prove that `ProofSeeker`(P) halts. The first is to actually trace through its execution: if it halts after, say, a million steps, L can verify this step by step, and the proof that `ProofSeeker`(P) found along the way is itself a direct L-proof of P. In this case, $L \vdash \Box P$ and $L \vdash P$ seems to come hand in hand. But there is a second way: L might reason *abstractly* about the program's behaviour without ever simulating it. (This is analogous to how you might argue that a sorting algorithm must eventually finish without tracing through every swap it makes.) Such a proof establishes that `ProofSeeker`(P) halts — and therefore that *some* proof of P exists — but the proof itself is about the *program*, not about P. It need not contain, or even hint at, what the actual proof of P looks like. This is the gap that L cannot close in general: knowing abstractly that a proof is "out there" is not the same as having the proof in hand.

What "provable in L" means (and what it does not). It is important to distinguish between being *convinced* that something is true and *proving it in L*. When we say "L can carry out this argument" or "L proves P," we do not mean that a reasonable person reading the argument would find it convincing. We mean something much more specific: that there exists a sequence of formulas, each of which is either an axiom of L or follows from earlier formulas by one of L's explicitly listed inference rules, and whose last line is P. The formal system L is a *machine*: it has no understanding, no intuition, and no ability to say "well, this obviously follows." Every single step must be justified by a specific rule.

From the outside, we can see that if `ProofSeeker`(P) halts then a proof of P exists, so P is provable. But can L carry out this reasoning internally, always concluding P from $\Box P$? Löb's theorem (Exercise 2) shows that the answer is no: any consistent L that derives P from $\Box P$ for all P is in fact inconsistent.

1 Exercise 1: Gödel's second incompleteness theorem

Key fact. If a program M actually halts (say, after 17 steps), then L can verify this by checking the execution step by step, so $L \vdash \mathsf{Halts}(M)$. However, if M runs forever, L cannot necessarily prove $\neg\mathsf{Halts}(M)$. (Informally: it is easy to certify that something stops, because you just exhibit the stopping point; but certifying that something runs *forever* is much harder, because you cannot check infinitely many steps.)

In fact, no consistent formal system can correctly settle the question "does M halt?" for *every* program M. To see why: if L could do this, we could write a program that, given any M, searches for an L-proof of either $\mathsf{Halts}(M)$ or $\neg\mathsf{Halts}(M)$. Since L is assumed to settle every case, this search would always find a proof and halt, giving us a mechanical procedure that decides whether any program halts. But such a procedure cannot exist (this is the *undecidability of the halting problem*, a fundamental result in computer science that we take as given here).

A self-referencing program. It is possible to write programs that refer to their own source code. (As a simple example, a program can carry its own source code as a string and then operate on it.) Using this idea, define the following program:

$$Z(A) \;:=\; \text{"search for an } L\text{-proof of } \neg\mathsf{Halts}(A(A))\text{; halt if one is found."}$$

Here $A(A)$ means "run program A with its own source code as input." So $Z(A)$ searches for a proof that the program A-run-on-itself runs forever.

Now consider feeding Z its own source code. The program $Z(Z)$ searches for an L-proof that $Z(Z)$ runs forever. Define the statement:

$$G \;:=\; \neg\mathsf{Halts}(Z(Z)).$$

In words: G says "$Z(Z)$ runs forever." Notice the self-referential structure: $Z(Z)$ halts if and only if it finds an L-proof of G (i.e. a proof that $Z(Z)$ runs forever).

Part 1(a). Show that G is true, assuming L is consistent.

Hint: Consider two cases. Either $Z(Z)$ halts or it doesn't. In each case, use the bridge between programs and L (if a program halts, L can prove it; if L proves something, the corresponding proof-search program finds that proof and halts) to derive what follows. One of the two cases leads to a contradiction with the consistency of L.

Part 1(b). Show that if L can prove its own consistency (i.e. $L \vdash \neg\Box\bot$), then L is in fact inconsistent.

