Mapping the Landscape of Literate Computing

Bjarke Vognstrup Fog
Department of Digital Design
and Information Studies
Aarhus University
bfog@cc.au.dk

Clemens Nylandsted Klokmose
Department of Digital Design
and Information Studies
Aarhus University
clemens@cavi.au.dk

Abstract

Literate computing is a computing paradigm that interweaves executable code with more conventional media such as prose, images, and video. It has recently seen uptake particularly in the data science community with tools such as Jupyter Notebook, which is an open source system inspired by Mathematica notebooks. These Mathematica-inspired tools are often referred to as computational notebooks. We, however, argue that computational notebooks are just a special case of literate computing tools and that there is an uncharted design space for computing tools that dissolves the traditional distinction between programming and using computers, but also between using and developing software tools.

We examine various programming environments; some new, some old, some fully developed, some research prototypes. By looking at how we got here—the history—and the challenges identified in the research so far, we analyze the environments through a range of themes, such as purpose, user community, system metaphor, malleability, etc. We conclude with a discussion of design considerations for future literate computing environments.

1. Introduction

Literate computing is human-computer interaction where programming and computation is interwoven with manipulation of text, images, video, and other digital media. The output of literate computing is computational media that in the simplest case allows the reader to rerun computations but in more sophisticated cases provide rich interaction capabilities. The output may be in a narrative form (Pérez & Granger, 2015), take the form of a tool or application, or something in between. A *literate computing environment* is software that is designed to enable literate computing, such as a computational notebook.

The term literate computing is derived from Knuth's *literate programming* (Knuth, 1992) but where Knuth's goal is to make a tool to document code for "for system programmers, not for high school students or for hobbyists", literate computing environments mix editing and execution of code to create interactive media and computational narratives, and is particularly designed for non-system programmers. One salient attribute of a literate computing environment is that code coexists in the same space as its output and the objects and data it operates on. This attribute is represented visually, for instance in computational notebooks when code cells are adjacent to their output cells in the same document, but also technically when code is addressable as data.

This computing paradigm has in recent years gained significant popularity, most visible in the uptake of computational notebooks in data science. In 2017, according to Rule et al. (2018), GitHub hosted around 1.2 million publicly available computational notebooks, while Kery et al. (2018) established that by 2015, there were already more than two million users of the popular computational notebook, Jupyter. One can only assume that these numbers have grown since then. That makes literate computing a phenomenon that requires scholarly scrutiny: Where does it have its origins? What are its different forms? What are its strengths and weaknesses? What is the scope of its design space? What is its scope of its (potential) user base?

Literate computing has its legacy in the work by diSessa (diSessa, 2001; diSessa & Abelson, 1986), Papert (Papert, 1993), and Kay (A. Kay, n.d.; A. Kay & Goldberg, 1977), among others, that see computational media as not just an extension of conventional media, but rather as a different type of epistemologically interesting media: "what we have to become in order to use it fluently" (A. Kay,

2013).

Literate computing was coined to describe the particular kind of human-computer interaction present in computational notebooks (Pérez & Granger, 2015). However, we argue that computational notebooks are just one particular expression of the literate computing genre, and that it is worth exploring the broader potentials of the genre. In this paper we will give an overview of the genealogy of literate computing, present the current state of the art of technologies and research and discuss future directions both in terms of the design and study of literate computing environments.

2. Literate computing environments

In this section we will present a range of software environments that enable literate computing or exhibit characteristics related to literate computing. They are chosen for their popularity, novelty, historical importance or because they incorporate specific design choices. We will divide them into three separate but overlapping groups: *Historical* environments that have exemplary significance but generally no longer see widespread use; *in-use* environments that currently have a broad real-world uptake; experimental *research* prototypes or proof-of-concepts that explore certain aspects of literate computing and push the state-of-the-art forward.

