Changes to CSCC63 in 2026, and their rationale

The offering of CSCC63 in the Winter of 2026 incorporated significant changes to the material taught in the course, as well as to certain presentation and teaching methods. The main goals underlying the changes were:

1. Prioritize complexity and computability material that is conceptually meaningful and relevant to general non-theory students in computer science.
2. Revise the presentation (and selection) of core complexity materials, to align with current prevalent standards and best practices in teaching basic complexity.

> **What is the course's focus?** This course teaches how to mathematically model general computation and how to analyze the models.
>
> The main conceptual focus of the course is impossibility results (i.e., lower bounds), which rigorously prove limits for models of computation. A secondary focus is utilizing impossibility results for productive purposes, i.e. building algorithms and protocols from lower bounds.
>
> The key points I'd like students to take from the course are:
>
> - We can abstract computation into mathematical models, and prove impossibility results for these models.
> - This abstraction is relevant to real-world scenarios.
> - Impossibility results can be leveraged to build useful algorithms.

Brief changelog for the course, compared to some past offerings:

1. Computability (reducing the volume of this part): Removed the computability hierarchy (and all references to problems "above $\mathcal{RE}$"), Kleene's recursion theorem, the proof of Rice's theorem, and various technical notions (e.g., transducers, quines, T-predicates, operations on computability classes, etc.).
2. Basic complexity (aligning content and presentation with current standards): Added the time hierarchy and space hierarchy theorems. De-emphasized nondeterministic machines. Split the proof of Cook-Levin into modular steps.
3. Extensions of $\mathcal{P}$ vs $\mathcal{NP}$ (adding core materials from the last 20-30 years): Added fine-grained lower bounds for problems in $\mathcal{P}$, and hardness of approximation for problems in $\mathcal{NP}$.
4. Applications of lower bounds (new part): The last two weeks demonstrate how to use lower bounds for productive purposes, using commitment schemes and interactive proofs (zero-knowledge).

Contents

Context. At UofT Scarborough (UTSC), the course numbered CSCC63 is the last in the chain-of-dependencies of mandatory theory courses for undergraduate students specializing in computer science.[1] The course can be learned in parallel to the standard algorithms course CSCC73, although it is often taken after the latter.

Students arrive at this course having learned asymptotic analysis and induction, basic algorithms (sorting, BFS/DFS) and data structures, automata theory, and sometimes even a lower bound (e.g., comparison-based sorting).

This course is the first time students posses enough background to mathematically model general computation, and to analyze the models. It is also, for the vast majority of students, the last undergraduate course in which they learn about the theory of computation. Therefore, this course is the main one (typically the only one) in which students learn how to model general computation and analyze it, which is a key part of the theoretical foundations of computer science.

Comparisons to other institutions are not precise, since different institutions organize their undergraduate teaching differently. However, many universities have courses analogous to CSCC63, being last in the chain-of-dependencies for undergraduate theory courses, and teaching how to model and analyze general computation. At many universities (mentioned below), the content of the course changed in the last several decades, in a way similar to the changes made to CSCC63 in 2026.

Organization. The rest of this document explains the changes in more detail, first elaborating on each of the four points in the brief changelog above, in order, and then explaining trade-offs related to two less conventional choices.

1 Removed material: Reducing the volume of computability

Computability theory asks which problems are computable given an arbitrarily large (finite) amount of resources. The fact that certain problems are uncomputable is important for the theoretical foundations of computer science.

However, the distinction between "computable" and "uncomputable" problems is crude, and provides only limited information as to which problems can actually be computed, because it does not take into account resource limitations. Specifically, even computable problems may be beyond reach of any real-world computational procedure, whose resources are inherently limited. More refined information is obtained by analyzing resources-bounded models, in the study of complexity theory.

Moreover, the detailed study and classification of uncomputable problems, while mathematically interesting, does not have direct meaning and implications for modeling computation. This is because this study focuses, by definition, on problems that are beyond the reach of any computational procedure.

[1] That is, they are mandatory for students that take the equivalent of a major in CS, which is called a "specialist" at UTSC.

> My view is that an undergraduate course about the theoretical foundations of computer science should present computability theory mostly as a preamble to complexity theory, since the latter is *more relevant to the topic* and has a *clearer conceptually meaning*.
>
> It is good for students to learn that some problems cannot even be computed, regardless of resources. This is important, surprising, and also serves as a first example of a lower bound. But other than this message and a few examples (e.g., halting problem, universal problem, Kolmogorov complexity), course time would be better spent on other materials.