*Hint: In Part 1(a), you argued from outside L that G is true, and the argument used only one assumption about L: that L is consistent. If L can prove its own consistency, then every step of your outside argument can be carried out **inside** L as a formal derivation. What would L then be able to prove? And what would the corresponding program do?*

Part 1(c). Suppose L can vouch for all of its own proofs, meaning $L \vdash \Box P \to P$ for every statement P. (Read this as: "whenever L can prove P, then P is actually true," and L itself asserts this.) Show that L is inconsistent, in two steps:

(i) First, show that L proves it never proves anything false. That is: for any P with $L \vdash \neg P$, show that $L \vdash \neg\Box P$.
Hint: The statement "if A then B" is logically equivalent to "if not B then not A". Apply this to $\Box P \to P$.

(ii) Apply (i) with $P = \bot$, using the fact that $\neg\bot$ ("a contradiction is false") is a tautology. Conclude that $L \vdash \neg\Box\bot$, and use Part 1(b) to finish.

Remark (Self-trust, tiling agents, and the Löbian obstacle).

Any sufficiently advanced AI may eventually be able to modify its own code or build a successor system more capable than itself. But this raises a subtle problem. If the successor is genuinely smarter, the original agent *cannot* predict exactly what it will do — just as the programmers of

a chess engine can reason that their program is "trying to win" without knowing its exact moves. So the original agent cannot verify its successor's safety by simulating it move by move. Instead, it must reason *abstractly* about the successor's design: "whatever my successor does, it will only take actions that it has proved lead to good outcomes."

This reasoning strategy is called *tiling*: the parent agent A_1 builds a child agent A_0, and wants to conclude not merely that A_0 will only take actions that A_0 has *proved to be safe*, but that those actions *actually are* safe. After all, A_1 can verify from A_0's source code that A_0 says "only take action x if I can prove that x leads to good outcomes." But this only tells A_1 that A_0 acts on what A_0's proof system certifies — it does not yet tell A_1 that what A_0's proof system certifies is actually *true*. To close this gap, A_1 needs to be able to prove, within its own reasoning, that A_0's proof system is *sound*: whenever A_0's system proves P, then P is actually true. When both agents use the same formal system L, this amounts to $L \vdash \Box P \to P$ for all P.

Part 1(c) shows this is impossible: any consistent system that asserts $\Box P \to P$ for all P is already inconsistent. A consistent system cannot vouch for its own proofs in the abstract — it can only trust a proof once it has *witnessed* it directly. This is the **Löbian obstacle**: the barrier created by Löb's theorem to self-trusting formal reasoning.

One might hope to work around this by having the parent use a *stronger* proof system than the child: a stronger system *can* trust a weaker one's proofs. But this means each successive agent in a chain of self-improvements must use a strictly weaker proof system than its predecessor, resulting in a "telomere" of logical strength that shortens with each generation. Eventually the chain runs out of trust. The *tiling agents* research programme studies how (and whether) this obstacle can be overcome, seeking agent architectures that can undergo indefinite self-improvement without their reasoning guarantees degrading at each step.

2 Exercise 2: Löb's theorem

Löb's theorem says: if $L \vdash \Box C \to C$ (i.e. L can prove "if C is provable then C is true"), then $L \vdash C$ (i.e. C is already provable in L). In other words, the only statements for which L can close the gap between "provably provable" and "provable" are the ones that were already provable to begin with.

The proof uses three properties of the provability predicate $\Box$. We state them here together with informal explanations of what they say about the proof-search program `ProofSeeker`.

(N)	*Necessitation.*	If $L \vdash \varphi$ then $L \vdash \Box\varphi$.
(K)	*Distribution.*	$L \vdash \Box(\varphi \to \psi) \to (\Box\varphi \to \Box\psi)$.
(4)	*Löb condition.*	$L \vdash \Box\varphi \to \Box\Box\varphi$.

Part 2(a). Understanding Necessitation. Necessitation says: if φ is provable in L, then L can prove that φ is provable. Explain why this is true from the perspective of `ProofSeeker`.

Hint: If a proof of φ exists, then `ProofSeeker`(φ) will find it and halt.

Part 2(b). Understanding Distribution. It is helpful to think of a proof of $\varphi \to \psi$ by analogy with a *function*: given any proof of φ as input, one can mechanically produce a proof of ψ as output (by writing down the proof of φ, attaching the proof of $\varphi \to \psi$, and applying the logical rule that from φ and $\varphi \to \psi$ one may conclude ψ).

