The Future of Computational Mathematics With AI Coding Agents

Alex Townsend *

30th December 2025

1 Introduction

I'm calling it right now. Computational mathematics is about to become much easier and, at the same time, much more demanding in 2026. The field is about to change because of AI coding agents.

Everyone is familiar with large language models such as ChatGPT, particularly in their role as aids to writing, exposition, and communication. Fewer, however, regularly use AI coding agents: systems designed to construct, modify, and test substantial pieces of code. Have you ever used one? Examples include Codex, GitHub Copilot, Cursor, and a growing collection of research-oriented agents integrated into scientific computing environments. Some require a modest subscription, others are freely available. In this article, I will not compare and contrast these tools; in my view, they are rapidly converging in features and functionality at a level of granularity where the differences are superficial for most academics.

Here, I also won't discuss the sheer number of AI agents now available, the pace at which they improve, or demonstrations in which substantial programs are produced in a matter of minutes. You can find this information online. I want to focus instead on the intellectual consequences for a computational mathematician. And I mean a computational mathematician in the broadest sense, as any mathematician who regularly writes code—in systems such as C, C++, Fortran, Julia, Lean, Magma, MATLAB, Python, `R`, Sage—to explore examples, test conjectures, perform numerical experiments, or support theoretical reasoning. By this inclusive definition, a pure mathematician is a computational one if they use software to explore conjectures or compute algebraic objects. We are a group of academics who use computers to simulate the mathematical world as an experimental laboratory.

My central claim is that AI coding agents should be used deliberately by all of us. That they will shift where our intellectual effort is spent as a field: away from implementation and toward experimentation and judgment, and that this shift should come along with an expectation of higher standards.

There is a close historical analogy. In the early days of digital computers, one could focus on how many arithmetic operations could be performed per second, or how long a calculation that once took weeks could now be completed in hours. However, the more profound effects laid in the kinds of questions that could be asked, in the willingness to explore analytically messy examples, and in the gradual reorganization of what we came to value.

Consider, for example, the pride once taken in extensive hand calculation. Gauss famously carried out prodigious numerical work himself. Throughout the nineteenth century, entire careers were built around the careful production of tables of logarithms, trigonometric functions, and special functions by human "computers." During the French Revolution, de Prony organized hundreds of clerks to compute logarithmic tables by hand; during the Manhattan Project, large teams were employed to carry out numerical calculations essential to physics and engineering. These efforts were central to scientific practice and highly valued.

Today, we compute such quantities on the fly, often without conscious thought. No mathematician regards the manual production of logarithm tables as a meaningful intellectual activity, nor do we feel

*The author is an Associate Professor of Mathematics at Cornell University. His email address is townsend@cornell.edu.

that something essential has been lost by abandoning them. Instead, the availability of reliable computation and high-level programming languages reshaped what we consider worth doing, allowed more people to compute, and shifted attention away from execution details and toward structure, interpretation, and design.

AI coding agents suggest that we may be at a similar juncture. They do not eliminate the need for mathematical insight, any more than early computers did. Instead, they render specific forms of work—once laborious—as routine. If history is any guide, this is not a loss. By shortening the distance between mathematical ideas and their computational realization, such tools may allow attention to shift more quickly toward questions of structure, robustness, and meaning. From this perspective, the sooner this transition occurs, the more rapidly computational mathematics can advance. Therefore, if you're a computational mathematician and haven't tried AI coding agents, take one out for a spin.

However, for computational mathematicians, AI raises several questions that are easy to overlook amid discussions of automation. When a substantial portion of implementation can be delegated to an agent, what remains our responsibility? How does this change the balance between conceptual design and technical execution? And, perhaps most importantly, does it clarify or obscure the location of genuine mathematical insight?

