

# Manim-DFA: Visualising Data Flow Analysis and Abstract Interpretation Algorithms with Automated Video Generation

Lucas Berg<sup>a</sup>, Gonzague Yernaux<sup>b</sup>, Mikel Vandeloise<sup>c</sup> and Wim Vanhoof<sup>d</sup>

Faculty of Computer Science, University of Namur, Belgium

{lucas.berg, gonzague.yernaux, mikel.vandeloise, wim.vanhoof}@unamur.be

**Keywords:** Visual Learning, Video Generation, Abstract Interpretation, Data Flow Analysis, Static Analysis.

**Abstract:** In this paper, we introduce *Manim-DFA*, an extension of the *Manim* library for generating video visualisations to teach data flow analysis and abstract interpretation. Despite the importance of data flow analysis in static program analysis, educational visualisation tools remain scarce. *Manim-DFA* addresses this gap by enabling educators to animate control flow graphs and lattice structures, illustrating their transformation during program analysis. Currently, the tool supports automated animation of the worklist algorithm, as well as lattice visualisation. Designed with established pedagogical principles, *Manim-DFA* promotes active learning, reduces cognitive load, and enhances conceptual understanding. Preliminary evaluations suggest that it effectively complements traditional resources and supports autonomous learning.

## 1 INTRODUCTION


Data flow analysis examines how *data* enters, evolves, and exits a program (Khedker et al., 2017). This can be achieved with *dynamic analysis* techniques, which observe program behaviour through concrete executions. However, dynamic analysis sometimes becomes impractical due to the vast number of possible inputs. Instead, with *static analysis*, it is possible to infer program properties without having to execute the programs, providing a more scalable solution.


A foundational milestone in static analysis is incarnated by Kildall's algorithm, which was introduced in the 1970s for compiler optimisation (Kildall, 1973). It was followed by the definition and formalisation of *abstract interpretation* by the Cousot couple (Cousot and Cousot, 1977). Abstract interpretation underpins the design of most modern data flow analyses, and its results are notably used to drive optimisation, bug detection, and program comprehension (Khedker et al., 2017). As such, they play a crucial role in computer science, particularly in compiler design (Lattner et al., 2021; Marat Akhin, 2011).


Despite their importance, data flow analysis algorithms tend to remain difficult to grasp by computer science students. This is mainly due to their reliance


on abstract interpretation notions, which require a different mental model than traditional programming (Norman, 2014). The current advent of generative AI may further accentuate these mental model gaps, since the developers tend to write less code lines themselves (Prasad and Sane, 2024). Meanwhile, computer science education research tends to highlight the role of program and algorithm visualisation in addressing programming difficulties, due to their effectiveness in fostering accurate mental models and improving learning outcomes (Sorva et al., 2013; Hundhausen et al., 2002).

To the best of our knowledge, no tools currently exist that provide visualisations for data flow analysis algorithms. To address this gap, we introduce *Manim-DFA*, an extension of the *Manim* library that generates illustrative videos of key data flow analysis concepts. The videos use tabular, textual, as well as graphical representations to depict successive iterations of the worklist algorithm, a procedure derived from Kildall's algorithm that loops through a program's control flow graph in order to approximate some properties of interest. The application is designed so that its inputs can be easily parametrised by pedagogical teams, while its outputs serve to guide students in step-by-step analysis executions, all in compliance with core pedagogical principles.

<sup>a</sup>  <https://orcid.org/0009-0008-8595-1986>

<sup>b</sup>  <https://orcid.org/0000-0001-6430-8168>

<sup>c</sup>  <https://orcid.org/0009-0002-0858-471X>

<sup>d</sup>  <https://orcid.org/0000-0003-3769-6294>

## 2 TECHNICAL BACKGROUND

Algorithm and program visualisation are often distinguished as follows (Price et al., 1993):

- **Algorithm Visualisation (AV)** represents high-level visualisations illustrating algorithmic behaviour without displaying underlying code.
- **Program Visualisation (PV)** refers to low-level visualisations that explicitly depict code execution and runtime data structure changes.

Interestingly, the visualisation of data flow analysis presents a conceptual challenge for this taxonomy. By incarnating both an algorithm and an abstract execution method, it tends to exhibit a hybrid nature that incorporates aspects of both categories. We now define the notions that will be used throughout the paper.