Computability theory is intellectually rooted in mathematics and logic of the early 20^{th} century. It is still fruitfully studied, but perhaps less so in theoretical computer science, given that it receives far less attention in flagship conferences (e.g., in STOC and FOCS), compared to areas such as complexity theory, algorithm design, cryptography, distributed systems, etc. See, e.g., [Kou23] for a brief historical survey of how modeling efficient computation (i.e., complexity theory) developed as a more refined alternative, and also see [Soa16] for a survey of past and current computability theory.

Topics that were removed from previous versions of the course.

1. Problems outside RE, arithmetical computability hierarchy, and *co*RE. This is a topic that may be quite niche even within theoretical computer science, and as far as I can see, its conceptual meaning is very limited when modeling real-world computation. I do not think it should be part of mandatory curriculum in CS.

2. Kleene's recursion theorem. This theorem has an interesting conclusion ("a Turing machine knows its own code, wlog"), but its value and meaning are more technical and low-level, rather than conceptual. There is no damage in including it, but in my opinion it is not worth the time in a basic undergraduate course.

3. The proof of Rice's theorem. This is an interesting theorem, and it is worth stating briefly in class (as I do). However, I don't think that its proof is worth the time, since it doesn't add an interesting idea to other computability-type reductions.

4. Auxiliary technical notions: Such as transducers, quines, T-predicates, operations on computability classes, and more.

This approach is current mainstream. Reducing the volume of computability theory when teaching the theoretical foundations of computer science, in favor of more complexity-theoretic content, is the current common approach. It is very common in analogous undergraduate courses, it is the approach taken by relevant textbooks, and it has been advocated for by researchers in the area long before the current text:

- In Goldreich's 2008 textbook [Gol08], the "Teaching Note" about computability explicitly recommends to reduce its volume in *any undergraduate course about theory of computation*, and instead focus the course on Computational Complexity.

  The note suggests that the two topics of modeling computation (objects, functions, and computation as strings and Turing machines) and of computability results never occupy more than 25% of an undergraduate course.

- As explained by Koucký [Kou23] in his rationale for redesigning an analogous course, "the mainstream of theoretical computer science has changed since the 1970's. The main take-away message from the past 90 years of development of theory of computation is the quest to capture what is and what is not efficiently solvable. The notion of efficiency is not static. It progressed from being computable (recursion theory), to being in $\mathcal{P}$ (complexity theory), to being linear or quadratic (streaming algorithms and fine-grained complexity)."

- Many undergraduate courses on "computability and complexity" in the last 20 years assign relatively low volume to computability. Steve Cook's CSC463 [Coo18] excluded all of the computability topics that I'm excluding.[2] A similar approach has been taken at MIT, Stanford, Technion, U Chicago, Yale, etc. [Aar11; Tre05; Ale26; Asp24; Hoz24; C.S25], who devoted only $13\% - 16\%$ of their course content to computability, mostly showing a few examples of uncomputable problems and (sometimes) another result such as the statement of Rice's theorem.

  The above refers to theory courses that are last in the chain of dependencies for undergraduate theory, and teach both computability and complexity. The titles of these courses differ: some of them are titled only "computability" [Ale26] for historical reasons, whereas others are titled only "complexity" [Asp24; Hoz24]; but all of them teach both, and serve the same purpose. Indeed, regardless of the name, these courses give more weight to complexity.

- Standard textbooks in the area from the last 20 years present computability shortly, as an intellectual preamble to complexity theory. For example, in Wigderson's "mathematics and computation" [Wig19], computability theory serves as a short "Prelude" (sic); and in textbooks covering both computability and complexity, the former is presented very shortly and as setting the stage [AB09; Vio26].

2 Revised material: Selection and presentation of core complexity materials

In teaching core complexity materials, the changes made to the course are both in the selection of materials and in the ways of presenting these materials. Specifically, time

[2] That is, Cook excluded the arithmetical computability hierarchy, problems outside RE, Kellene's recursion theorem, and Rice's theorem. He had one more lecture about computability than me, showing more examples of undecidable problems.

and space hierarchies were added (see Section 2.1), the model of "non-deterministic machines" has been strongly de-emphasized (see Section 2.2), and the proof of the Cook-Levin theorem has been decomposed into meaningful steps (see Section 2.3).