2 What Is an AI Coding Agent?

It is helpful to clarify what I mean by an AI coding agent, since the term is often used loosely. An AI coding agent is not simply a conversational interface where one asks for code, copies the output into a Jupyter notebook (say), encounters an error, and then returns to ask for a correction. I am sure many readers have tried this sort of workflow at least once; our students certainly have. That mode of interaction, while sometimes convenient, is not what I have in mind.

An AI coding agent is best understood as a system that operates inside an existing programming environment. It typically has access to the contents of a codebase, can read and modify multiple files, can run code, execute tests, inspect error messages, and iterate on its own output. In many cases, the agent runs directly within an integrated development environment or interacts with a project via a terminal interface. Crucially, it is embedded in the same workflow that computational mathematicians already use for serious software development.

AI coding agents are best used in a version-controlled repository with unit tests, documentation, and backups. One does not want an agent changing files without a reliable way to revert or inspect those changes. From a practical perspective, it is recommended that code produced by an agent be tracked by `git` in precisely the same way as in a medium-to-large collaborative software project. Changes can be reviewed, reverted, branched, compared, and discarded. In this sense, working with an AI agent closely resembles supervising a collaborator rather than issuing isolated commands to a machine.

What distinguishes an AI coding agent from earlier forms of assistance is its ability to take initiative across multiple steps. Given a partially specified goal, it can propose an implementation, run it, observe failures, modify the code, and try again. It can refactor existing routines, suggest alternative approaches, generate tests that reflect stated mathematical properties, and explore design variations without explicit instruction at each stage. In effect, it performs the kind of local computational experimentation that computational mathematicians routinely do themselves, but at a different scale and speed. It is not unusual for an agent to work autonomously for tens of minutes, producing and testing substantial amounts of code along the way.

AI coding agents matter for academia because they change how one interacts with a computer. Rather than issuing isolated commands and manually stitching together results, the computational mathematician increasingly acts as a supervisor. One specifies intent, constraints, and correctness criteria, and then inspects, critiques, and guides what the agent produces. The intellectual burden shifts away from

writing syntax and toward articulating structure, assumptions, and validation. It is this shift in responsibility and judgment, rather than raw speed, that makes AI coding agents consequential for computational mathematics.

3 Clearing Away a Misconception

Any discussion of a new computational paradigm attracts a familiar set of misunderstandings. Some arise from enthusiasm, others from skepticism, but many share a common feature: they focus on surface properties while missing the deeper intellectual shift. In order to examine the potential impact of AI coding agents on computational mathematics, I want to clear away a misconception that recurs with striking regularity and, I admit, always makes me grind my teeth. The misconception can be paraphrased as follows:

"AI will make our brains turn to mush."

This concern appears in many guises. Sometimes it is framed as a worry about intellectual atrophy, sometimes as a fear that students will no longer learn to think, and sometimes as the claim that delegating computation to machines must inevitably erode understanding. Rarely is it stated precisely, and rarely is it defended explicitly. Instead, it tends to function as a vague unease: a sense that something essential must be lost whenever a task is made easier.

At its core, this is a category error. It confuses the difficulty of execution with the difficulty of understanding. Writing code, like carrying out algebraic manipulations, is a form of procedural fluency. Knowing what should be computed, how results should be interpreted, and why a method works or fails is a form of conceptual authority. AI coding agents reduce the cost of execution, but they do not reduce the need for understanding. If anything, they raise the bar for it.

The unease surrounding AI coding agents often rests on the implicit belief that the bottleneck in computational mathematics lies in typing speed, syntactic fluency, or low-level coding ability. These are skills that academic mathematicians rarely regard as central in the first place. Seldom does a permanent academic position hinge primarily on authorship of a software project. The real bottlenecks in research software projects are conceptual: deciding what should be computed, how mathematical objects should be represented, which structures matter, and how results should be validated.

In my own work, the coding phase takes a long time, not because the final implementation is intrinsically difficult, but because I am experimenting in order to discover how something ought to be computed. Once I know what I want to compute and why, the final implementation is usually comparatively quick. AI coding agents are particularly effective at accelerating this exploratory phase, where most intellectual value is actually created. They allow tentative ideas to be turned into working experiments quickly, which makes it easier to abandon bad approaches and refine promising ones.