*Mental models* of program execution are unconscious and subjective representations of program flow. Students should develop accurate models in order to reason effectively about code behaviour (Sorva et al., 2013). One way to foster mental models is through the concept of *notional machines* (Du Boulay, 2013) which provide simplified yet sufficiently detailed representations of program execution.

*Control-flow graphs* (CFGs) are widely used program representations in data flow analysis. A CFG visually maps all possible execution paths within a program. Nodes correspond to instructions or code blocks, while edges denote control flow transitions (Allen, 1970).

As for *abstract interpretation*, it is a field essentially concerned with statically approximating program behaviour by tracking concrete program values (such as the value held in a variable) by using so-called *abstract values*, the latter being generalised or over-approximative representations of the concrete values, that still preserve essential properties of interest. Abstract values are typically organised within a partial order that defines their relative *specificity*. This structure is formalised using the mathematical notion of a *lattice*, which amounts to a partially ordered set in which any pair of elements has both a common upper bound and a lower bound. Depending on the underlying domain, lattices can be finite or infinite.

## 3 RELATED WORK

At the moment of writing, the development of tools specifically tailored for the visualisation of data flow analysis algorithms (independent of a given programming language) is, to the best of our knowledge, still at point zero. However, various education-driven PV and AV tools have been developed over the years. In

this section, we give a (non exhaustive) overview of some major approaches in the field.

**Jeliot** (Moreno et al., 2004) aids novice Java programmers to visualise object-oriented concepts, while **VILLE** (Rajala et al., 2007), supporting both Java and C++, has been shown to improve student performance (Kaila et al., 2010). Meanwhile, **Python Tutor** (Guo, 2013), a program widely used online, enhances comprehension in Python, C/C++, and Java (Karnalim and Ayub, 2017; Karnalim and Ayub, 2018). **PITON** (Elvina et al., 2018) and its extension **PITON-DS** (Nathasya et al., 2019) integrate both PV and AV features and have demonstrated benefits in the support of programming tasks. Some other tools are tailored to visualise control flow graphs (CFGs), typically using Sugiyama-style layouts (Sugiyama et al., 1981), which is often achieved via Graphviz’s dot algorithm (Gansner et al., 1993). Example of such implementations include **CFGExplorer** (Devkota and Isaacs, 2018) and **CCNav** (Devkota et al., 2020).

The various programs (and papers) mentioned above primarily target compiler experts and researchers, as their focus is mostly set on debugging and optimisation tasks. Their output visualisations, which are typically optimised for reducing edge intersections, often result in complex and dense layouts, with similar structures being potentially represented differently across a given graph. Consequently, these approaches are challenging for students to interpret and, hence, can be considered unsuitable for educational purposes, where clarity and accessibility are paramount (Mayer, 2002).

## 4 RESEARCH QUESTIONS

Visualising data flow analysis algorithms, along with their corresponding CFGs, thus still represents a significant challenge, especially in a context where understanding programs tends to become as crucial as writing code oneself (Liu et al., 2024).

**Research Question 1 (RQ1): How Can One Create a Tool that Enables Teachers and Students to Visualise Data Flow Analysis with Minimal Effort and Maximum Pedagogical Value?** Since it will be used in an educational context, such a tool should be particularly helpful to visualise small-scale programs, prioritising clarity for students and ease of use for educators. More specifically, RQ1 entails the search for an adequate format for schematising data flow analysis algorithms. It also entails to decide which program, framework, or library should be used in the development of the envisioned tool. Moreover,

outputs should be easily shared, while their pedagogical effectiveness should be maximised. As for the inputs, these should preferably be resilient to the use of different programming languages or styles.

Additionally, since data flow analysis operates directly on CFGs rather than on the code itself or on its executions, its visualisation requires that of clear-to-read CFGs. The evolution of abstract values computed throughout an analysis should be clearly depicted as well, and the tool should provide a clear and structured representation of the underlying lattice(s), while addressing challenges such as readability and mixed (or complex) value types. Then, beyond CFGs and lattices, elements like text and table displays are required for visualising a program analysis. The critical challenge is to design an intuitive interface that allows educators to create animations without requiring extensive technical expertise.