2.1. Historical

2.1.1. Boxer

Boxer, created by Andrea diSessa and Harold Abelson in 1986, was presented as a *reconstructible computational medium*. Starting from the premise that programming could (and even should) be a commonplace skill like reading and writing, Boxer provides an environment for non-professionals to program: "Not only would professionals be able to construct grand images, but others would be able to reconstruct personalized versions of these same images" (diSessa & Abelson, 1986).

Boxer is a literate computing environment, and an example of how literate computing does not only equate computational notebooks. Whereas the typical notebook is structured linearly in one dimension, Boxer allows for creating a two dimensional organization of content, resembling whiteboards more than notebooks. Its user interface is simplistic: Everything is represented as (graphical) boxes that might contain code, prose, data, or other types of media, and the system is based on *naïve realism*—only what you see in the canvas is what is there. Like Seymour Papert with LOGO before him, diSessa has since done significant research on the use of Boxer in educational settings, investigating how schoolchildren might appropriate the system for learning and exploration (diSessa, 2001).

2.1.2. HyperCard

HyperCard is not a literate computing environment *per se*, as code is not a first-class citizen but hidden as scripts behind the scenes. HyperCard is, however, a computational media authoring environment that seeks to close the gap between the developer and the user. In this respect, it is the intention behind HyperCard that is of interest. HyperCard was inspired by hypertext as an organizing principle, and in turn inspired early web browsers.

As a programming environment, HyperCard was particular due to the fact that all data is essentially string-based (*A Eulogy for HyperCard*, 2004). In spite of this, several commercial games, such as *Cosmic Osmo* and *Myst*, were developed through the platform. The most interesting aspect of HyperCard is the ease with which it could be used to construct software, even for non-programmers. By providing an interface builder and a scripting language for creating interactivity, HyperTalk, HyperCard brought programming-like abilities to the average household.

2.1.3. Smalltalk

Smalltalk (A. C. Kay, 1993) deserves to be mentioned here, if only for its complete commitment to the *liveness* of software development. Following the object-oriented programming paradigm to its fullest, everything in Smalltalk is simultaneously represented as a conceptual object and a visible GUI element. Further, Smalltalk embodied a concept of liveness that is perhaps more likened to reactive programming, enabling a user to edit the running system from within said system. No compilations or restarts

required. These twin concepts of liveness and malleability are reappearing in several literate computing environments, as we will see later on.

2.2. In use

2.2.1. Mathematica

Mathematica was released in 1988 and is built around the Wolfram computational language. The creator, Stephen Wolfram, argues that Wolfram is not only a programming language as it is "not just a language for telling computers what to do. It's a language that both computers and humans can use to represent computational ways of thinking about things" (*Stephen Wolfram Blog*, 2019).

Since its inception, Mathematica has relied on a notebook format in the form of *computational essays* containing code and other media (Hayes, 1990), and its cell-based format has set the standard for the structure of other computational notebooks. Notably, Mathematica is the only non-historical platform on this list that is not free to use.

2.2.2. Jupyter Notebook

Jupyter Notebook is the most well-known computational notebook today. Previously called IPython, this environment has been around since 2007 (Pérez & Granger, 2007) and was originally based on the popular, open source language Python. While later made available as a web service through Google Colab[1], the environment is designed to be installed on a local machine and used by a single user through a web browser using an HTML based user interface.

Jupyter Notebook employs a conventional file-based structure, imitating the underlying structure of the operating system. However, the files themselves are akin to Knuth's WEB-files (Knuth, 1992) as they contain prose, code, images, etc. all at once. The content is structured into separate cells that can be run or viewed individually. Importantly, Jupyter is by far the most popular interactive notebook for data scientists (Perkel, 2018). Today, Jupyter supports over 100 languages[2]. Jupyter has an active community, and a myriad of extensions exist to tweak the user interface or to, e.g., create slide presentations from a notebook.