The same observation applies to large software projects that require careful testing. Writing unit tests is not intellectually deep, but maintaining them is laborious. Much of the effort goes into tracking down bugs, ensuring that changes do not break existing functionality, and verifying that numerical behavior remains consistent after refactoring or optimization. These tasks are essential, but most bugs are not big conceptual mistakes. AI coding agents can shoulder much of this burden, generating tests, running them repeatedly, and identifying failures quickly. This allows the mathematician to focus less on chasing errors and more on sharpening ideas.

What makes AI coding agents interesting to me, then, is not that they remove the effort of writing code, but that they externalize much of the experimental component of computational research. An algorithmic sketch, a half-formed conjecture, or a property one expects to hold can be elaborated into functioning code in a few minutes. In doing so, they shift the mathematician's effort away from implementation and toward specification. Far from making the work trivial, I find that this sharpens my thinking. I greatly benefit from reviewing, testing, and reflecting on what the agent produces.

However, one does have to be intimately part of

the process to gain understanding. I have already seen this effect quite clearly in the classroom. When I teach students how to use AI agents to code numerical methods or complete linear algebra tasks, the gap between the strongest students and the weakest, unfortunately, widens. The strongest students use the agent as a tool for exploration, probing why an approach works and where it fails. Weaker students are more likely to treat it as a shortcut, accepting output without interrogation. I find that AI agents amplify existing habits of thought rather than replace them.

For this reason, I suspect that world-class computational mathematics will increasingly be produced with the aid of AI coding agents. They will be part of how serious computational work is done—just as no one is writing a finite element method package in assembly language. At the same time, I worry that the noise will increase as it becomes faster to produce a seemingly reasonable, vaguely novel publication. I expect that AI agents will raise the mean quality of computational work, but they also raise its variance.

Part of the intellectual appeal of working with AI coding agents is that they can occasionally surface insights I would not have reached on my own, precisely because they lower the barrier between an idea and an experiment. Over the last few months, I have had several such experiences. In one case, while experimenting with the asymptotics of an orthogonal polynomial, I asked an AI coding agent to push a symbolic computation beyond what I considered reasonable. To my surprise, it produced two additional terms in the asymptotic expansion, using a guess-and-check approach that was then confirmed numerically. Without AI, my middle-aged brain would not have had the patience to type up the expression.

In another instance, I asked an agent to prototype a three-dimensional hierarchical Poincaré–Steklov scheme for solving partial differential equations on tetrahedral domains. The resulting implementation emerged over a few days, following dozens of prompts. More importantly, having a working prototype allowed me to experiment freely with different spectral methods, boundary decompositions, and coupling strategies, revealing which aspects of the method were essential and which were incidental. That clarity, without AI, would have been difficult to obtain without first enduring weeks of implementation work.

Perhaps most strikingly, while exploring numerical linear algebra routines, an AI coding agent inserted a comment suggesting a connection between the Cholesky algorithm on positive definite kernels and the so-called P-greedy algorithm—a relationship I was previously unaware of. A brief literature search revealed that the connection had been observed in a handful of manuscripts I had not previously encountered, which led me to a better understanding of the convergence behavior of the Cholesky algorithm.

Importantly, AI coding agents in their current form do not change where mathematical insight ultimately resides. They do not tell us which questions are interesting, which structures matter, or which results are worth trusting. We are doing research, and by definition, there isn't much training data at the frontier of human knowledge. Instead, what changes is the ease with which tentative ideas can be turned into concrete experiments and then interrogated. In this sense, AI coding agents do not replace the intellectual work of computational mathematics; they boost it.

4 Changing the Unit of Mathematical Work

The most significant impact of AI coding agents on computational mathematics will not be that they reduce the number of hours we work. I do not believe that. Instead, they will change the kind of work we do. In short, AI coding agents change the unit of mathematical effort. That shift will have far-reaching consequences for how we practice our profession.