**RQ2: Can Such Visualisations Help Students Understand Data Flow Analysis Algorithms and Perceive Their Educational Value?** This question focuses on assessing whether the tool enhances comprehension and aligns with the benefits of existing visualisation tools for programming education.

## 5 METHODOLOGY

Selecting an appropriate format for visualisations is crucial. Unlike conventional program visualisation, which typically targets a specific programming language and execution model, data flow analysis visualisation involves multiple interpretations of the same programming language, depending on the type of analysis being conducted. This requires a flexible tool that educators can easily customize and distribute. Additionally, deploying dedicated applications for each analysis tends to be impractical. Non-interactive formats like videos, GIFs, or slides offer a widely supported, cross-platform alternative that integrates seamlessly with educational platforms and requires no additional installation. Among these, videos were chosen for their ability to show the dynamic evolution of data flow analysis processes in a step-by-step manner, thereby enhancing engagement and comprehension (Seo et al., 2021; Zhang et al., 2006).

To minimise the effort required to create video visualizations while ensuring maintainability, flexibility, resource availability and mathematical support, we evaluated candidate tools. **Manim** (The Manim Community, 2024) emerged as a practical choice due to its strong support for  $\text{\LaTeX}$ , graph manipulation, and educational features, that have contributed to its

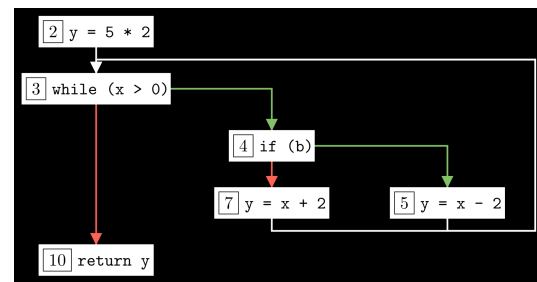


Figure 1: A control flow graph visualisation.

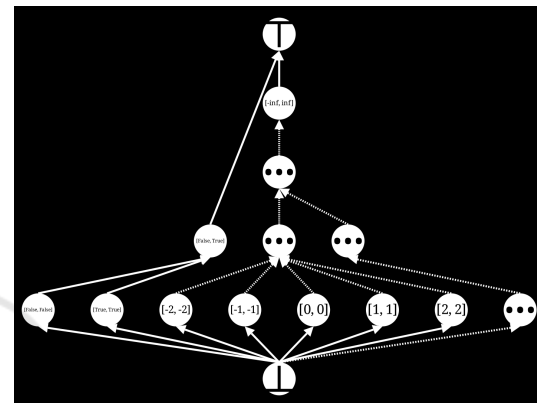


Figure 2: Lattice visualisation, height limited to 6.

widespread adoption. Alternatives such as **Javis.jl** (Humans of Julia Discord Community, 2025), **Reanimate** (Himmelstrup, 2025) and **Makie.jl** (Danisch and Krumbiegel, 2021) were considered but deemed less suitable due to limited mathematical capabilities and documentation or less active communities.

We first extended Manim with a new graph layout adapted to depict CFGs, excluding flow-breaking constructs (e.g. `break`, `continue`, `goto`). Our plugin greedily optimises edge placement, which reduces intersections and improves branch visibility when compared to Manim’s native graph visualisation algorithms. An example CFG is given in Figure 1.

The same holds for lattices. Our custom algorithm builds upon Sugiyama layouts, the main addition being the support for infinite lattices, which are depicted by incrementally linking nodes from both ends to the middle. For this to be possible, the lattices must be bounded with well-defined top and bottom elements. Figure 2 shows an example lattice representation.

For now, Manim-DFA is limited to visualising data flow analysis algorithms that use the so-called *worklist* approach. The worklist algorithm is one of the most widespread implementations of the Kildall algorithm (Kildall, 1973) mentioned above. The algorithm indeed boils down to calculating abstract values by looping through the program instructions (following the edges of its CFG) until a fixed point is reached

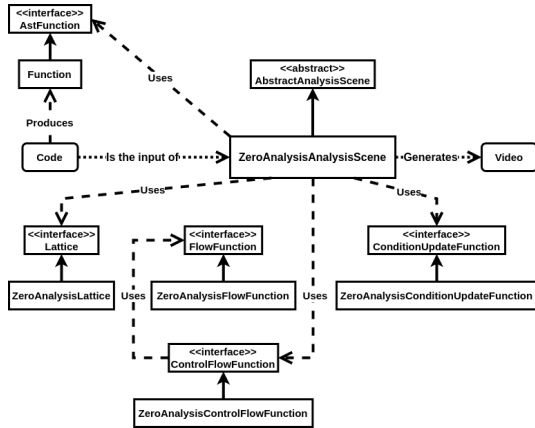


Figure 3: Overview of Manim-DFA's workflow.

for each computed abstract value.