2.2.3. Observable

Observable is presented by its creator as "a better way to code" (Bostock, 2017). Like Jupyter Notebook, Observable is a cell-based interactive notebook. However, a key difference is that Observable is web-based and employs a *reactive* programming paradigm in a JavaScript-like language. This allows for instant feedback and updates in the entire document. A notable feature of Observable is how the platform does not distinguish between editing and use—when viewing a notebook, the user can directly edit said notebook and fork it as their own.

Observable takes its departure in data-driven development through D3.js, a data analysis framework. The major difference from the other environments is Observable's graphical representations and inherent focus on data visualization (Bostock, 2018). Through its reactive nature and the easy of which sophisticated interactive visualizations can be embedded, it enables the creation of narratives resembling Victor's explorable explanations (Victor, 2011).

2.2.4. Notion

Although not a programming environment, Notion brings into play some of the design considerations that characterize literate computing. Sitting somewhere among Trello and Slack, Notion serves as an integrated platform for documents, data and media.

It is web-based, but provides desktop shells for popular operating systems, including mobile, that makes it at least feel like a stand-alone application. Particularly interesting in the current context is its script-like capabilities that are closely aligned with Raskin's notion of *commands* (Raskin, 2000, p. 109). These commands, such as `@Today`, execute automatically upon being entered—in this case, the command inserts today's date as a dynamic element. The next day, the string will instead read `@Yesterday`.

[1] `colab.research.google.com`

[2] `https://github.com/jupyter/jupyter/wiki/Jupyter-kernels`

While not technologically ground-breaking, the example serves to illustrate the potential for Notion to enhance a text-based medium with computational capabilities.

2.3. Experimental or research prototypes

2.3.1. iodide

A web-based data science tool, iodide[3] is closely aligned with Knuth's vision of a literate programming environment. The environment presents three different windows: the editor, a preview, and console/workspace. The editor looks like any other text-based code editor with consecutive lines. However, by writing `%%` followed by a certain command, one can insert a divider to change between a range of modes from that line forward. This is a particularly interesting feature of iodide: By changing modes, the user can write in Python, JavaScript, CSS, Markdown, HTML, or call `fetch` to import external data. The editor supports syntax highlighting for all modes. iodide is fully reactive; any change made in the editor is instantaneously reflected in the preview. Lastly, an iodide document is located at a specific URL that can be shared like any other address.

2.3.2. Live notebook

Live notebook[4] is an evolution of the principles from Jupyter Notebook. Live notebook employs a reactive programming environment—that is, cells that are linked together update automatically across the environment. Live notebook also supports native real-time collaboration. And finally, Live notebook provides a WYSIWYG editor and thus eliminates the need to know Markdown, HTML or a similar markup language.

2.3.3. Codestrates

Codestrates (Rädle, Nouwens, Antonsen, Eagan, & Klokmose, 2017) comes from an academic effort to provide new tools for computational thinking. Based on Webstrates (Klokmose, Eagan, Baader, Mackay, & Beaudouin-Lafon, 2015), a dynamic shareable medium, Codestrates operates as a purely web-based environment. Codestrates adopts a structure similar to word processors. Content (e.g., code, prose, data) is structured in sections that can be collapsed and unfolded. The programming language in Codestrates is JavaScript simply because it is the programming language of the web. One of Codestrates' trademarks is that the environment itself is extensible in the spirit of diSessa and Kay. If a function is missing, you can program that function yourself, and every notebook (called a codestrate) is self-contained similar to a Smalltalk image. Code can be shared between codestrates as packages (Borowski, Rädle, & Klokmose, 2018).

Vistrates (Badam, Mathisen, Radle, Klokmose, & Elmqvist, 2019) is an extension to Codestrates that allows users to create a reactive data processing and visualization pipeline. In Vistrates, a non-programmer might never write code themselves and still be able to construct pipelines of data processing and visualizations and aggregate these on an interactive canvas for presentation. Meanwhile, a programmer can unfold the code, edit and create components. In Vistrates, multiple users can simultaneously interact with and reprogram a shared reactive visualisation pipeline.