Traditionally, a nontrivial fraction of a computational mathematician's time, especially as a graduate student or early-career researcher, is devoted to implementation, coding, and testing. This is often framed as a training ground: you write code to develop intuition about what works in theory and what works in practice. In reality, a great deal of this effort goes into discovering what should be computed in the first place. One frequently has to write substan-

tial amounts of code to determine which approach is worth taking seriously.

A concrete illustration comes from my own experience developing a software system with Nick Trefethen to approximate bivariate functions, which eventually became an extension of Chebfun. In the early stages of that project, several plausible algorithmic strategies were explored. Nick Hale initially implemented a tensor-product Chebyshev approach. It worked in the sense that it produced reasonable approximations, but it was not conceptually deep. Still, any serious method worth its salt ought to outperform naive tensor products in a meaningful way.

Cécile Piret and I then investigated radial basis functions, motivated by their flexibility and success in other approximation settings. However, for two-dimensional problems on regular domains without scattered data, radial basis functions are not operating in their most natural regime. Different ideas were floated. Paul Constantine suggested active subspaces. We thought about sparse grids and Padua points. Each of these approaches had some merit, and each required implementation before it could be judged.

Ultimately, the strategy that proved both effective and conceptually satisfying was based on low-rank approximation using adaptive cross approximation. This allowed us to exploit and extend well-established one-dimensional ideas developed by Battles, Trefethen, and others. What matters here is not the specific outcome, but the process. At the time, each candidate idea had to be implemented before it could be evaluated. The cost of experimentation was high, limiting how widely we could explore. In retrospect, we examined only a small subset of the approaches that might have been considered. We implemented adaptive cross approximation and then, somewhat unfairly, perhaps, retroactively defended the decision.

With AI coding agents, expectations should change. If one understands the underlying mathematical ideas, then producing a tensor-product Chebyshev implementation, a radial basis function prototype, or a sparse grid method is now only a few prompts away. The intellectual bottleneck no longer lies in translating ideas into code, but in deciding how to judge an approach's success.

Put differently, the unit of work shifts. Where once the natural unit was a command or a script, it can now become an experiment, a comparison, or a hypothesis. Implementation recedes into the background, and specification comes to the fore. In practice, this means that computational mathematicians will spend less time asking whether it is worth coding something up. It will also allow us to become more ambitious in what we implement.

This shift has practical consequences. I no longer worry much about writing parallel code by hand or squeezing performance out of inner loops; that is often only a few prompts away. I do not code fluently in `R`, yet with modest effort I can translate a Python implementation, adapt it to local conventions, and have it pass the same tests. Refactoring code, adding comments, fixing minor bugs, writing documentation, or reorganizing a project to make its structure clearer are similarly easy to delegate.

As a result, our cognitive load moves away from managing technical details and toward deciding what properties the code should satisfy and how success should be measured. We can focus more on designing experiments that illuminate mathematical structure, on seeking out counterexamples, and on comparing competing ideas across regimes. In practice, this encourages broader exploration, faster backtracking, and a greater willingness to abandon ideas that do not pan out.

In this sense, AI coding agents reduce the friction associated with exploration. The result, at least for me over the past few months, has been a style of computational mathematics that is unusually enjoyable. Implementation serves inquiry more directly, with a much tighter feedback loop. The intrinsic value of routine programming plummets, and the value of good ideas rises. That is a trade I am happy to make.

5 Impact on Computational Mathematics

AI coding agents will not merely accelerate existing workflows; they will reshape expectations in compu-

tational mathematics. Their impact will be felt along several concrete dimensions.