While being widely used in program analysis, and in particular in introductory courses, the worklist algorithm is not the only technique for computing abstract values in an input program. Manim-DFA is designed to be extensible on that regard; animating other algorithms does, however, require some more manual operations. For example, when working with the well-known GEN/KILL algorithm, which is concerned with computing *program properties* enclosed in sets (rather than classical abstract values) (Fernandes and Desharnais, 2004), one can easily leverage the previously developed CFG and lattice visualisation methods, along with the features provided by Manim, to animate the algorithm.

The mandatory inputs to Manim-DFA are:

- an imperative program, represented as an abstract syntax tree (AST);
- a lattice structure defining the abstract domain;
- flow functions and condition update functions specifying how abstract value are updated during the analysis.

As shown in Figure 3, these inputs are required to comply to specific classes and interfaces. Then, Manim-DFA generates an animation that demonstrates the execution of the analysis on the input program(s). Educators can optionally integrate a text parser; this allows to convert raw source code from any language into an abstract syntax tree, to alleviate (and generalise) the input requirements.

Manim-DFA is written in object-oriented Python, using the core Manim library as well as our lattice and CFG visualisation helpers. The code is available as an online repository<sup>1</sup>. The different classes of Manim-DFA implement general interfaces in such a way that one can generate videos without the need to place the elements directly on the screen. A sep-

<sup>1</sup>See <https://tinu.be/ManimDFA>.

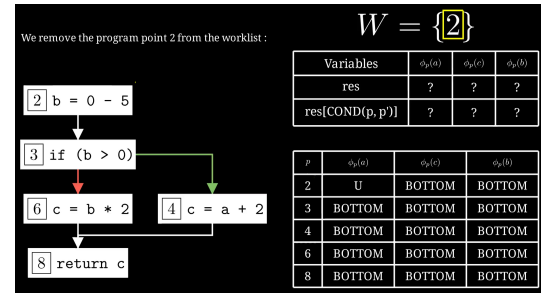


Figure 4: Removal of the current program point from  $W$ .

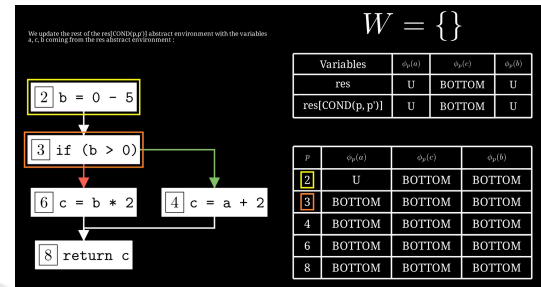


Figure 5: Update of abstract values based on flow and condition update functions.

arate repository is used to store the source code and documentation of a lightweight compiler for the so-called *Small* toy language, being a simplistic imperative language that can be used to illustrate program analysis concepts in classrooms. The documentation of this example source language can be found along with its compiler's source code in a separate repository<sup>2</sup>. The *Small* compiler has been implemented using ANTLR 4 (Parr, 2013); however, as detailed above, the user is free to use any language as input to the analysis process included in the first repository, for as long as its programs can be translated into compatible abstract syntax trees— which is the reason why the *Small* compiler is stored in a separate repository.

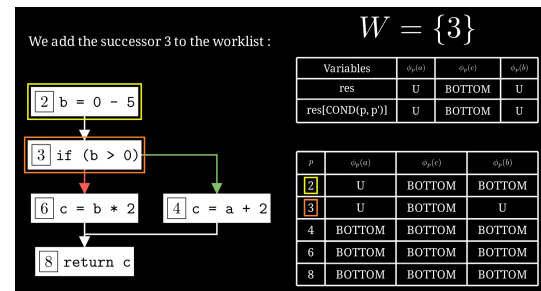


Figure 6: Storage of updated abstract values and selection of the next program point.

