Participatory live coding seen from a learning theory perspective

Lex Nederbragt, januar 2020. Utviklingsnotat for kurset i universitetspedagogikk, vår-høst 2019. Skrevet på Engelsk siden jeg ønsker å dele det med internasjonale kolleger på sikt.

Aim of this essay

When I teach programming, I use a technique called participatory live coding. In my experience, this technique works very well, both with graduate students and other researchers, as with my undergraduate bioscience students, to enhance learning. However, it is not yet clear to me whether there is any theoretical framework that would support the fact that participatory live coding benefits learning to program. With this essay, I intend to investigate this aspect. I will investigate to what extent placing the method in a cognitive psychological theoretical context, and analyzing relevant empirical research, can help explain its success. I will end with some possible research questions that could are relevant in order to support the claim that participatory live coding under certain conditions can benefit learning programming.

Participatory live coding in programming education

Participatory live coding is a technique where a teacher (or instructor) does live programming, with the students copying and executing the exact code or commands that are being written by the teacher. The instructor reads what is being typed out loud, explaining the different elements and principles. Teacher and students all execute the commands or program, leading to an immediate evaluation of the results - hence the term ‘participatory’. Crucially, the session contains regular, often short, exercises, where students are asked to solve a small relevant problem on their own or in pairs or small groups.
This approach works better than lecturing about programming, or relying on students reading a textbook or compendium. What is taught is immediately applied and students’ questions arising from it can immediately be addressed. The thought process behind the actual programming can be conferred as well, as students also see *how* the teacher is developing their code. It slows the teacher down relative to using slides to show the concepts and code, giving students more time to actively engage with the material before moving on to the next concept. Exercises enable immediate practice using the material.
Participatory live coding for teaching programming should not be confused with Live coding used to demonstrate software (for example, at a conference, with an audience passively observing), or used as a form of performing art (e.g. while creating computer music[1]).

I have learned the technique of participatory live coding through my involvement in Software Carpentry and Data Carpentry (together with Library Carpentry, collectively organised as a

[1] For example, see https://www.cambridge.org/core/journals/organised-sound/article/live-coding-in-laptop-performance/08F42B84BBCA427C345030481A3DDA0D

global non-profit called “The Carpentries”, https://carpentries.org). Software Carpentry “teaches researchers the computing skills they need to get more done in less time and with less pain” and is mostly aimed at researchers at the PhD and postdoc level. Participatory live coding is the main method of teaching in the two-day workshops, and it is part of the training and assessment for becoming a certified instructor[2] (see also Wilson, 2019).

In 2017 I started to apply participatory live coding to undergraduate teaching in my course “Introduction to Computational Modelling in the Biosciences” (BIOS1100) at the Bioscience program at the University of Oslo. Students of the Bioscience study program take this non-elective course in their first semester. Learning programing is done in the context of solving relevant biological problems using simple mathematical models implemented in the programming language Python. Participatory live coding is used as the main teaching method to introduce programming concepts and for solving exercises during the group sessions. Students report great satisfaction with the technique, and it appears to lead to good understanding of the basic programming principles, although this has not been quantified. I also have the impression that for students who are not convinced they can learn programming, or see it as something very difficult, introducing programming using participatory live coding helps them to overcome this fixed mindset (‘programming is not for me’) and establishes a growth mindset (‘with effort, I can learn programming’) instead (Dweck, 2006).

Empirical work on live coding in teaching programming

Research studies into live coding in teaching programming to novice students often refer to live coding as a technique to replace static lectures or code examples. However, the technique described in these studies is almost exclusively of a non-participatory nature, i.e. demonstrating building a working program without the students coding along. A recent study (Raj et al., 2018), used live-coding during lectures giving the students the option of coding along. The feedback from the students indicated that they would actually have preferred to have to code along to make the sessions more useful. Other studies (summarised in the Raj et al. article) concluded that students preferred seeing code being produced live rather than handed out as a final product. No study has so far been able to provide clear evidence of a benefit of live coding towards the learning the material, although one study found that “teaching via [non-participatory] live-coding is as good as if not better than using static code examples” (Rubin, 2013).