These changes reflect current standard best practices in teaching basic complexity theory. They are used by all major textbooks from the last 20 years (sometimes even explicitly recommended in teaching notes), and also used in the vast majority of complexity courses I could find; I elaborate below. Moreover, these choices became standard in teaching complexity for *good reasons*, which I'll explain below.

2.1 Time and space hierarchies

Time and space hierarchies are arguably the most basic results in complexity theory. They're also easy to learn, since they are provable using the same diagonalization-type techniques as the ones used in computability theory.

> I can see why someone might want to exclude these, since the conceptual point appears almost trivial: "More resources $\Rightarrow$ more power".
>
> However, I view them as a sanity check for our models (i.e., the models admit basic sensible results), and in addition as a good first demonstration to students that *resource-based lower bounds can, indeed, be proved*.

Including these results is a completely standard choice. Any textbook or course about complexity theory that I know teaches these results, almost invariably as one of the first in complexity (see [AB09; Gol08; Vio26; Wat26], as well as the older books such as [Sip96]; also see the courses [Aar11; Tre05; Coo18; Ale26; Hoz24; Asp24; C.S25]).

2.2 De-emphasizing non-deterministic machines

The notion of a "non-deterministic machine" is an auxiliary technical notion, which doesn't correspond to any real-world model. The conceptually meaningful notion is that of verifying proofs, or equivalently a prover-verifier system. Since the two formalizations are technically equivalent, it is better for students to study the meaningful notion that makes conceptual sense, rather than the auxiliary technical notion.

> This is *important*. The reason that $\mathcal{P}$ vs $\mathcal{NP}$ is interesting is because it's an abstract model for an important question about computation (i.e., verifying vs computing). Students are more likely to *understand its meaning and importance* when it is presented using definitions of computation that model the relevant phenomena directly, rather than in an auxiliary technical way that needs a proof-of-equivalence to make sense.

De-emphasizing non-deterministic machines is standard by now in teaching complexity. The reasons for this de-emphasis (i.e., the ones mentioned above) have been explicitly spelled out in standard complexity textbooks, starting around 20 years ago:

> "We try to avoid non-deterministic machines as much as possible... we believe that these fictitious "machines" have a negative effect both from a conceptual and technical point of view. The conceptual damage caused by using non-deterministic machines is that it is unclear why one should care about what such machines can do. Needless to say, the reason to care is clear when noting that these fictitious "machines" offer a (convenient but rather slothful) way of phrasing fundamental issues [i.e. solving verifiable search problems – R.T.]. The technical damage caused by using non-deterministic machines is that they tend to confuse the students". [Gol08]

> "One should note that, unlike standard TMs, NDTMs are not intended to model any physically realizable computation device." [AB09]

The Arora-Barak textbook [AB09] defined $\mathcal{NP}$ using prover-verifier systems, and presented non-deterministic machines once, in retrospect, along with the equivalence. Watson's textbook [Wat26] has no mention of non-deterministic machines. Viola's textbook [Vio26] presents the quantifier-based definition, calling it "proof verification".

2.3 Breaking down the Cook-Levin proof into meaningful steps

The proof that SAT is $\mathcal{NP}$-complete is notoriously challenging for students. Older presentations of the proof tended to be "one-shot", directly reducing each $\mathcal{NP}$-problem to SAT with long and formidable quantifier-expressions.

It has become, however, very common to decompose the proof into several components, each of them (a) Having an interesting conceptual meaning; (b) More easily understandable at the technical level. Specifically, the steps are:

- Reduce any $L \in \mathcal{NP}$ to a natural problem, often called *CircuitSAT*, which calls for deciding the satisfiability of a given Boolean circuit. This computational problem appeals to non-theory students (since these are natural models of electrical circuits), and the reduction itself is visual and relatively intuitive.[3]

- Reduce *CircuitSAT* to 3SAT. Separating out this reduction helps students isolate and identify the most important technical message of the Cook-Levin reduction, which is that "computation is local and hence can be verified locally".

  Again, the visual component here helps. It's also useful pedagogically to separate this part, so that the technical message above doesn't get confused with other technical content (which is put into the initial reduction to *CircuitSAT*).

Breaking down a hard proof into several modular parts is valuable in and of itself, and doubly so when each of the components is natural and has an interesting point. It allows to digest the point of each step separately. I see *no advantage in teaching the "one-shot" version* instead of teaching the nicer decomposition. It obscures the conceptual and technical content of each step, and makes the proof harder to digest. See also

[3] Interestingly, *CircuitSAT* is an important problem in current complexity research.