As an illustration of Manim-DFA's typical output, Figure 4 displays the removal of an item from the

<sup>2</sup>See <https://tinu.be/SmallCompiler>.



worklist  $W$  after an iteration of the eponym algorithm on an example `Small` program given as input, while Figure 5 shows how the abstract interpretation functions are then used to update the abstract values associated with program variables. Lastly, Figure 6 illustrates how the worklist algorithm proceeds to store the new abstract values and mark the following `Small` program instruction to be handled, resulting in the new worklist value  $W = \{3\}$ . These three figures are screenshots of video frames created by `Manim-DFA`.

## 6 PEDAGOGICAL ASPECTS

To ensure the effectiveness of our visualisation tool, we anchored its design in established educational theories and frameworks. Below is an overview of largely adopted cognitive pedagogy principles, which steered the design and development of `Manim-DFA` at every stage of its conception.

First, Bruner’s **discovery learning** theory highlights the importance of active exploration in building meaningful knowledge (Bruner, 2009), while Ausubel states that **meaningful learning** is more effectively achieved when new information is explicitly linked to prior knowledge (Ausubel, 1963). `Manim-DFA` aligns with both these statements by enabling learners to observe program state transformations, which are structured into clear, conceptual stages, and supported by examples that help integrate new knowledge.

Similarly, Flavell’s concept of **metacognition** emphasises the importance of monitoring and adapting learning strategies (Flavell, 1979). The sequential and detailed visualisations provided by `Manim-DFA` encourage learners to identify and address their misconceptions, thereby enabling a certain degree of autonomy in problem-solving.

In fact, by illustrating algorithmic branching decisions, `Manim-DFA` transforms abstract concepts into interactive visual experiences. This is also in line with Kolb’s **experiential learning model**, which stresses the role of concrete experiences in knowledge acquisition (Kolb, 2014).

Yet another framework, Gagné’s **nine learning events model**, identifies a series of steps that optimise learning, ranging from capturing attention to providing feedback (Driscoll, 2005). `Manim-DFA` follows the progression detailed in his work, e.g. by using dynamic animations to maintain engagement, gradually presenting concepts, and delivering immediate visual feedback that reinforces understanding. Related notions such as Mayer’s **segmentation effect** and Sweller’s **cognitive load** theories are also

covered by this. Simply put, these two intuitive frameworks respectively suggest that breaking down complex tasks into smaller segments enhances understanding (Mayer, 2009) and that learning methods should minimise the number of unnecessary cognitive demands (Paas et al., 2004).

`Manim-DFA` combines animations with explanatory annotations. In that regard, it also complies to Paivio’s **dual coding** framework, which states that learning improves when information is processed both visually and verbally (Paivio and Clark, 2006).

By integrating these eight core principles, `Manim-DFA` aims to enable a comprehensive and lasting understanding of complex concepts, such as the rather intricate worklist algorithm. To do this, the tool reduces cognitive barriers and promotes active engagement, encourages reflective thinking, and allows an extensive transfer of knowledge to new contexts. This is, at least, our conjecture; the next section describes an evaluation carried out based on a concrete use of our tool.

## 7 PRELIMINARY EVALUATION

We conducted a questionnaire following empirical standards in software engineering (Ralph and al., 2021). The survey targeted Master’s students in computer science at the University of *Namur*, all of whom had already completed a course on programming language semantics. They were provided with `Manim-DFA`-generated videos as supplementary resources in their program analysis course, which was graded based on an individual project.

Participation to the survey was voluntary and responses were anonymised. The questionnaire combined Likert-scale questions for quantitative analysis with an optional open-ended section. Out of seventeen students, six completed the questionnaire ( $n = 6$ ), four of which had watched the videos.

The students ranked various activities based on their usefulness in understanding data flow analysis (Table 1). Videos and slides were rated the least useful, whereas practical work and theoretical courses were ranked highest.

Next, the Likert responses denoted in Table 2 suggest that, while students generally found available resources sufficient and clear, opinions varied on the clarity in the course’s project statement. The project was also perceived as challenging.