With this analogy, $\Box(\varphi \to \psi)$ says that L has proved such a "proof-transforming function" exists. Distribution then says: if L knows that a function from φ-proofs to ψ-proofs exists, and L knows that a φ-proof exists, then L can conclude that a ψ-proof exists.

Explain why Distribution is true from the perspective of `ProofSeeker`.

Hint: If both `ProofSeeker`$(\varphi \to \psi)$ *and* `ProofSeeker`(φ) *halt, what can you do with their outputs? How can L deduce that* `ProofSeeker`(ψ) *halts?*

We now prove Löb's theorem. The key ingredient is a self-referential sentence constructed using the same idea as in Exercise 1 (a sentence that talks about its own provability). Specifically, there exists a sentence λ such that

$$L \vdash \lambda \leftrightarrow (\Box\lambda \to C).$$

In words: λ says "if I am provable, then C is true." (The existence of such a sentence is guaranteed by the same self-referential construction used to build G in Exercise 1; we take it as given here.)

Part 2(c). Show that $L \vdash \Box\lambda \to \Box C$.

Hint: The Löb sentence says $L \vdash \lambda \to (\Box\lambda \to C)$. Apply Necessitation to get this fact inside a $\Box$, then use Distribution twice (once to "unwrap" the outer implication, once to handle $\Box\lambda \to C$ inside). Use the Löb condition to handle the resulting $\Box\Box\lambda$.

Part 2(d). Now assume $L \vdash \Box C \to C$. Using Part 2(c) and the Löb sentence, derive $L \vdash C$.

Hint: From Part 2(c), you have $L \vdash \Box\lambda \to \Box C$. Chain this with the assumption $L \vdash \Box C \to C$ to get $L \vdash \Box\lambda \to C$. Now compare this with what λ says about itself.

Remark (The Santa Claus paradox and what Löb adds beyond Gödel).

The Löbian sentence $\lambda \leftrightarrow (\Box\lambda \to C)$ is a formal analogue of the *Santa Claus sentence* (also known as Curry's paradox). Consider the sentence S: "If this sentence is true, then Santa Claus exists." Let us try to determine whether S is true or false.

Well, S is an "if ... then ..." statement, so to check whether it is true, let us assume the "if" part and see whether the "then" part follows. So assume S is true. Since S says "if S is true then Santa Claus exists," and we are assuming S is true, it follows that Santa Claus exists. We have therefore shown: if S is true, then Santa Claus exists. But that is exactly what S says! So S is true. And since S is true and S implies Santa Claus exists, Santa Claus exists. Since nothing about this argument was specific to Santa Claus, the same reasoning "proves" any statement whatsoever.

The reason L does not fall prey to this paradox is that L cannot form a sentence that refers to its own *truth*. (A fundamental result called Tarski's theorem shows that no sufficiently powerful formal system can define a truth predicate for itself.) What L *can* do is refer to its own *provability*: the predicate $\Box\varphi$ is a legitimate statement within L. The Löbian sentence λ therefore substitutes "provable" for "true," asserting "if I am *provable*, then C." The proof of Löb's theorem shows that this substitution is enough to force $L \vdash C$ — but only when $\Box C \to C$ is already assumed, not unconditionally.

This mirrors a pattern: replacing "true" with "provable" transforms semantic paradoxes into precise theorems. The liar's paradox ("this sentence is false") becomes Gödel's sentence ("this sentence is not provable"), yielding the incompleteness theorems. The Santa Claus paradox ("if this sentence is true, then C") becomes the Löbian sentence ("if this sentence is provable, then C"), yielding Löb's theorem.

What Löb adds beyond Gödel. Gödel's second incompleteness theorem (Exercise 1) says that L cannot prove its own consistency. This is already a serious obstacle, but one might hope that consistency is a special case — perhaps L can still trust its proofs in less sweeping ways. Löb's theorem crushes this hope completely. It says that for *any* statement C, if L can prove "my provability of C implies C is true" (i.e. $L \vdash \Box C \to C$), then C was already provable. There are

no statements, not for which L can assert "well, if I *could* prove this, it would be true" without already being able to prove them. L does not trust its own proofs until it has witnessed them directly.