As mentioned above, this decomposition has been very common in the last 20 years. See, e.g., [Gol08, Section 2.3.3.1] for an explicit argument in favor of it. Also see [AB09; Wat26; Vio26] for similar presentations of the proof,[4] as well as courses such as [Coo18; Tre05; Hoz24; C.S25].

(In the 2026 offering of CSCC63, the second step was decomposed further: First reducing $CircuitSAT$ to $3CSP$, and then reducing $3CSP$ to 3SAT. The former reduction has the interesting conceptual observation, and the latter reduction is easier and concludes the proof. This further decomposition is relatively minor.)

3 Added material: Fine-grained lower bounds and hardness of approximation

Instead of the material in computability theory that was removed, the course included two topics that are natural extensions of $\mathcal{P} \neq \mathcal{NP}$. The first topic topic is fine-grained polynomial lower bounds for interesting problems in $\mathcal{P}$ (see Section 3.1) and the second topic is hardness of approximation for problems in $\mathcal{NP}$ (see Section 3.2).

As explained below, each of these topics is *very meaningful and directly relevant to students*, especially to general non-theory students. This is because both topics model important questions about real-world computation, in a way that makes complexity theory directly relevant to practical situations students may encounter. Moreover, the two topics have been central in complexity theory research in the last 20-30 years.

3.1 Fine-grained lower bounds

Fine-grained complexity tries to discover the precise time complexity of solving specific interesting problems. For example, fixing some computational model (e.g., multi-tape TMs), for certain problems unconditionally solvable in polynomial time n^k, fine-grained complexity tries to prove a matching lower bound, e.g. of the form $n^{99 \cdot k}$.

Relevant results include, for example, showing (under an assumption that is a strengthening of $\mathcal{P} \neq \mathcal{NP}$) that it's impossible to solve k-clique in time $n^{\epsilon \cdot k}$ for some $\epsilon > 0$, and ditto for k-SUM, k-Orthogonla-Vectors, and similar related problems. Another relevant conjecture in the area is that it's impossible to improve on the standard cubic-time APSP algorithm, which is taught in basic algorithms courses.

Results in this area rely on stronger versions of $\mathcal{P}$ vs $\mathcal{NP}$, which assert that some problems in $\mathcal{NP}$ require exponential time to solve ("exponential time hypotheses"). These are interesting extensions of $\mathcal{P} \neq \mathcal{NP}$, which have a clear meaning that is relevant to students, although I personally wouldn't teach these extensions in CSCC63 only for their own sake (i.e., without the connection to fine-grained complexity).

Fortunately, the proofs in this area are technically *very similar to standard $\mathcal{NP}$-completeness reductions!* In fact, I suspect that this is the one of the easiest parts of the

[4] The $CircuitSAT$ problem is explicitly defined in the textbooks [Gol08; Vio26], and implicitly defined in the textbooks [AB09; Wat26].

course for students, and it only seems "advanced" to more experienced researchers because we haven't learned it in our own undergraduate courses. In CSCC63 I taught three proofs, all of them direct extensions of $\mathcal{NP}$-completeness reductions.

> It is important for non-theory students to know the conceptual message that complexity theory *can prove fine-grained lower bounds for many natural problems in* $\mathcal{P}$. Moreover, the specific problems studied in the area are important, and students may encounter them in practice.
>
> *The bang (conceptually) for buck (technical difficulty) in this topic is wonderful.* In my opinion, fine-grained lower bounds should be a staple of every undergrad course that teaches complexity theory.

The topic is relatively new, only $\approx$ 15-20 years old, so it may still be in the process of being incorporated into undergraduate complexity courses.

3.2 Hardness of approximation

Hardness of approximation shows that if $\mathcal{P} \neq \mathcal{NP}$, then certain problems are hard not only to solve precisely, but also to approximate; for example, it's hard to decide if a given graph has a clique of size $n/2$ or if all cliques are of size at most $n/100$.

> It is important for students to know that the lower bounds they learned are robust, i.e. they don't break down by allowing small approximation errors. This helps them understand that *complexity theory is relevant to real-world scenarios*, which almost always tolerate some approximation error.

