Six Opportunities for Scientists and Engineers to Learn Programming Using AI Tools such as ChatGPT

Philip J. Guo[1]

[1]Affiliation not available

August 21, 2023

Abstract

This article demonstrates how scientists and engineers can use modern AI (artificial intelligence) tools such as ChatGPT and GitHub Copilot to learn computer programming skills that are relevant for their jobs. It begins by summarizing common ways that AI tools can already help people learn programming in general. Then it presents six learning opportunities that are catered to the needs of scientists and engineers, including using AI tools to 1) create customized programming tutorials for one's own domain of work, 2) learn complex data visualization libraries, 3) learn to refactor exploratory code into more maintainable software, 4) learn about inherited legacy code, 5) learn new programming languages on-demand within the context of one's workflow, and 6) question the assumptions that one's scientific code is making. Taken together, these opportunities point toward a future where AI can help scientists and engineers learn programming on-demand within the context of their existing real-world workflows.

Introduction

Over the past year (2022-2023) technology companies have released a staggering variety of AI (Artificial Intelligence) tools that can generate text, code, images, music, and even video clips on-demand. Lately it feels like new AI tools are coming out every week, and it is impossible to keep up with the latest buzzwords and marketing slogans. In this article I want to zoom in on one popular kind of AI tool that is especially relevant for scientists and engineers: Large Language Model (LLM)-based tools such as ChatGPT and GitHub Copilot that take text (or code) as input from the user and generate text (or code) in response. Specifically, I will present six opportunities for scientists and engineers to use these LLM-based AI tools to *learn* computer programming skills that are relevant for their jobs.

As these AI tools have grown more popular, educators have started writing about ways to use them to teach and learn programming. However, most writing on this topic so far has been about using AI (especially within computer science departments) to teach students who aim to become full-time programmers (Becker et al., 2023; Lau & Guo, 2023; Denny et al., 2023). Here I want to provide a complementary perspective by addressing the unique needs of scientists and engineers who do *not* intend to become professional software developers. For instance, a climate scientist may need to pick up a bit of Python or R to analyze data for their research; or a mechanical engineer may learn command-line scripting on embedded Linux systems because they are rigging together hardware components for a measurement device. This article presents six ways that AI tools can help them learn what they need directly within the context of their existing real-world workflows (Guo, 2023).

Background: Frequently-Mentioned Thoughts about Using AI Tools to Learn Programming

Before focusing on scientists and engineers in particular, I will first set the stage by summarizing what others have already mentioned about using AI to learn programming in general. I distilled these sentiments from several overview papers (Becker et al., 2023; Lau & Guo, 2023; Denny et al., 2023), which each cite more detailed research studies in their respective bibliographies.

As of around November 2023, only a year after ChatGPT was released (OpenAI, 2022), here are some widely-acknowledged capabilities of AI tools for learning programming:

- AI tools can solve many kinds of programming exercises that are now used as homework and exam questions in introductory computer science courses. As a result, some instructors are worried about students using them to cheat and are reconsidering what kinds of homework assignments and exams to give in the future.
- On a more constructive note, since AI can generate a variety of different solutions for a programming exercise, those can serve as *worked examples* (also known as *worked-out examples (Renkl, 2005)*) that students can use to learn different approaches to solving a problem. Seeing different *variations* of solutions can help students learn better (Thuné & Eckerdal, 2009).
- AI can automatically generate a wide variety of programming exercises to meet a given pedagogical goal (e.g., teaching how to join multiple data tables in Python using the pandas library). This capability can help instructors to prepare assignments more efficiently and give students extra practice opportunities on-demand.
- AI can explain what a piece of code does step-by-step in a novice-friendly way. This capability can benefit students who may feel embarrassed to ask someone for help on a seemingly 'simple' question; they can now freely ask the AI without fear of being judged.
- AI can help debug students' code, which can enable them to make progress on their homework without feeling self-conscious about asking someone for help.
- AI can automatically write tests for students' code, which can help them spot more bugs.
- AI can perform *code reviews* to give students feedback on their coding style and aesthetics.

And here are some frequently-mentioned limitations of these tools:

- First and foremost, AI tools can generate code that is incorrect, buggy, insecure, or violates other known best practices. Moreover, students may have trouble spotting the bugs in AI-generated code since it often appears to be well-written.
- Relatedly, since AI-generated code looks convincing at first glance, students may grow *over-reliant* (Passi & Vorvoreanu, 2022) and reflexively copy-paste it into their projects without questioning whether it is correct or not. As a result, instructors have emphasized that it is critical for students to learn to write test cases for AI-generated code (Lau & Guo, 2023).
- Even if AI-generated code is correct and of high-quality, the fact remains that the student did not write that code themselves. This makes it harder for them to understand what the code does and to make future updates to it.
- AI tools are optimized for "doing" rather than teaching. This means when a student asks the AI a question, it will directly give them an answer. While this can be convenient, it may hinder learning. In contrast, a good human tutor would teach the underlying concepts and gradually guide the student to solve the problem on their own (with well-timed hints along the way) instead of giving them the answer right away.
- It is hard for novices to develop a mental model of what these AI tools are and are not capable of, since how they work is mysterious even to experts. Thus, some students may get frustrated that AI cannot help them with seemingly-straightforward requests, when in reality those requests appear vague or unclear to the AI tool. In practice one must learn to become good at *prompt engineering (Denny et al., 2023)* to be able to write prompts (i.e., textual requests to the AI tool) that can consistently elicit

high-quality responses.
- The code or explanations that AI tools produce by default may be too complex for novices to understand, since AI mimics the style of code found on the internet and is not specifically tuned to be beginner-friendly.
- AI tools may generate text that reinforces existing social biases, contains toxic content, or uses copyrighted materials without the original creators' permission.

As you read about the opportunities below for AI to help scientists and engineers learn programming, please also keep the above limitations in mind.

Opportunity 1: Using AI to Create Customized Programming Tutorials for Your Own Domain

The opportunity I am most excited about is using AI tools to create programming tutorials that are customized for a scientist or engineer's own domain of work. Why is this significant? Because scientists and engineers often learn programming to analyze their own data, but existing tutorials all use *generic* datasets such as the popular 'iris' or 'mtcars' data that come pre-installed with the R programming language. The 'iris' dataset describes 150 samples of iris flowers with features such as their sepal and petal dimensions; and 'mtcars' catalogs information about 32 car models from the early 1970s, with features such as their number of carburetors, horsepower, and rear axle ratio. Unless you happen to be a flower or car enthusiast, chances are that you do not care at all about this data, so learning programming using these datasets may not be that motivating for you.

Instead of reading tutorials that use generic datasets, you can now use an AI tool like ChatGPT to generate customized tutorials with example data from your own domain of work. No matter if you are a geologist or astrophysicist or structural engineer, AI tools possess enough knowledge about the basics of your field to be able to generate synthetic data and code examples that you can relate to more than 'iris' or 'mtcars.' For instance, if you are learning how to do multiple linear regression in MATLAB, it can be far more relatable to see examples using data from your own domain (e.g., geology) rather than, say, predicting car horsepower from rear axle ratios. You can even paste in real datasets from your own research (e.g., a neuroscience study) and have the AI tool generate a programming tutorial based on your data. Learning programming using authentic data in a domain you personally care about can make that knowledge stick better than reading generic tutorials (Kjelvik & Schultheis, 2019). For more details, I walk through an example of this idea in the "Intermission 1: ChatGPT as a Personalized Tutor" section of my *Real-Real-World Programming with ChatGPT* article (Guo, 2023).

Opportunity 2: Using AI to Learn Complex Data Visualization Libraries

A staple of scientific programming is writing code to produce data visualizations ranging from simple bar charts all the way to interactive multi-scale dynamic diagrams. Scientists face an inevitable tradeoff here – they can either a) use a point-and-click interface like Excel or Google Sheets, or b) write custom Python/R/JavaScript/MATLAB/etc. code using libraries such as Matplotlib, Seaborn, ggplot2, Bokeh, Plotly, Altair, or D3.js. The former is easier to learn but offers less expressive power, while the latter is more expressive but harder to learn.

AI tools can lower the barriers to learning the latter category of data visualization libraries. Similar to Opportunity 1, you can use AI to generate tutorials for how to use these libraries *within the context of the data you are currently working with*. For instance, let's say you are a marine biologist wanting to make a scatterplot to correlate observations of different types of fish, and you want the data points to vary in color,

shape, and size according to certain fishy properties. Even though you may not know how to write the exact code to do so, you have a clear vision of what the output should look like. By expressing this request to an AI tool, it can both generate the data visualization code and add inline comments to teach you how that code works step-by-step.

Using AI to write data visualization code for you can be effective since it is something that can be hard for humans to do but easier for humans to *verify*. Let's face it – not even seasoned programmers remember how to write Matplotlib, ggplot2, or other visualization code from scratch. These complex libraries contain hundreds of different functions, each with a heap of different parameters that interact in idiosyncratic ways. It's a waste of our human brainpower to memorize all these mundane details, but AI is great at "remembering" these details for us. Since we have an intuitive sense of what the output visualization should look like, we can verify whether the AI-generated code looks more-or-less correct and make adjustments if needed. This ease of human verification gets around a core limitation of AI tools, which is that they might generate incorrect code or output. However, note that looking at the visualization alone may not be enough to fully verify correctness, so it is still important to inspect the AI-generated code to make sure that its logic makes sense.