2.3.4. Eve

Eve is the product of Chris Granger, former lead developer of Visual Studio, who tried to create a "human-first programming platform"[5]. Speaking of Visual Studio, Granger remarked how "it missed the fundamental problems that people faced. And the biggest one that I took away from it was that basically people are playing computer inside their head." (Somers, 2017).

Eve was an attempt to rectify this, although sadly short-lived. Its files embodied a modular approach through which a document was assembled from smaller blocks—what Granger calls "programming by composition". The language of Eve worked through pattern matching, where each block of code was a named group that was matched in real-time with a database to create relations between blocks. This allowed for a novel and reactive programming environment. Eve programs were structured like essays,

[3] `https://alpha.iodide.io/`
[4] `https://livebook.inkandswitch.com/`
[5] `http://witheve.com/`

mixing prose and code modules and showing the resulting graphical application in an adjacent pane.

2.3.5. Capstone

Capstone[6] provides a 2D environment for creative professionals that can be understood as a whiteboard-like canvas with cards. One interesting feature has brought it here: bots. Through a series of experiments in end-user programming, Capstone was augmented with embodied programs (*End-user programming*, 2019). Embodied in the sense that these programs exist on the screen as visual elements. A bot is, then, a daemon-like service that provides computational capabilities, such as automatically reordering the boards on the canvas. Importantly, bots can be created by users themselves to augment their workflow. Further, users can program not only the bots' functionality, but also their visual representation and interactivity.

2.4. Research on literate computing

The research done on literate computing is surprisingly sparse, considering the popularity of computational notebooks in both industry and academia. Broadly, we might distinguish between two kinds of research while bearing in mind that there is a great deal of overlap between them: 1) the use of literate computing environments, and 2) the design of literate computing environments, respectively.

2.4.1. Use

Much research on literate computing environments tend to investigate the use of existing tools. Kery and Myers (Kery & Myers, 2018) have explored how data scientists keep track of notebook versions, while Kery et al. have looked into how scientists curate notebooks into proper narratives (Kery et al., 2018).

Rule et al. investigated how academic data analysts view their own notebooks as "personal, exploratory, and messy", and how they subsequently share the results of their analyses (Rule, Tabard, & Hollan, 2018). Rule's PhD dissertation (2018) likewise shows how people use computational notebooks and goes on to present several design solutions to the obstacles found. Owing to the massive popularity of Jupyter Notebook, Shen and Perkel sought to present the particularities and possibilities of the environment to a wider audience through articles in Nature (Shen, 2014; Perkel, 2018).

2.4.2. Design

Research efforts in literate computing often culminates in new systems that explore possibilities, limits, and opportunities. One such system was Pérez and Granger's IPython (predecessor to Jupyter), presented to their peers in the natural sciences (Pérez & Granger, 2007). More recently, Rädle et al. have developed Codestrates—a shareable, dynamic literate computing environment as part of a wider research effort on developing new tools for computational thinking (Rädle et al., 2017).

Another group of research projects seek to augment existing tools with new capabilities. Rule et al. have investigated how a computational notebook might be structured and collaboratively re-used through cell-folding (Rule, Drosos, Tabard, & Hollan, 2018). Head et al. recently presented extensions to computational notebook environments for "code gathering", helping data scientists de-clutter and organize their work (Head, Hohman, Barik, Drucker, & DeLine, 2019).

Critizing the linear narrative structure that is often implied in computational notebooks, Mathisen et al. have presented InsideInsights (Mathisen, Horak, Klokmose, Grønbæk, & Elmqvist, 2019) that is structured around multiple non-linear narratives, eschewing the data layer in favor of a narrative layer. Wood et al. likewise favor a more complex storytelling featuring branching narratives and a focus on *literate visualizations* (Wood, Kachkaev, & Dykes, 2019).