An extra bonus is that this topic involves a connection with probabilistically checkable proofs. The latter topic is *conceptually meaningful for non-theory students* ("how to check proofs without looking at all of them?"), and it is also widely used in practice (e.g., in cryptographic applications and zero-knowledge proof systems). Thus, it is directly relevant to general CS students, who will appreciate knowing that there is theory underlying the PCP constructions they may encounter.

Sadly, at the technical level one cannot teach even a fragment of the PCP theorem in an undergraduate course. However, it is useful to state the theorem, explain its meaning and significance, and show students the equivalence between PCPs and hardness of approximation. The latter is technically easier than other parts of the course.

Students can internalize the topic and practice by proving hardness of approximation results based on PCPs that are given to them as a black-box; and by building PCPs based on hardness of approximation results (given as a black-box).

4 Added material: Applications of lower bounds

A major discovery in theoretical computer science is that *lower bounds are useful*; that is, lower bounds can be leveraged to build useful algorithms and protocols. The last

part in the course (of about two weeks) is dubbed "lemons into lemonade", and its purpose is to communicate this message and to show examples.

> In my opinion this is *one of the most meaningful and valuable messages for students*, many of whom appreciate "building things" more than "proving that we can't build things". Also, the message can be very surprising for students, and in this sense its information content is high.
>
> To make the message as relevant as possible, a good choice is to present algorithms and protocols that are (a) central in theoretical research and (b) popular in practice (so students may encounter them).

The downside is that these are typically advanced materials, so one has to choose topics carefully and cover them relatively superficially.

4.1 Cryptographic applications

In CSCC63, the two applications presented were cryptographic, i.e. protocols for secure computation. In general, I believe that it's useful to include some cryptographic content at the end of undergraduate complexity courses. This is due to two reasons:

- Cryptographic constructions are ubiquitous both in theoretical research and in practice, which makes them good candidates for lower bound applications. In particular, cryptographic applications can be *naturally appealing for all students*, especially non-theory students.
- It is useful for students to understand the *relationship between lower bounds from complexity theory and "secure computation"* (i.e., cryptography), since they are at least superficially familiar with the latter, and encounter it in many situations. In particular, it's useful for students to know that cryptographic protocols they encounter can be *backed by rigorous theory*, and this makes the material learned in CSCC63 directly relevant to students.

Another consideration is specific to UTSC, but may apply to some other places: There is currently (2026) no course at UTSC fully teaching the theoretical foundations of cryptography with this type of rigorous approach (e.g., PRGs based on OWFs, constructions of ZK/PKE/FHE with rigorous security reductions, etc.).

4.2 Two specific choices

The two specific choices for CSCC63 were commitment schemes, preceded by pseudorandomness; and zero-knowledge, preceded by interactive proofs. These are currently very popular in practice, and involve learning about major achievements of complexity theory from the last half-century (see, e.g., the recent series of public talks by Wigderson [Wig26b; Wig26a]). For both topics, the course included definitions, a statement of a general interesting theorem, and one toy construction.

Nevertheless, it's important to emphasize that these choices have downsides, mainly that the definitions are complicated and may "feel" unfamiliar.

Pseudorandomness and commitment schemes. A commitment scheme is a two-party protocol, where in the first step a "committer" commits to a value without revealing the value itself, and in the second step the committer reveals the value (in between, the "receiver" holds a commitment to the value, but doesn't know what the actual value is). This is a natural object that is easy for all students to appreciate.

The course included the definitions of pseudorandomness and indistinguishability, the basic conjecture in cryptography that PRGs exist, and a proof that this conjecture implies $\mathcal{P} \neq \mathcal{NP}$.[5] Students then learned how to construct commitment schemes from PRGs (using Naor commitments).

The construction and proof is less technically challenging than the rest of the course, and the main challenge is internalizing the complicated definitions.

Interactive proofs and zero-knowledge proofs. Interactive proofs are a staple of research in complexity, ubiquitous in practice, and a natural extension of $\mathcal{NP}$. Thus, they fit well in a complexity course, and naturally appeal to all students.

Zero-knowledge proofs are interactive proofs that reveal nothing to the verifier, other than the "bit" indicating the validity of the claim. They have been central in theoretical research for many decades, and in the last 15 years became hugely popular in the (perhaps dubious) area of blockchains, cryptocurrencies, etc.

The course included a toy example of a zero-knowledge proof (a simplified variant of graph non-isomorphism), the definition of zero-knowledge, and the statement that every $L \in \mathcal{NP}$ has a zero-knowledge proof. As before, the main challenge is internalizing the definition of ZK, rather than any technical statement. My hope is that the simple example can serve to elucidate it.