Opportunity 3: Learning to Refactor Exploratory Code into More Maintainable Software

Scientific programming is often exploratory in nature and done in a mix of creatively-named files (e.g., MY_-ANALYSIS_SCRIPT_v2_param53_final_FINAL.py) and computational (e.g., Jupyter or R) notebooks. The main goal, especially during early stages of research, is to iterate quickly to explore and refine hypotheses, not to produce clean maintainable code. But if these initial explorations are successful, then inevitably this draft code ends up living far longer than originally intended. So one big challenge for scientists and engineers is learning to *refactor* this prototype code into something more maintainable longer-term.

Refactoring is a software engineering technique where a programmer revises code to be more clear and maintainable while still maintaining the same functionality. AI tools can help you here by inspecting your code and suggesting refactoring opportunities such as creating more descriptive variable names, encapsulating common snippets into their own functions, making indentation and spacing more consistent, and adding inline comments to describe what each part of your code does. By seeing how AI refactors your code, you can learn habits that you can apply in the future. In this way, AI plays the role of a senior colleague who demonstrates best practices within the context of your own codebase. These in-situ, context-specific lessons can stick better than if you had read a general guide to code refactoring.

Similar to Opportunity 2, this can be a great use case for AI since it is relatively easy for a human to verify whether the output is correct. Since you already have code that works, the AI-refactored code should behave the exact same way when you run it. You can look at the old and new versions side-by-side and run both to give you confidence that the AI worked as intended. Using AI to refactor can be less risky than using it to write brand-new code.

Opportunity 4: Learning about Inherited Legacy Code

Scientists often inherit code from former lab members who have graduated or move onto new jobs. As mentioned above, ideally everyone would take the time to refactor their exploratory code into something more maintainable. But in reality lots of old code is hard to understand since it may have been duct-taped together in a hurry to get experiments working for a paper submission deadline. And even if the original authors intended to clean it up and document it well, they are always under pressure to move onto the next project, aim for the next publication or grant deadline, and so on. Plus, we as scientists are not professional

software developers, so we may lack the expertise to follow industry best practices for code quality even if we have the best of intentions. The end result is that inherited code (formally called *legacy code (Feathers, 2004)*) can be very hard to understand and work with, which slows down scientific progress.

AI tools can help here by automatically inspecting a pile of old legacy code and generating step-by-step explanations, clarifying code comments, and supplemental documentation to summarize how that code works. These explanations may not be 100% accurate, but they can serve as a starting point for human investigation. Think of the AI here as an intrepid explorer buddy who can help you out when spelunking into a deep cave of unexplored legacy code. By working alongside the AI when exploring an unknown codebase, you can learn both how to work with it specifically and also more general skills for how to effectively deal with legacy code in the future.

Opportunity 5: Learning New Programming Languages On-Demand Within the Context of Your Workflow

Scientists and engineers may have to learn a new programming language on short notice if an important library they need is available only in that language. Since they need to get their job done efficiently, they cannot easily put their main work on hold in order to take a formal course or work through a textbook. Instead, it would be much more convenient to be able to learn just-in-time and on-demand within the context of their own existing workflow. AI tools can facilitate this type of learning in two ways:

1. The scientist can write code for their task in the language they are most comfortable with (e.g., Python) and then use an AI tool to *automatically translate* it into the language they want to learn (e.g., Julia). While this translation is by no means perfect, it is likely "good enough" to show the correspondences between the two languages (e.g., which Python features map to which Julia features). This way, someone can learn a bit of Julia directly within the context of a piece of Python code that they are familiar with.
2. Going the other way, a scientist can find a piece of example code in an unfamiliar language (e.g., Julia) and then use an AI tool to translate it back into a language they already know well (e.g, Python). This can come in handy if, say, that piece of example code implements an important algorithm that they need for their research, but they do not understand how it works since they are not familiar with the language it is written in.

Opportunity 6: Questioning the Assumptions Your Scientific Code is Making

One of the most challenging aspects of writing scientific code is making sure the assumptions that underlie your code are well-justified. Even if you implement the most elegant, efficient, and bug-free algorithm to run on your data, if that is not the appropriate algorithm to use, then your code is still useless (or may even be harmful if it gives misleading or biased results). However, it is impossible for existing code analysis tools to tell whether your code may be making incorrect assumptions since these tools do not know anything about the underlying scientific or engineering questions your code is trying to address. Modern AI tools have the potential to overcome this limitation via a clever rhetorical trick: by asking *you* whether the assumptions you are making are sound and having you come up with answers on your own.