Learning to program is difficult

Learning to program is generally considered difficult (Guzdial, 2015; Jenkins, 2002; Robins et al., 2003). However, for a counterpoint (and an excellent summary of the research supporting this position) see (Luxton-Reilly, 2016). Learning a programming language is a bit like learning a new written language: there is not only the learning of the syntax, but also of how to combine words to give meaningful sentences (the ‘grammar’). Mistakes lead to error messages which are not written for novices (Denny et al., 2014). Code that runs but does not give the expected or correct output, i.e., has ‘bugs’, is hard to correct - debugging is a

[2] https://carpentries.github.io/instructor-training/14-live/index.html

skill in itself (McCauley et al., 2008). Asking a student to ‘write a computer program that does X’ from scratch requires many skills at the same time: knowing which coding constructs and data structures to use, being able to correctly apply the computer programming languages syntax, being able to design and test the program, ensuring that the solution actually solves the problem being addressed (i.e., ‘X’), for a problem that may require knowledge from a specific domain (e.g, biology in the case of my BIOS1100 students). It is safe to say that it is easy to arrive at a situation where the amount of information that needs to processed simultaneously is too much for a novice to cope with. In other words, learning to program quickly leads to what in Cognitive psychology is called high *cognitive load*.

Relevant cognitive psychology research

Cognitive load theory

Learning theory starts from acknowledging that humans have two memory systems: *working memory* and *long-term memory*. Working memory is where new information is processed, and coupled to pre-existing information present in long-term memory. It is said that learning happens if this new information is transferred to long-term memory. While long-term memory can contain vast amounts of information, working memory is considered small, and has a capacity of about “seven plus or minus two” pieces of information.
Cognitive load theory is “a set of learning principles that deals with the optimal usage of the working memory” (Caspersen and Bennedsen, 2007). This theory, as defined in a recent review on the subject (Sweller et al., 2019), “aims to explain how the information processing load induced by learning tasks can affect students’ ability to process new information and to construct knowledge in long-term memory”. The theory argues that the limited capacity of working memory severely restricts how much new information can be processed at any one time. When too much is asked from this working memory, there is a risk of overloading it, hampering learning. Overloading working memory prevents new knowledge to be transferred effectively to long-term memory, which is required for learning. It is argued that instructional methods need to take these limits into account.

Cognitive psychology researchers distinguish between three categories of cognitive load

- *Germane load* is “a non-intrinsic cognitive load that contributes to, rather than interferes with, learning” (Caspersen and Bennedsen, 2007). It is related to the desirable mental effort to achieve learning.
- *Intrinsic load* is “cognitive load intrinsic to the problem that cannot be reduced without reducing understanding” (Caspersen and Bennedsen, 2007). students cannot avoid this load, but it is related to students’ prior knowledge and it can be kept at a lower level by decreasing the complexity of the material to be learned.
- *Extraneous load* is “caused by instructional procedures that interfere with, rather than contribute to, learning” (Caspersen and Bennedsen, 2007). This load distracts from learning, it uses up working memory for tasks not contributing to learning.

In this framework, optimal learning happens when extraneous load is minimized, and germane load is maximized, while carefully managing intrinsic load.

Techniques for reducing cognitive load

There are a series so-called cognitive load *effects* reported in the literature.These effects result from experimental studies aiming to reduce cognitive load during instruction. Those that are relevant for discussing participatory live coding are briefly described here (Sweller et al., 2019):

- **The worked examples effect:** “Worked examples provide a full problem solution that learners must carefully study” (Sweller et al., 2019). Worked examples can be considered a form of learning from examples. Once the solution to a problem has been worked out by demonstration, students are asked to solve similar problems on their own.
- **The completion problem effect:** rather than providing a complete solution, a partial solution is given which students need to complete. This forces students to spend more time on studying the problem and its partial solution.
- **Split-attention effect:** information can come from different sources, for example audio and visual. Paying attention to different streams of information requires mental effort and thus may increase cognitive load. “Learners must mentally integrate the two sources of information in order to understand the solution, a process that yields a high cognitive load and hampers learning” (Sweller et al., 2019). An example is showing a slide with much information (e.g., a long quote) but verbally conveying other information, distracting the listener. Ensuring that the information on all relevant channels is redundant can alleviate the problem.