3. Themes

It is yet too early to create a comprehensive taxonomy on literate computing environments (comparable to, e.g., Kelleher & Pausch, 2005). However, we now present seven themes that can frame an analysis of these environments and provide directions for future research.

[6] https://www.inkandswitch.com/capstone-manuscript.html

3.1. System metaphor

A system metaphor can be understood as "how do I make sense of this thing" by providing a well-known idea, concept, or artifact through which *this thing* can be made meaningful. This is referred to as, among other names, conceptual blending (Fauconnier & Turner, 2003). A conceptual blend arises from the combination of two conceptual inputs, for example the idea of an ordinary desktop mixed with the file system of a computer.

Boxer and HyperCard, two discontinued platforms, utilize a canvas-like metaphor and a stack of cards, respectively. In contrast, most contemporary literate computing environments draw upon the idea of a *notebook*—for some, it's even part of their names. This is characterized by a certain layout that forces a vertical, linear structure upon the work. Even if the linearity can be discontinued, e.g., by running code cells in a different order, the linear structure still persists, causing confusion: "Did I already run this cell?". In their study of more than a million Jupyter Notebooks, Rule et al. (2018) point out that almost half of these are non-linear in their execution. This is one of the many challenges inherent in the notebook format. Another is that the notebook metaphor does not support multiple purposes well, such as data exploration and data explanation, at the same time (Mathisen et al., 2019).

Codestrates, Mathematica and EVE are more akin to a Word-like document. For example, Codestrates structures the page in sections and paragraphs rather than cells. They do still, however, implement a vertical, linear structure. There is an inherent clash between the computer's non-linear execution model and the linear narratives presented in most current literate computing software. Functions, a core concept in most programming languages, violate the expectation of code running from line 1 to line N linearly—something most teachers in programming probably have experienced student confusion about. The difference between a notebook metaphor and a text document metaphor might seem superficial. However, the metaphor clearly signifies intention as we will see next.

3.2. Intention & Purpose

We might at a high level distinguish between multiple goals of programming: programming for computation, programming for interaction and programming for software engineering. Data science falls into the first category; the purpose of programming is not to construct running systems, but rather to use the environment as a glorified calculator and compute, evaluate, and present a given slice of the world. In contrast, programming for interaction is making a certain system come alive. In HyperCard, programming scripts did just that. Programming for software engineering is what some non-programmers would probably imagine a typical programmer doing—writing more or less complex software that can be installed, run, and utilized.

Characteristically, the platforms that embody a notebook metaphor typically have their roots in a very specific form of computing: data science. Data exploration, for instance, can be—and often is—a (sub-)goal for data scientists. Through data exploration, scientists *make sense* of their data, often followed by its counterpart activity, explanation (Rule, Tabard, & Hollan, 2018). That is, the subsequent sharing, presentation, or discussion of the findings from data analysis. A similar notion is found in the concept of *explorable explanations*—however, this concept entails creating finished compositions in which the end-user might explore an interactive paper, for instance. This brings it closer to the concept of computational media in general, which we might also see reflected in Eve and Observable.

3.3. Threshold & ceiling

These labels refers to how easy is it for programmers and non-programmers to get hold of, appropriate, learn, and eventually master a given system. Myers, Hudson, and Pausch (2000) offer the terms *threshold* and *ceiling* to illustrate, respectively, how difficult it is to learn to use a system, and how much can be done with it. They remark that most successful systems often have either low threshold and low ceiling or high threshold and high ceiling. HyperCard was exemplary for its low threshold—you could construct complicated systems without knowing much—or even anything—about programming. For instance, many educators in Melbourne took up HyperCard to create classroom activities and educational software (Lasar, 2019). Not considering the high price its and proprietary computational language,

Mathematica employs a very powerful symbolic manipulation. Once you know the basic syntax, (almost) everything can be expressed and computed.