AI tools still cannot do your science for you, but what they can do is serve as a skeptical inquisitor to question the assumptions your scientific code is making. For instance, if you are a geneticist writing scripts using the Biopython library to process a specific type of gene sequencing data, an AI tool may know enough about this domain to ask skeptical questions about why you decided to, say, run a specific parametric statistical test, and whether that test is justified given the properties of the data set you are using. Or the tool can

question you about why you decided to use linear instead of logistic regression when your outcome variable seems to be binary. The AI does not necessarily know the correct answer to those questions, but it likely knows "enough" about genetics and statistical tests to pose these questions for you to reflect on. This use case is like having the AI serve as a sort of Socratic tutor (Al-Hossami et al., 2023) to get you to introspect more deeply on your thought process.

Parting Thoughts and Call to Action

In summary, AI tools like ChatGPT offer unique potential for scientists and engineers to learn programming, while also aiding in various aspects of their work. These tools can generate personalized programming tutorials, facilitate learning of complex data visualization libraries, suggest refactoring of exploratory code, and assist in understanding inherited legacy code. Moreover, they may help in learning new programming languages in the context of work and encourage the questioning of assumptions in scientific code. However, it's essential to maintain a balanced perspective and acknowledge the limitations of these AI tools. They are only as good as their training data, can sometimes produce incorrect output, and, importantly, lack true comprehension of context and domain-specific knowledge. They should be seen as aides that can help streamline certain tasks, but are not replacements for the essential expertise, judgment, and creative problem-solving abilities of human scientists and engineers. Therefore, I encourage the scientific community to explore these tools with a critical eye, understanding their strengths and limitations. Use them where they can genuinely add value and continue relying on human intellect and insight where it matters most. In doing so, we can use technology to enhance our work without over-relying on it, striking a beneficial balance in this era of digital advancement.

I promise this prior paragraph is the only one in this entire article that was written by an AI tool (ChatGPT with GPT-4). I asked ChatGPT to summarize what I wrote, and the original paragraph it generated was too overenthusiastic about the benefits of AI (very self-serving of it!). So then I instructed ChatGPT to "make it more balanced and less pro-AI," and it generated the more nuanced response that you just read. Although it is a well-written summary, its style doesn't really match my own. So at least for now I will still be doing my own writing without AI assistance, even though I do use AI to help me in programming (Guo, 2023). It's hard to predict how these tools will evolve in the coming years, but hopefully the ideas I presented here can serve as starting points for you to learn more about this fast-changing topic.

Acknowledgments

Thanks to Lorena Barba for encouraging me to write this article, Shannon Ellis for helping me brainstorm ideas for it, and Ashley Juavinett for feedback. This material is based upon work supported by the National Science Foundation under Grant No. NSF IIS-1845900.

References

Programming Is Hard - Or at Least It Used to Be: Educational Opportunities and Challenges of AI Code Generation. (2023, March). *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1*. https://doi.org/10.1145/3545945.3569759

From "Ban It Till We Understand It" to "Resistance is Futile": How University Programming Instructors Plan to Adapt as More Students Use AI Code Generation and Explanation Tools such as ChatGPT and GitHub Copilot. (2023). *ACM Conference on International Computing Education Research (ICER)*.

Computing Education in the Era of Generative AI. (2023). *Communications of the ACM*.

Real-Real-World Programming with ChatGPT: Taking AI Far Beyond Small Self-Contained Coding Tasks. (2023). *O'Reilly Radar*. https://www.oreilly.com/radar/real-real-world-programming-with-chatgpt/

Introducing ChatGPT. (2022). https://openai.com/blog/chatgpt. https://openai.com/blog/chatgpt

The Worked-Out Examples Principle in Multimedia Learning. (2005). In *The Cambridge Handbook of Multimedia Learning* (pp. 229–246). Cambridge University Press. https://doi.org/10.1017/cbo9780511816819.016

Variation theory applied to students' conceptions of computer programming. (2009). *European Journal of Engineering Education*, *34*(4), 339–347. https://doi.org/10.1080/03043790902989374

Overreliance on AI: Literature Review (Microsoft Research Technical Report). (2022). Microsoft Research.

Conversing with Copilot: Exploring Prompt Engineering for Solving CS1 Problems Using Natural Language. (2023, March). *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1*. https://doi.org/10.1145/3545945.3569823

Gardner, S. (Ed.). (2019). Getting Messy with Authentic Data: Exploring the Potential of Using Data from Scientific Research to Support Student Data Literacy. *CBELife Sciences Education*, *18*(2), es2. https://doi.org/10.1187/cbe.18-02-0023

Working Effectively with Legacy Code. (2004). Prentice Hall Professional Technical Reference.

Socratic Questioning of Novice Debuggers: A Benchmark Dataset and Preliminary Evaluations. (2023). *Proceedings of the 18th Workshop on Innovative Use of NLP for Building Educational Applications (BEA 2023)*. https://doi.org/10.18653/v1/2023.bea-1.57