Direct instruction

A 2006 article whose title started with “Why Minimal Guidance During Instruction Does Not Work” (Kirschner et al., 2006) argues that educational research supports the need for guided instruction, especially when students are just started learning a subject. The opposite approach, asking novices to learn by discovery and constructing their knowledge themselves (variously called Constructivist, Discovery, Problem-Based, Experiential, or Inquiry-Based Teaching), has been shown by empirical studies to be less efficient. Throwing a problem at a student without any guidance increases cognitive load, and can lead to the load being too high for effective learning (Anthony R., Jr., 2008).
Direct instructional guidance is defined as “providing information that fully explains the concepts and procedures that students are required to learn as well as learning strategy support that is compatible with human cognitive architecture“ (Kirschner et al., 2006). Guided instruction provides techniques that are argued to reduce cognitive load, thus enhancing learning. The proponents of direct instruction over minimal guidance root their arguments solidly in Cognitive Load Theory. Asking a student to ‘write a computer program that does X’ from scratch would thus be a good example of asking students to learn by discovery.

Cognitive load in programming education

As described above, learning to program quickly leads to high cognitive load. For example,

- students need to be aware of strict syntax requirements, e.g. they need to pair brackets ‘(‘ and ‘)’, a case of intrinsic load

- students have to understand how to work with the coding environment, an example of germane load
- I once made the mistake of introducing a second, quite similar, but sufficiently different way of using a data structure too soon in my course, leading to a lot of confusion. This can be considered an example of extraneous load, as students needed to keep two ways of doing the same thing in their head at the same time, even though they did not need the second way to solve their problems until much later in the course.

Although few studies measure cognitive load during learning how to program directly (e.g., Morrison et al., 2014), many attempt to devise interventions that reduce cognitive load. Some of these interventions can directly be described in terms of the previously mentioned 'effects' for reducing cognitive load.

Worked examples: In a review on the subject, the authors state that they "believe that the use of worked examples to demonstrate problem solving and software development is a signature pedagogy for Computer Science" (Skudder and Luxton-Reilly, 2014). They conclude, based on the (limited) existing research on the topic, that certain types of worked examples are more successful than others:

- example-problem pairs, where each worked example is paired with a similar problem for the students to solve immediately afterwards
- faded working examples, where after presenting a worked out complete solution to a problem, problems are presented with successively more steps missing that the students are asked to solve, ending with a problem fully to be solved by the students (an example of the completion problem effect)

Parson's problems: providing a complete program, but randomising the order of the lines/instructions, optionally adding unnecessary lines of code to solve the problem (Parsons and Haden, 2006).

Participatory live coding as a means to reduce cognitive load

As argued above, for many novices, learning to program is difficult, and the act of programming is often a skill that is quite distant from what they are used to learn. It thus quickly leads to a high cognitive load. I here argue that participatory live coding can help reduce this load. Participatory live coding is a form of direct instruction where the teacher guides the students through the material. Rather than tasking the students to solve a program problem from scratch based on slides or a textbook, the students are shown how to work with the programming language and how to build the solutions, all the while doing the actual programming and execution. There is immediate feedback upon program execution in the form of the (lack of) results of running the program, or error messages.
During this form of direct instruction, students are shown not only what to program and how each element works, but also how to program, i.e. how to go from a problem formulation to a working solution (the thinking process). Intrinsic load can be kept to a minimum by teaching small chunks at a time, reinforcing learning by adding frequent exercises. Germane load is high as the activity exactly matches what is required for demonstrating learning, i.e. running

a syntactically correct program that correctly solves the problem at hand. Extraneous load can be minimized in several ways, some of which I'll discuss below.

Below I will discuss several aspects that I and others[3] have found need to be considered when operationalizing participatory live coding as a teaching technique to reduce cognitive load and enhance learning[4].