This is what makes Löb's theorem, rather than Gödel's, the fundamental obstacle for tiling agents. Recall the setup from Exercise 1: a parent agent A_1 builds a child A_0 that only takes actions it can prove to be safe. For A_1 to trust A_0, it needs to know that A_0's proofs track reality — that is, A_1 needs $\Box P \to P$ for the statements P that A_0 might act on. Gödel tells us A_1 cannot prove A_0's system is *consistent*. But Löb tells us something far stronger: A_1 cannot even trust A_0's system on a *case-by-case* basis. For any individual statement C, the only way L can derive "if my proof system proves C, then C is really true" is if C was already provable — in which case the trust was never needed in the first place.

Part 2(e). In a one-shot Prisoner's Dilemma, two players each choose to either *cooperate* (C) or *defect* (D). In this variant, instead of choosing directly, each player submits a *program* that receives the opponent's source code as input and outputs C or D. Consider the following agent, *FairBot*:

> **algorithm** `FairBot`(opponent):
> Search for an L-proof that `opponent(FairBot)` $= C$.
> **if** proof found **then return** C
> **else return** D

In words: FairBot cooperates with an opponent if and only if it can find an L-proof that the opponent cooperates with FairBot. Note that FairBot is *unexploitable*: if L is sound (i.e. L only proves true statements), then FairBot never cooperates with an opponent that defects against it.

The interesting question is what happens when FairBot plays against itself. At first glance, both mutual cooperation and mutual defection seem like stable outcomes.

Consider two copies $\texttt{FairBot}_1$ and $\texttt{FairBot}_2$ (identical programs with different implementations). Let A be the statement "$\texttt{FairBot}_1(\texttt{FairBot}_2) = C$" and B be the statement "$\texttt{FairBot}_2(\texttt{FairBot}_1) = C$." Prove that $L \vdash A \wedge B$, i.e. that the two FairBots mutually cooperate. Use Löb's theorem.

Hint: From FairBot's source code, $\Box A$ implies that $\texttt{FairBot}_2$ finds a proof that $\texttt{FairBot}_1$ cooperates, so $\texttt{FairBot}_2$ cooperates, i.e. B is true. Similarly $\Box B$ implies A. Combine these to show $L \vdash \Box(A \wedge B) \to (A \wedge B)$, and apply Löb's theorem.

3 Exercise 3: The Complete Class Theorem

The setup: decision-making under uncertainty. Imagine you must choose an action, but you don't know which *environment* you are in. There are finitely many actions $\mathcal{A} = \{a_1, \ldots, a_k\}$ and finitely many possible environments $\Omega = \{\omega_1, \ldots, \omega_n\}$. If you take action a and the true environment turns out to be ω, you receive a reward $u(a, \omega) \in \mathbb{R}$.

Example. You are deciding whether to carry an umbrella (a_1) or not (a_2). The environment is either "rainy" (ω_1) or "sunny" (ω_2). The rewards might be:

	ω_1 (rain)	ω_2 (sun)
a_1 (umbrella)	8	5
a_2 (no umbrella)	2	10

Decision rules. A *decision rule* $\delta = (\lambda_1, \ldots, \lambda_k)$ is a (possibly randomized) strategy: you play action a_j with probability λ_j, where $\lambda_j \geqslant 0$ and $\sum_{j=1}^{k} \lambda_j = 1$. A *pure* decision rule puts all its

weight on a single action (e.g. "always carry the umbrella"). A *mixed* rule randomizes (e.g. "carry the umbrella with probability 0.7").

The *reward of δ in environment ω* is the average reward under the randomization:

$$u(\delta, \omega) \;:=\; \sum_{j=1}^{k} \lambda_j \, u(a_j, \omega).$$

Risk vectors and the risk set. To compare decision rules across all environments simultaneously, we package the rewards into a single vector. The *risk vector* of a decision rule δ is

$$r(\delta) \;=\; \big(u(\delta, \omega_1),\; \ldots,\; u(\delta, \omega_n)\big) \;\in\; \mathbb{R}^n.$$

Each coordinate records how well δ performs in one environment. In the umbrella example, $r(a_1) = (8, 5)$ and $r(a_2) = (2, 10)$.