Students who watched the videos rated their effectiveness, as shown in Table 3. The videos were considered useful complements to other resources, with clear animations aiding comprehension. However, the

Table 1: Usefulness of resources (1=lowest, 7=highest).

Activity/Resource	Mean	Std	Min	Max
Videos	2.67	1.37	1	4
Textbook	4.67	1.75	2	7
Slides	2.00	1.55	1	5
Practical works	5.50	1.05	4	7
Example project	3.67	2.34	1	7
Course	5.00	1.41	3	7
Individual project	4.50	2.43	2	7

Table 2: Statements (1=strongly disagree, 5=fully agree).

Statement	Mean	Std	Min	Max
Lack of resources	3.00	1.41	1	5
Theoretical/practical courses were clear	4.33	0.52	4	5
Addit. resources were clear	4.17	0.41	4	5
Project instructions were clear	3.67	1.21	2	5
Project instructions were too complex	2.50	1.05	1	4
Example project was helpful	4.17	1.17	2	5

students suggested using shorter videos and including verbal explanations.

The students who did not watch the provided videos declared that it was due to a lack of time, or because they did not feel the need for it (Table 4). All students, however, saw value in educational videos, provided they included verbal explanations (Table 5). Then, when asked to rate some statements about resources or activities, the students expressed an overall interest in additional course materials, particularly videos (Table 6).

## 8 DISCUSSION AND CONCLUSIONS

By leveraging the Manim library, Manim-DFA automates the animation of hard-to-grasp data flow analysis techniques into instructional videos, all while

Table 4: Reasons for not watching the videos.

Reason	Value
I did not have time	50%
I did not feel the need	100%
The videos are too long or too slow	0%
I did not find them	0%

Table 5: Potential improvements for videos.

Answer	Value
The videos should be commented	100%
The videos should be more theoretical	0%
The videos are fine as-is	0%
The videos are not useful at all	0%

Table 6: Suggestions (1=strongly disagree, 5=fully agree).

Activity / Resource	Mean	Std	Min	Max
More exercises	3.83	1.17	2	5
More videos	4.17	0.75	3	5
More example projects	3.83	1.17	2	5
Improve practical sessions	3.17	1.33	1	5
Improve theoretical course	2.33	1.21	1	4
Improve slides	3.5	1.38	1	5
Improve textbook	2.17	1.47	1	4
Improve videos	3.0	1.10	1	4

adhering to well-established learning methodologies. Although preliminary feedback suggests it enhances comprehension, limited adoption and mixed reactions to the video format highlight areas for refinement.

For instance, integrating Manim-DFA more seamlessly into formal coursework could enhance its adoption and impact. Future versions of the tool could address student feedback by incorporating shorter, more focused videos and enriching them with verbal explanations so as to accommodate diverse learning preferences.

Future work will also focus on expanding the scope of supported algorithms, such as incorporating the GEN/KILL framework and other common data flow analysis techniques.

Table 3: Likert answers regarding the videos (1=strongly disagree, 5=fully agree).

Statement	Mean	Std	Min	Max
The videos effectively complement other educational resources	4.25	0.5	4	5
The videos allowed me to understand something new	4.0	0.0	4	4
The videos are too long	3.25	0.957	2	4
The videos would be better if they were commented	4.25	0.5	4	5
The animations used in the videos are clear	4.5	0.577	4	5
The content of the videos is adapted to my level of understanding	4.0	1.414	2	5
I interacted with the videos (e.g. pause, rewind)	4.25	1.5	2	5
The videos helped me memorize concepts better than other media	3.0	0.0	3	3
The videos helped me progress autonomously in understanding concepts	3.5	0.577	3	4
The videos encouraged me to test or reproduce the concepts myself	2.5	1.732	1	5

Moreover, improving user accessibility, e.g. by simplifying the interface and streamlining the process of defining analyses, should enable educators with limited technical expertise to benefit more freely from the tool. We also plan on investigating the potential for interactivity, such as allowing students to directly manipulate visualisations or test hypothetical scenarios.

Collaborations with educators from other institutions and disciplines may also uncover new use cases and drive broader adoption.

Additionally, refining the visual and structural design of animations, particularly for complex lattice structures and large control flow graphs, is part of the planned ongoing work as the tool evolves.