In contrast, Observable has a much higher threshold. The language is JavaScript-but-not-quite, meaning there is likely a transitional phase for most people. Jupyter (and Python) is often lauded for its ease-of-uptake, but installing the environment still requires particular knowledge and advanced computer skills such as terminal use. Importantly, threshold and accessibility are very subjective measures. Theoretically, any system that is Turing complete can do anything that another can, while people naturally have different thresholds for systems. The question of threshold and ceiling is still valid, though. Even though you could build a 3D-shooter in Excel, does not mean that you ought to.

3.4. Liveness

There are several aspects of liveness, the first of which is related to how the environment behaves. Does it feel static, running code once, doing something, then coming to a stand-still? Or does it instead react to user input in a real-time manner, affording a dynamic, interactive environment. In other words, does the program *live*, in the sense that it has state, provides feedback, and exists over time? Some systems, such as Observable, Live notebook, Eve, and iodide utilize the latter principle to its fullest. Just the act of adding or removing a character triggers immediate reactions throughout the environment. These systems are *responsive*, but not truly *live*, as defined in Tanimoto's four levels of liveness (Tanimoto, 1990).

Liveness is a fickle attribute and might likewise refer to (re)-programmability. According to Basman et al. (2016), software can be considered live when the *divergence* is closed. Divergence is here defined as the opposition between the running software and the mental representation of said software. Hidden states, as for instance in Jupyter, might increase divergence, while a system like Boxer specifically sought to avoid this through, e.g., naïve realism.

3.5. Malleability & extensibility

How malleable and extensible an environment is, is really about its plasticity, both from the inside and outside. A classic example is Excel, allowing users to program their own extensions through macros. Likewise, the seminal text editor emacs allows users to extend the software in real-time by programming new functionality.

Codestrates is the most obvious example of a fully extensible environment in our list—the introduction of packages (Borowski et al., 2018) provides an easy way for developers to write extensions to the system itself through repositories. In contrast, environments such as Jupyter Notebook and Live notebook, while publicly available as source code, requires the user to edit said code from outside the system in a different editor. And some, like Mathematica or Observable, are impossible to change due to their being proprietary.

3.6. User community

We can divide the user community into two parts, the intended versus the actual community. It is not given that these are the same or even overlap. As a product of university research, Codestrates has mainly been taken up by researchers themselves and by their students. However, its flexibility makes it well-suited for many uses, and currently there are ongoing projects adapting Codestrates to citizen science, lab notebooks and high school education. In contrast, Jupyter Notebook was created by natural scientists for their community peers in the sciences. Like Mathematica, this creates a certain community—however, Mathematica's price tag clearly separates the actual user groups.

Two of the more interesting examples are Observable and HyperCard. Observable is explicitly oriented towards visualizations (Bostock, 2018), implying a user community that needs to communicate data analysis to outsiders, e.g., journalists. HyperCard, famously, was taken up by people who did not already know how to program. In doing so, many people ended up as happenstance programmers as part of a user community that was ill-defined from the start (*A Eulogy for HyperCard*, 2004).

3.7. Collaboration

Collaboration relates to the ways that collaboration is allowed, or even possible through an environment. For example, Observable and Jupyter Notebook are to a high degree based on solitary work practices. Recently, however, both tools have acquired after-the-fact support for collaboration either through Google Colab or natively (Ashkenas, 2018). In contrast, Codestrates and Live notebook natively support real-time collaboration.

There are other aspects of collaboration than real-time, equal collaboration. For instance, asynchronous collaboration ideally requires tools for managing and viewing changes that have happened between access. There are also different levels of *equality* in collaboration. Two people might collaborate equally, synchronously or asynchronously, each having the same access and responsibility. In contrast, one can also imagine, for example, a teacher collaborating with students, where the level of participation is unequal—maybe the teacher should only be able to read and comment on students' code. Vistrates is one environment that explicitly explores collaboration between unequal participants, where a programmer can edit the code of a visualization while a user interacts with it on another machine.