- **Raising the bar for computational evidence.** As implementing and comparing numerical methods becomes easier, superficial comparisons become harder to justify. Claims supported by only a narrow slice of experiments should become less acceptable, especially in papers that make performance or robustness claims. We, as a broad community, should expect more extensive sweeps across parameter regimes, more careful stress testing, and more systematic comparisons across methods. I should add that while it is easy to get an AI agent to implement a well-known standard method, one has to be extremely clear and precise for it to successful implement a novel algorithm. To me that's fine, we should be comparing most frequently against the well-established methods.
- **Lowering the barrier to serious experimentation.** Computational exploration becomes accessible to a broader range of mathematicians. For busy faculty members who no longer have the time or inclination to keep up with evolving software practices, AI coding agents reduce the friction associated with experimentation. For pure mathematicians who have historically coded less, they enable the implementation of surprisingly sophisticated numerical experiments. Computational exploration should become more mainstream, and that can only strengthen the discipline.
- **Shifting effort from writing code to interrogating it.** Reading, modifying, and extending existing code is often more informative than writing it from scratch. AI coding agents excel at supporting this mode of engagement. As a result, computational work can focus more on interpretation, comparison, and failure analysis rather than on technical construction alone. The interesting questions move from "does it run?" to "when does it break?"
- **Weakening linguistic constraints.** Many computational mathematicians effectively work in one or two programming languages over the course of their careers. The cost of switching languages is real and quietly shapes which tools and communities one engages with. AI coding agents weaken this constraint by making it easier to translate ideas across languages, adapt them to different ecosystems, and exploit language-specific features. This encourages broader cross-pollination of numerical ideas.
- **Blurring the boundary between computation and proof.** There is an intriguing interaction with formal methods. Projects such as `Lean` have already altered how some mathematicians think about proof and verification. AI coding agents suggest a hybrid workflow in which a theorem statement serves as a test, and a proof attempt becomes an iterative process guided by failures. Whether AI coding agents will ultimately prove beneficial for proving theorems remains to be seen. Still, with the right error messages and the correct training data, one can imagine a machine trying a million proof strategies overnight. I could envision that some theorems could be proved by established techniques this way quite soon.

Taken together, I believe that AI coding agents will make the strongest researchers even stronger, and weaker ones weaker. This is not a moral judgment; tools that lower friction amplify differences in taste, judgment, and intellectual discipline. By reducing technical overhead, AI coding agents allow computational mathematics to lean more fully into what it already does best: using computation as a lens through which mathematical structure comes into sharper focus. In turn, we must learn how to use our new pair of binoculars. These are exciting times for computational mathematics.

6 Conclusion

At present, I am broadly optimistic. Everything I see suggests that AI coding agents represent genuine progress for computational mathematics and that their immediate effects are overwhelmingly positive. They lower the cost of exploration, raise expectations for computational evidence, and allow more of our intellectual effort to be devoted to structure, interpretation, and insight rather than to technical execution. Used well, they make it easier to think clearly.

That said, it would be disingenuous to claim that I have no concerns. We may be living in a particularly fortunate period, one in which mathematical ideas can be explored and tested with unprecedented ease. At the same time, the responsibility for insight still unmistakably rests with us. At present, AI systems do not originate genuinely deep mathematical ideas or implement truly novel algorithms; at least I have seen no evidence of that. They recombine, elaborate, and accelerate, but they still rely on human judgment to decide what is interesting, what is correct, and what is worth pursuing. For now, the partnership is productive.

It is natural to wonder how long this balance will last. We may be in a middle ground, where human intuition and machine assistance complement one another unusually well. It is conceivable that we may become so far removed from the underlying mathematical structures that the nature of mathematical work changes again, in ways that are difficult to anticipate, and not readily accessible to human insight. Whether or not that happens, I believe it lies beyond the horizon of our current practice.

For the foreseeable future, however, AI coding agents give us the chance to raise our standards, broaden our explorations, and deepen our understanding. In their current form, if we use them thoughtfully, they will not diminish mathematical insight, but amplify it. This seems like a moment worth embracing: a time to experiment more boldly, to demand more from our computations, and to enjoy the beautiful alignment of powerful tools with human curiosity and judgment.