The *risk set* $\mathcal{R}$ is the set of all risk vectors achievable by some decision rule:

$$\mathcal{R} \;=\; \Big\{ \sum_{j=1}^{k} \lambda_j \, r(a_j) \;:\; \lambda_j \geqslant 0,\; \sum_{j=1}^{k} \lambda_j = 1 \Big\}.$$

Geometrically, $\mathcal{R}$ is the *convex hull* of the pure-action risk vectors $\{r(a_1), \ldots, r(a_k)\}$: the set of all weighted averages of these points.

Admissibility (not being dominated). A decision rule δ is *admissible* if there is no other rule δ' that does at least as well as δ in *every* environment and strictly better in at least one. If such a δ' exists, we say δ' *dominates* δ, and any rational agent should prefer δ' — after all, switching from δ to δ' never hurts and sometimes helps, regardless of which environment is the true one.

The *Pareto frontier* $\mathcal{F} \subseteq \mathcal{R}$ is the set of risk vectors of all admissible decision rules. A pure action a_j is *Pareto-optimal* if $r(a_j) \in \mathcal{F}$.

Bayesian expected utility. A different approach to decision-making is to assign a *prior* $\pi = (\pi_1, \ldots, \pi_n)$ representing your beliefs about how likely each environment is, where $\pi_i \geqslant 0$ and $\sum_i \pi_i = 1$. (For example, $\pi = (0.3, 0.7)$ means you believe there is a 30% chance of rain.) The *expected utility* of δ under π is the weighted average reward:

$$\mathrm{EU}(\delta, \pi) \;:=\; \sum_{i=1}^{n} \pi_i \, u(\delta, \omega_i) \;=\; \pi \cdot r(\delta).$$

A decision rule δ is *Bayes-optimal* under π if it achieves the highest expected utility among all decision rules: $\mathrm{EU}(\delta, \pi) \geqslant \mathrm{EU}(\delta', \pi)$ for all δ'.

The theorem. These two approaches to rational decision-making — admissibility ("never use a dominated strategy") and Bayesian expected utility maximization ("assign beliefs and maximize average reward") — turn out to characterize exactly the same set of decision rules:

Theorem (Complete Class). *Every admissible decision rule is Bayes-optimal under some prior π with $\pi_i > 0$ for all i. Conversely, every Bayes-optimal rule under such a prior is admissible.*

In other words: the decision rules that survive the "no domination" criterion are precisely those that arise from maximizing expected utility under some set of beliefs that doesn't rule out any environment entirely. This is significant because admissibility is an extremely weak rationality requirement — it says only that you shouldn't use a strategy when a strictly better one is available — yet it already forces expected utility maximization.

Part 3(a). Show the following two facts:

(i) Any admissible decision rule $\delta = \sum_j \lambda_j a_j$ places zero weight on dominated pure actions: $\lambda_j = 0$ whenever a_j is not Pareto-optimal. (In other words, the Pareto frontier $\mathcal{F}$ is contained in the convex hull of the Pareto-optimal pure actions alone.)
Hint: If δ places positive weight on a dominated pure action a_j, replace a_j with the action that dominates it. Does the resulting rule dominate δ?

(ii) Every Bayes-optimal rule under a prior π with $\pi_i > 0$ for all i is admissible.
Hint: If some δ' dominated δ, compare their expected utilities. What does $\pi_i > 0$ ensure?

Part 3(b). A *face* F of the Pareto frontier is a maximal convex subset of $\mathcal{F}$ of the form

$$F \;=\; \Big\{ \sum_{l=1}^{p} \mu_l \, r(a_{i_l}) \;:\; \mu_l \geqslant 0,\; \sum_{l=1}^{p} \mu_l = 1 \Big\}$$

for some subset of Pareto-optimal pure actions $\{a_{i_1}, \ldots, a_{i_p}\}$. Define the *difference vectors* $v_l := r(a_{i_l}) - r(a_{i_1})$ for $l = 2, \ldots, p$, and let $H = \operatorname{span}\{v_2, \ldots, v_p\}$. Show that H consists precisely of the directions along which one can move within F: that is, if $r(\delta) \in F$, then $r(\delta) + h \in F$ for some h only if $h \in H$.