As specified in the empirical standards for surveys and questionnaires (Ralph and al., 2021), so-called "threats to validity" may affect the precision and credibility of our findings in Section 7. In particular, the small sample size, due to course enrolment and response rates, obviously limits broader applicability or generalisation of the results. Future research should extend the study to larger and more diverse cohorts. Secondly, biases may persist in the questionnaire, as respondents were students actively engaged in the course. While efforts were made to ensure representativeness through anonymity and brevity, the order of the questions, for example, remains potentially confusing. Thirdly, despite our best efforts, evaluating comprehension is inherently complex. Future studies should incorporate objective measures, such as project grades, to complement self-reported learning outcomes. Lastly, future iterations should assess the questionnaire's consistency across different student groups.

## REFERENCES

- Allen, F. E. (1970). Control flow analysis. *ACM SIGPLAN Notices*, 5(7):1–19.
- Ausubel, D. P. (1963). *The Psychology of Meaningful Verbal Learning*. Grune & Stratton.
- Bruner, J. S. (2009). *The Process of Education*. Harvard university press.
- Cousot, P. and Cousot, R. (1977). Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages - POPL '77*, pages 238–252, Los Angeles, California. ACM Press.
- Danisch, S. and Krumbiegel, J. (2021). Makie.jl: Flexible high-performance data visualization for Julia. *Journal of Open Source Software*, 6(65):3349.
- Devkota, S., Aschwanden, P., Kunen, A., Legendre, M., and Isaacs, K. E. (2020). CcNav: Understanding compiler optimizations in binary code. *IEEE transactions on visualization and computer graphics*, 27(2):667–677.
- Devkota, S. and Isaacs, K. E. (2018). CFGExplorer: Designing a Visual Control Flow Analytics System around Basic Program Analysis Operations. *Computer Graphics Forum*, 37(3):453–464.
- Driscoll, M. P. (2005). Psychology of learning for instruction. *Person Education*.
- Du Boulay, B. (2013). Some difficulties of learning to program. In *Studying the Novice Programmer*, pages 283–299. Psychology Press.
- Elvina, E., Karnalim, O., Ayub, M., and Wijanto, M. C. (2018). Combining program visualization with programming workspace to assist students for completing programming laboratory task. *JOTSE: Journal of Technology and Science Education*, 8(4):268–280.
- Fernandes, T. and Desharnais, J. (2004). Describing gen/kill static analysis techniques with kleene algebra. In Kozen, D., editor, *Mathematics of Program Construction*, pages 110–128, Berlin, Heidelberg. Springer Berlin Heidelberg.
- Flavell, J. H. (1979). Metacognition and cognitive monitoring: A new area of cognitive–developmental inquiry. *American psychologist*, 34(10):906.
- Gansner, E. R., Koutsofios, E., North, S. C., and Vo, K.-P. (1993). A technique for drawing directed graphs. *IEEE Transactions on Software Engineering*, 19(3):214–230.
- Guo, P. J. (2013). Online python tutor: Embeddable web-based program visualization for cs education. In *Proceeding of the 44th ACM Technical Symposium on Computer Science Education*, pages 579–584, Denver Colorado USA. ACM.
- Himmelstrup, D. (2025). Reanimate: Build declarative animations with svg and haskell. <https://reanimate.github.io/>.
- Humans of Julia Discord Community (2025). Javis.Jl. <https://juliaanimators.github.io/>.
- Hundhausen, C. D., Douglas, S. A., and Stasko, J. T. (2002). A meta-study of algorithm visualization effectiveness. *Journal of Visual Languages & Computing*, 13(3):259–290.
- Kaila, E., Rajala, T., Laakso, M.-J., and Salakoski, T. (2010). Effects of course-long use of a program visualization tool. In *Proceedings of the Twelfth Australasian Conference on Computing Education-Volume 103*, pages 97–106.
- Karnalim, O. and Ayub, M. (2017). The use of python tutor on programming laboratory session: Student perspectives. *Kinetik: Game Technology, Information System, Computer Network, Computing, Electronics, and Control*, pages 327–336.
- Karnalim, O. and Ayub, M. (2018). A Quasi-Experimental Design to Evaluate the Use of PythonTutor on Programming Laboratory Session. *International Journal of Online Engineering*, 14(2).
- Khedker, U., Sanyal, A., and Sathe, B. (2017). *Data Flow Analysis: Theory and Practice*. CRC Press.