Align the learning environment

The learning environment of teacher and students should be aligned. With this, I mean that the programming environment is identical for both, the layout of the screen and programs used is mirrored. An example from BIOS1100 is that students as well as teachers use a cloud-based server, called JupyterHub, for programming in so-called Jupyter Notebooks. The environment the students see the teacher code in is identical to what they themselve use. When students, assistants and teacher all are seeing and commenting code in the same environment, discussing problems and helping each other does not the extraneous load of switching environments. An additional benefit of this programming environment is that is saves the students from installing software on their own laptop: as long as they have a working internet connection they can log in (using university credentials) to the server and start working.
There is of course still a component of extraneous load that plays into this: while there is some 'start-up' cost to getting familiar with the Jupyter Notebook environment (upfront load that could be considered extraneous), once this is overcome, it is identical for teachers and students, aiding in reducing extraneous load also when teachers, teaching assistants or students switch between computers to discuss code amongst each other. The use of the notebook thus becomes part of the germane load. In my experience, after a short while students don't even think anymore about how to deal with the notebook when participating in a participatory live coding session.

Require active participation

An important element of participatory live coding is the *active participation* of the student. Rather than passively observing the presentation of a set of slides and an oral explanation of what is on them, or even passively observing the teacher doing the live coding, students need to repeat for themselves the code concurrently with the teacher writing and executing it (they need to 'code along'). This increases germane load and makes the learning experience authentic. Aligning the learning environment as described above reduces the load associated with mirroring the teacher.

Go slow

Live coding the material is necessarily going to be slower than presenting a deck of slides. It takes time to type the code, explain the elements, make a small change and run it again, etc. This in itself is an advantage, but it may be worth adjusting the pace even more to ensure

[3] For example, in The Carpentries community
[4] Some of these points I have previously made in a blog post on the Software Carpentry blog, and are now part of The Carpentries Instructor Training material.

your students are able to follow along. Participatory live coding also lends itself very well to teaching small bits at a time. Isolated programming concept and constructs allows for thoroughly understanding those first, followed by integrating these with previously taught concepts to show these can be used to solve more complicated problems. This strategy contributes to managing intrinsic load. Regularly checking understanding with the students helps maximising learning.

Check that learning is happening

Learning the next step in programming often requires enough understanding of the previous steps, which otherwise may increase cognitive load. The regular use of formative assessment enables the teacher to check in with their students and diagnose lack of knowledge and misconceptions that then immediately can be addressed by further demonstration and explanation using live coding. I often use multiple choice questions for this: I pose a question checking understanding of the material that just has been taught, which has four plausible answers, one wrong one, and three, the distractors, that are not too obviously wrong. Well-written distractors expose misunderstandings that then can be addressed. Using an online tool such as Mentimeter, students can vote for what they think is the correct answer, and the tally of votes can be shown to the students, without revealing the correct answer. If there is disagreement amongst the students, a good approach is to have them discuss the problem in pairs and vote again. Often, one will see the voting converging on the right answer. This technique is a relatively straightforward implementation of Peer Instruction (Crouch and Mazur, 2001).
A very practical technique I learned from The Carpentries is the use of two-colored sticky notes (Wilson, 2019). One color, for example red, is used by the student to indicate a problem, a request for help. Instead of having to raise their hand waiting for attention, the student can continue troubleshooting. These sticky notes also signals to the teacher to wait with further teaching until all of student's problems have been resolved. The other color sticky note, for example blue, can be used by students to indicate that they have finished an exercise. The teacher will quickly see where all students are: done (blue), needing help (red) or still working (no sticky note), before continuing the teaching.

Use exercise types known to reduce cognitive load

Participatory live coding lends itself excellently to the techniques of worked examples and problem completion. Demonstrating the solution to a problem and then asking students to solve a related problem directly afterwards is easily achieved in a participatory live coding session. Also, providing students with Parson's problems, scrambled code that they need to rearrange to obtain a workable program (Parsons and Haden, 2006), is a suitable exercise where both the correct and incorrect solutions can shown using live coding. Faded examples, or 'fill-in-the-blank(s)' exercises are other examples. These types of exercises reduce the load of having to type a lot of code as most or all of the code is given, avoiding unnecessary typos leading to (syntax) errors. Thus, they reduce extraneous load and contribute to maximise germane load, as well as saving time. When using Jupyter

notebooks, I use a technique for providing bits of code to each student without having them to type it in[5], which saves even more time.