5 Two unconventional choices

The following two choices are relatively non-standard, especially compared to the standard changes described in Sections 1 to 3.

- Removing *co*$\mathcal{NP}$ and the polynomial-time hierarchy. This is an important area of research for theorists, and it can be made meaningful to non-theory students (by presenting it as studying games with several quantifiers). It is good to include it in an undergraduate complexity course, and many undergraduate complexity courses do include this content, though not all of them (e.g., [C.S25]).

  However, I believe that *non-theory students* will find more value in other topics, which are just as central in complexity research. The trade-off I see is between

[5] A more standard route would have been to present OWFs as an extension of $\mathcal{P} \neq \mathcal{NP}$ and then state the result that OWFs imply PRGs.

teaching applications of lower bounds (interactive proofs, commitment schemes, zero-knowledge) and teaching the polynomial-time hierarchy.[6]

- Not classifying reductions to types such as Cook-reductions, Levin-reductions, many-to-one, logspace-reductions, etc. I only teach: (a) The general intuitive notion of a reduction; (b) Karp reductions as a special case. I omit all other "types" of reductions, and de-emphasize the notion of classifying reductions.

  I don't think that classifying reductions into formal types is valuable to students per-se, and view their role in the course as helping students digest the notion of reductions. Since my teaching style emphasizes abstracting intuitive thinking into formal models, over learning syntactic patterns, I prefer not to use classifications to help digest the notion.

References

[Aar11] Scott Aaronson. *Automata, Computability, and Complexity (MIT)*. https://ocw.mit.edu/courses/6-045j-automata-computability-and-complexity-spring-2011/pages/lecture-notes. 2011.

[AB09] Sanjeev Arora and Boaz Barak. *Computational complexity: A modern approach*. Cambridge University Press, Cambridge, 2009.

[Ale26] Gadi Aleksandrowicz. *Computability Theory (Technion)*. gadial.net/Documents/LectureNotes/Computability.pdf. 2026.

[Asp24] James Aspnes. *Computational Complexity Theory (Yale)*. https://www.cs.yale.edu/homes/aspnes/classes/468/notes.pdf. 2024.

[Coo18] Steve Cook. *463*. https://www.cs.toronto.edu/~sacook/csc463h/. 2018.

[C.S25] Karthik C.S. *Computability and Complexity Theory (Rutgers)*. https://cskarthikcs.github.io/Fall25A.html. 2025.

[Gol08] Oded Goldreich. *Computational Complexity: A Conceptual Perspective*. New York, NY, USA: Cambridge University Press, 2008.

[Hoz24] William Hoza. *Introduction to Complexity Theory (U Chicago)*. https://williamhoza.com/teaching/spring2024-intro-to-complexity. 2024.

[Kou23] Michal Koucký. "Automata and formal languages: shall we let them go?" In: *Bulletin of the European Association for Theoretical Computer Science EATCS* 140 (2023), pp. 80–87.

[Sip96] Michael Sipser. *Introduction to the Theory of Computation*. PWS Pub. Co, 1996.

[Soa16] Robert I. Soare. *Turing Computability: Theory and Applications*. Springer-Verlag, Berlin, 2016.

[6] Another reason for my mild aversion is that the "take-home" message is slightly disappointing: The multi-quantifier model generalizes $\mathcal{P}$ vs $\mathcal{NP}$, and the message is that the generalization behaves like the base case (i.e., we suspect that all levels are separated). It's nice, but the "bang for buck" isn't amazing.

[Tre05] Luca Trevisan. *Computability and Complexity (Stanford)*. https://theory.stanford.edu/~trevisan/cs172-05/. 2005.

[Vio26] Emanuele Viola. *Mathematics of the Impossible*. Cambridge University Press, 2026.

[Wat26] Thomas Watson. *Complexity in Computer Science*. Cambridge University Press, 2026.

[Wig19] Avi Wigderson. *Mathematics and computation*. A theory revolutionizing technology and science. Princeton University Press, Princeton, NJ, [2019] ©2019, pp. xiii+418. ISBN: 978-0-691-18913-0.

[Wig26a] Avi Wigderson. *Randomness*. https://www.math.ias.edu/avi/talks. 2026.

[Wig26b] Avi Wigderson. *The Value of Errors in Proofs*. https://www.math.ias.edu/avi/talks. 2026.