Part 3(c). Let π be a vector perpendicular to the subspace H from Part 3(b), normalized so that $\pi_i > 0$ for all i and $\sum_i \pi_i = 1$. You may assume that the entire risk set $\mathcal{R}$ lies on one side of the hyperplane defined by H (i.e. no point in $\mathcal{R}$ scores strictly higher under π than the points on F). Show that:

(i) $\mathrm{EU}(\delta, \pi)$ takes the same value for all δ with $r(\delta) \in F$.

(ii) Every rule on the face F is Bayes-optimal under π.

Conclude the Complete Class Theorem: every admissible rule lies on some face of $\mathcal{F}$, and the prior π constructed from that face makes it Bayes-optimal.

4 Exercise 4: The Do-Divergence Theorem

Motivation: optimization as steering. A useful way to think about what it means for an agent to be *optimizing* is that it reliably steers the world into a narrow set of outcomes — outcomes that would be extremely unlikely to arise out of any random process . A thermostat keeps a room at 20 C despite varying weather; a chess player steers toward checkmate despite the opponent's moves. In each case, the actual outcome is concentrated in a small region of possibility space, whereas without the agent's intervention, outcomes would be spread broadly.

This exercise makes that intuition precise in an information-theoretic setting. We will show that an agent's ability to concentrate outcomes ("steer") is bounded by the amount of information the agent extracts from its observations.

Background: KL divergence. Given two probability distributions p and q over the same set of outcomes, the *Kullback–Leibler (KL) divergence* from q to p is

$$D_{\mathrm{KL}}(p \,\|\, q) \;:=\; \sum_x p(x) \log \frac{p(x)}{q(x)}.$$

This quantity is always $\geqslant 0$, and equals 0 only when $p = q$. It measures how "different" p is from q, with a particular asymmetry: $D_{\mathrm{KL}}(p\|q)$ is large when p places significant probability on outcomes where q assigns very little. In other words, it is large precisely when p concentrates on outcomes

that would be *surprising* under q. This can be seen as a signature of optimization: the agent's policy makes certain outcomes likely that would be very unlikely under a baseline policy.

Background: mutual information. The *mutual information* between two random variables A and O is

$$\mathrm{MI}(A;\, O) \;:=\; D_{\mathrm{KL}}\big(P[A,O] \,\|\, P[A]\,P[O]\big) \;=\; \sum_{a,o} P[a,o] \log \frac{P[a,o]}{P[a]\,P[o]}.$$

This measures how much knowing O tells you about A (and vice versa). It is zero when A and O are independent, and large when they are tightly coupled.

Setup. Consider an agent (the "demon") that observes some information O about the world and then takes an action A based on what it observed. The action and observation together produce an outcome X. The joint distribution is

$$P[X, A, O] \;=\; P[X \mid A, O]\; P[A \mid O]\; P[O].$$

The factor $P[A \mid O]$ encodes the demon's *policy*: how it chooses actions as a function of its observations.

Now consider a *blind baseline*: the demon still acts, but ignores its observations, choosing actions independently of O. We write the blind baseline distribution as

$$P[X, A, O \mid do(A)] \;=\; P[X \mid A, O]\; P[A]\; P[O].$$

The notation $do(A)$ means we have "intervened" on the action, replacing the demon's observation-dependent policy $P[A \mid O]$ with the marginal $P[A]$ (the overall frequency of each action, ignoring which observations prompted them). The mechanism $P[X \mid A, O]$ by which actions and observations produce outcomes is unchanged — only the demon's strategy has been lobotomized.

The theorem. Prove the Do-Divergence Theorem:

$$D_{\mathrm{KL}}\big(P[X] \,\|\, P[X \mid do(A)]\big) \;\leqslant\; \mathrm{MI}(A;\, O).$$

The left side measures how much the sighted demon's outcome distribution differs from the blind baseline's. In the language of steering: it measures how much the demon has concentrated outcomes into regions that would be unlikely without observation-dependent action. The right side is the mutual information between actions and observations — how much the demon's actions depend on what it sees. The theorem says that **the degree of steering is bounded by the information the demon uses**.