Embrace mistakes

An important challenge when learning to program is learning how to deal with errors and error messages. I always explain to my students that error messages (syntax errors or run-time errors) are not formulated with novices in mind, but for people with a decent knowledge of the programming language and its mental model. Error message can thus lead to extraneous load. When live coding, the teacher will undoubtedly make an occasional mistake, leading to such an error. Seeing the teacher diagnose and correct the mistake is an important learning opportunity. It also normalises the introduction of such mistakes, hopefully making students less afraid of making them themselves ('if this even happens to the teacher, maybe it is not such a big deal if it happens to me'). Mistakes can also deliberately be added as part of the instructional material to focus specifically on dealing with them.
A more difficult type of errors are bugs, or logical errors that allow the program to run and finish but give incorrect results (McCauley et al., 2008). Carefully worked out examples can introduce this type of error and teach students strategies to diagnose and correct them.

Counter the split-attention effect

It is advised when doing live coding as a teacher to verbalise what is typed, i.e. to say out loud all programming elements, either simultaneously with, or directly following, the actual typing. This gives students multiple channels of information to work with (auditory, visual), but these are redundant and thus prevent the split-attention effect. For example, these lines of Python code print the first letter for the name of each animal in a list:

```
for animal in animals:
    print(animal[-1])
```

When teaching this, I might say out loud: "for animal in animals colon enter indent print open bracket animal open square bracket -1 close square bracket close bracket enter."

Improvise sparsely

It is advised to follow a script, i.e. lesson material carefully prepared in advance, when live coding. Improvisation is of course a great way of conferring knowledge to the students, but there is a risk of something unexpected happening that require troubleshooting. This may take time, and distracts the students from what is actually to be learned (increasing extraneous load).

Make learning authentic

In my course BIOS1100, I use biological examples, problems and datasets as much as I can. This helps frame programming (and computational modelling, an important aspect of

[5] students can import code into the notebook from files they have on disc using the so-calles '%load magic', see https://carpentries.org/blog/2018/09/teaching-tip-exercise-discussion/

the course) as a relevant skill for the students. For example, rather than calculating cumulative interest, we calculate how many bacteria we have at a certain time point given exponential growth. Using domain-relevant material prevents adding cognitive load from having to consider another domain not familiar to, and relevant for the student.

Possible research questions

I am hoping to be able to pursue some research questions around the effectiveness of participatory live coding in teaching novices programming. Some example questions I would like to address are:

- is the participatory (code-along) part beneficial for learning or is observing a teacher live coding sufficient in itself?
- does seeing the teacher make and solve mistakes reduce the fear novice students may have of making mistakes themselves?
- how long after teaching can the effect of this possible benefit be measured using standardised tests?
- when students become more proficient in programming, at what point, if at all, does the benefit reduce or disappear?
- do students with low confidence in their ability to learn programming at the start of the course benefit from the participatory live coding approach in that it helps them obtain a growth mindset towards learning programming?
- do students from computer science benefit to a lesser degree from participatory live coding (i.e., they may not really ‘need’ the participatory approach to get started learning programming) relative to students in subjects, such as biology, that traditionally do not teach programming?

Summary

Learning to program is difficult and is prone to lead to high cognitive load. In this essay, I have argued that the technique of participatory live coding can be useful to reduce cognitive load while learning programming. Participatory live coding is a form of direct instruction where the teacher guides the students through the material. Using my own experience in implementing the technique in when teaching programming to beginners, I have shown several practical aspects that need to be considered when operationalizing participatory live coding as a teaching technique to reduce cognitive load. Finally, I posed some possible research questions that would establish to what degree the observed benefits of participatory live coding can be quantified.

Bibliography

Anthony R., Jr., A. (2008). Cognitive Load Theory and the Role of Learner Experience: An Abbreviated Review for Educational Practitioners. AACE J. *16*, 425–439.