- Kildall, G. A. (1973). A unified approach to global program optimization. In *Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages - POPL '73*, pages 194–206, Boston, Massachusetts. ACM Press.
- Kolb, D. A. (2014). *Experiential Learning: Experience as the Source of Learning and Development*. FT press.
- Lattner, C., Amini, M., Bondhugula, U., Cohen, A., Davis, A., Pienaar, J., Riddle, R., Shpeisman, T., Vasilache, N., and Zinenko, O. (2021). MLIR: Scaling compiler infrastructure for domain specific computation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 2–14.
- Liu, J., Xia, C. S., Wang, Y., and Zhang, L. (2024). Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. *Advances in Neural Information Processing Systems*, 36.
- Marat Akhin, M. B. (2011). Control- and data-flow analysis - Kotlin language specification. <https://kotlinlang.org/spec/control-and-data-flow-analysis.html>.
- Mayer, R. E. (2002). Cognitive theory and the design of multimedia instruction: An example of the two-way street between cognition and instruction. *New Directions for Teaching and Learning*, 2002(89):55–71.
- Mayer, R. E. (2009). Segmenting principle. *Multimedia learning*, 2:175–188.
- Moreno, A., Myller, N., Sutinen, E., and Ben-Ari, M. (2004). Visualizing programs with Jeliot 3. In *Proceedings of the Working Conference on Advanced Visual Interfaces*, pages 373–376, Gallipoli Italy. ACM.
- Nathasya, R. A., Karnalim, O., and Ayub, M. (2019). Integrating program and algorithm visualisation for learning data structure implementation. *Egyptian Informatics Journal*, 20(3):193–204.
- Norman, D. A. (2014). Some observations on mental models. In *Mental Models*, pages 15–22. Psychology Press.
- Paas, F., Renkl, A., and Sweller, J. (2004). Cognitive load theory: Instructional implications of the interaction between information structures and cognitive architecture. *Instructional science*, 32(1/2):1–8.
- Paivio, A. and Clark, J. M. (2006). Dual coding theory and education. *Pathways to literacy achievement for high poverty children*, 1:149–210.
- Parr, T. (2013). *The Definitive ANTLR 4 Reference*. Pragmatic Bookshelf, 2nd edition.
- Prasad, P. and Sane, A. (2024). A self-regulated learning framework using generative ai and its application in cs educational intervention design. In *Proceedings of the 55th ACM Technical Symposium on Computer Science Education V. 1, SIGCSE 2024*, page 1070–1076, New York, NY, USA. Association for Computing Machinery.
- Price, B. A., Baecker, R. M., and Small, I. S. (1993). A principled taxonomy of software visualization. *Journal of Visual Languages & Computing*, 4(3):211–266.
- Rajala, T., Laakso, M.-J., Kaila, E., and Salakoski, T. (2007). VILLE: A language-independent program visualization tool. In *Proceedings of the Seventh Baltic Sea Conference on Computing Education Research-Volume 88*, pages 151–159. Citeseer.
- Ralph, P. and al. (2021). Empirical Standards for Software Engineering Research.
- Seo, K., Dodson, S., Harandi, N. M., Roberson, N., Fels, S., and Roll, I. (2021). Active learning with online video: The impact of learning context on engagement. *Computers & Education*, 165:104132.
- Sorva, J., Karavirta, V., and Malmi, L. (2013). A Review of Generic Program Visualization Systems for Introductory Programming Education. *ACM Transactions on Computing Education*, 13(4):1–64.
- Sugiyama, K., Tagawa, S., and Toda, M. (1981). Methods for visual understanding of hierarchical system structures. *IEEE Transactions on Systems, Man, and Cybernetics*, 11(2):109–125.
- The Manim Community (2024). Manim – a community maintained python library for creating mathematical animations. <https://www.manim.community/>.
- Zhang, D., Zhou, L., Briggs, R. O., and Nunamaker, J. F. (2006). Instructional video in e-learning: Assessing the impact of interactive video on learning effectiveness. *Information & Management*, 43(1):15–27.