Useful fact:

- **Monotonicity of KL divergence.** For any two joint distributions $p(x, y)$ and $q(x, y)$, marginalizing out y can only decrease KL divergence: $D_{\mathrm{KL}}(p(x) \,\|\, q(x)) \leqslant D_{\mathrm{KL}}(p(x, y) \,\|\, q(x, y))$. (Intuitively: forgetting information can only make two distributions look more similar, never less.)

Hint: Compute $D_{\mathrm{KL}}(P[X, A, O] \,\|\, P[X, A, O \mid do(A)])$ by expanding the log ratio using the factorizations above

Remark (Maxwell's demon and the thermodynamics of optimization).

Maxwell's demon is a thought experiment in which a tiny intelligent being controls a door between two halves of a box of gas. By observing each molecule's position and selectively opening the door, the demon can sort all molecules to one side, creating a highly ordered (low-entropy) state from

an initially disordered one. In our notation: the outcome X is the final configuration of molecules, the observations O are the demon's measurements of molecular positions, and the actions A are its door openings.

Under the blind baseline (opening the door at random), molecules are roughly equally likely to be on either side, so $P[X \mid do(A)]$ is spread broadly. If the demon perfectly sorts all n molecules to the right, $P[X]$ is concentrated on a single configuration, and the KL divergence between these distributions is $n \log 2$ (i.e. n bits). The theorem therefore says that perfectly sorting n molecules requires $\mathrm{MI}(A; O) \geqslant n$ bits: the demon must gather at least n bits of information about the molecules to reduce the gas's entropy by n bits. This illustrates the idea that any agent that steers a system into a narrow, unlikely region of outcome space (low entropy) must pay for this steering with mutual information.

5 Exercise 5: Channel Additivity

Consider two independent channels $X_1 \to Y_1$ and $X_2 \to Y_2$ with a fixed joint channel

$$P[Y \mid X] \;=\; P[Y_1 \mid X_1]\, P[Y_2 \mid X_2].$$

That is, output Y_1 depends only on input X_1, and Y_2 depends only on X_2; the two channels do not interact. We are free to choose the input distribution $P[X]$ (which may correlate X_1 and X_2), and the joint distribution over everything is then $P[Y, X] = P[Y \mid X]\, P[X]$. The goal is to maximize the mutual information $\mathrm{MI}(X; Y)$, called the *information throughput* of the channel.

Part 5(a). Show that for any input distribution $P[X]$,

$$\mathrm{MI}(X; Y) \;=\; \mathrm{MI}(X_1; Y_1) \;+\; \mathrm{MI}(X_2; Y_2) \;-\; \mathrm{MI}(Y_1; Y_2).$$

Hint: Write each mutual information as a KL divergence between the joint and the product of marginals, expand the log ratios using the channel factorization, and collect terms.

Part 5(b). Given any input distribution $P[X_1, X_2]$, define

$$Q[X_1, X_2] \;:=\; P[X_1]\, P[X_2],$$

i.e. the product of the marginals. Show that $\mathrm{MI}_Q(X; Y) \geqslant \mathrm{MI}_P(X; Y)$.

Hint: Under Q, the inputs are independent. What happens to $\mathrm{MI}(Y_1; Y_2)$ when $X_1 \perp\!\!\!\perp X_2$, given the channel factorization? Use Part 5(a).

Part 5(c). Use Part 5(b) to conclude that there always exists a throughput-maximizing input distribution under which $X_1 \perp\!\!\!\perp X_2$.

Remark. This result has a natural interpretation in terms of optimization and agency. Think of X as the actions of an agent and Y as the outcomes it cares about. The channel $P[Y \mid X]$ — how actions influence outcomes — is fixed by the environment and outside the agent's control; the agent only gets to choose its policy $P[X]$. The factorization condition on $P[Y \mid X]$ says that the environment is *modular*: the two groups of outcomes Y_1 and Y_2 are each influenced only by their respective actions X_1 and X_2. Our theorem says that if the environment is modular in this sense, then the agent can always find an optimal policy that is modular in a corresponding sense — specifically, the two groups of actions need not be coordinated at all and can be chosen independently.