Caspersen, M.E., and Bennedsen, J. (2007). Instructional Design of a Programming Course: A Learning Theoretic Approach. In Proceedings of the Third International Workshop on Computing Education Research, (New York, NY, USA: ACM), pp. 111–122.

Crouch, C.H., and Mazur, E. (2001). Peer Instruction: Ten years of experience and results. Am. J. Phys. *69*, 970–977.

Denny, P., Luxton-Reilly, A., and Carpenter, D. (2014). Enhancing Syntax Error Messages Appears Ineffectual. In Proceedings of the 2014 Conference on Innovation & Technology in Computer Science Education, (New York, NY, USA: ACM), pp. 273–278.

Dweck, C.S. (2006). Mindset: the new psychology of success (New York: Random House).

Guzdial, M. (2015). Learner-Centered Design of Computing Education: Research on Computing for Everyone (Morgan & Claypool).

Hermans, F., and Aldewereld, M. (2017). Programming is Writing is Programming. In Proceedings of the International Conference on the Art, Science, and Engineering of Programming - Programming '17, (Brussels, Belgium: ACM Press), pp. 1–8.

Hermans, F., Swidan, A., and Aivaloglou, E. (2018). Code Phonology: An Exploration into the Vocalization of Code. In Proceedings of the 26th Conference on Program Comprehension, (New York, NY, USA: ACM), pp. 308–311.

Jenkins, T. (2002). On the difficulty of learning to program. In Proceedings for the 3rd Annual Conference of the LTSN Centre for Information and Computer Sciences, (Loughborough University), pp. 53–58.

Kirschner, P.A., Sweller, J., and Clark, R.E. (2006). Why Minimal Guidance During Instruction Does Not Work: An Analysis of the Failure of Constructivist, Discovery, Problem-Based, Experiential, and Inquiry-Based Teaching. Educ. Psychol. *41*, 75–86.

Luxton-Reilly, A. (2016). Learning to Program is Easy. In Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education, (New York, NY, USA: ACM), pp. 284–289.

McCauley, R., Fitzgerald, S., Lewandowski, G., Murphy, L., Simon, B., Thomas, L., and Zander, C. (2008). Debugging: a review of the literature from an educational perspective. Comput. Sci. Educ. *18*, 67–92.

Morrison, B.B., Dorn, B., and Guzdial, M. (2014). Measuring Cognitive Load in Introductory CS: Adaptation of an Instrument. In Proceedings of the Tenth Annual Conference on International Computing Education Research, (New York, NY, USA: ACM), pp. 131–138.

Parsons, D., and Haden, P. (2006). Parson's Programming Puzzles: A Fun and Effective Learning Tool for First Programming Courses. In Proceedings of the 8th Australasian Conference on Computing Education - Volume 52, (Darlinghurst, Australia, Australia: Australian Computer Society, Inc.), pp. 157–163.

Raj, A.G.S., Patel, J.M., Halverson, R., and Halverson, E.R. (2018). Role of Live-coding in Learning Introductory Programming. In Proceedings of the 18th Koli Calling International Conference on Computing Education Research, (Koli, Finland: Association for Computing Machinery), pp. 1–8.

Robins, A., Rountree, J., and Rountree, N. (2003). Learning and Teaching Programming: A Review and Discussion. Comput. Sci. Educ. *13*, 137–172.

Rubin, M.J. (2013). The Effectiveness of Live-coding to Teach Introductory Programming. In Proceeding of the 44th ACM Technical Symposium on Computer Science Education, (New York, NY, USA: ACM), pp. 651–656.

Skudder, B., and Luxton-Reilly, A. (2014). Worked Examples in Computer Science. In Proceedings of the Sixteenth Australasian Computing Education Conference - Volume 148, (Darlinghurst, Australia, Australia: Australian Computer Society, Inc.), pp. 59–64.

Sweller, J., van Merriënboer, J.J.G., and Paas, F. (2019). Cognitive Architecture and Instructional Design: 20 Years Later. Educ. Psychol. Rev.

Wilson, G. (2019). Teaching tech together: how to make lessons that work and build teaching community around them (Chapman and Hall).