This theme links back to the systems' intended purposes—are they made for a single developer to write applications, do they support presenting data analysis findings, and does the collaboration imply an equal stake in the work?

3.8. The future of literate computing

We have presented a view of literate computing environments as it currently presents itself and brought to the fore seven salient themes. These are, of course, not exhaustive. Further, we would like to point at possible futures of these and other systems, as the design space of literate computing environments is far from fully explored.

The most prevalent expression of literate computing right now is the computational notebook. However, as we have presented, it has its drawbacks, such as the implied linear structure and the insistence in supporting *one* computational narrative. We see potential in expanding on the work by, e.g., Wood et al. (2019), which incorporates a more nuanced view of the narrative as a structuring factor. We might also do away with the narrative altogether, and instead scaffold literate computing systems with a spatial metaphor in line with, e.g., Boxer. Here, we would like to once again point to the software execution model. The idea of linearity is a myth in contemporary computing, and so we need to experiment with different forms of program flow. Vistrates, for instance, makes explicit the execution model by enforcing the creation of pipelines to structure the running program code.

Most current literate computing tools support only programming for computation. While it *is* possible, you would be hard pressed to develop sophisticated software architecture in Jupyter Notebook. It would likely prove fruitful to explore the borderlands between different goals of programming. One system does not need to support all goals. For instance, Jupyter Notebook is well-suited for programming for computation, but not for software engineering. In contrast, Codestrates is highly extensible and re-configurable, but this seems to hinder the ease of doing programming for computation.

Newer literate computing environments employ a reactive model in which any changes made are instantly reflected across the system. They can, however, still contain hidden states, obscuring the system's inner workings. We might look towards Eve's *absolute transparency* effort, Boxer's concept of naïve realism, or Bret Victor's essay on *learnable programming*[7] for inspiration on how to make the systems less opaque. Finally, we suggest that new system metaphors might serve as a starting point in exploring what literate computing could also be. The environments presented in this paper mainly employ notebook or essay metaphors. Just as the desktop metaphor in computing is ubiquitous but not essential, we might look to other real-world phenomena around which to construct new systems.

[7] `http://worrydream.com/\#!/LearnableProgramming`

4. References

Ashkenas, J. (2018, November). *Fork, Share, Merge.* Retrieved from https://observablehq.com/@observablehq/fork-share-merge

Badam, S. K., Mathisen, A., Radle, R., Klokmose, C. N., & Elmqvist, N. (2019, Jan). Vistrates: A component model for ubiquitous analytics. *IEEE Transactions on Visualization and Computer Graphics*, *25*(1), 586–596. doi: 10.1109/tvcg.2018.2865144

Basman, A., Church, L., Klokmose, C. N., & Clark, C. B. D. (2016). Software and how it lives on - embedding live programs in the world around them. In *Proceedings of ppig 2016.*

Borowski, M., Rädle, R., & Klokmose, C. N. (2018). Codestrate packages: An alternative to "one-size-fits-all" software. In *Extended abstracts of the 2018 chi conference on human factors in computing systems* (pp. LBW103:1–LBW103:6). New York, NY, USA: ACM. doi: 10.1145/3170427.3188563

Bostock, M. (2017, April). *A Better Way to Code.* Retrieved from https://medium.com/@mbostock/a-better-way-to-code-2b1d2876a3a0

Bostock, M. (2018, March). *Why Observable?* Retrieved from https://observablehq.com/@observablehq/why-observable

diSessa, A. A. (2001). *Changing minds: Computers, learning, and literacy*. Cambridge, Mass.: MIT Press. (OCLC: 464582529)

diSessa, A. A., & Abelson, H. (1986, September). Boxer: A reconstructible computational medium. *Communications of the ACM*, *29*(9), 859-868. doi: 10.1145/6592.6595

End-user programming. (2019, Mar). Retrieved from https://www.inkandswitch.com/end-user-programming.html

A eulogy for hypercard. (2004, Mar). Retrieved from https://due-diligence.typepad.com/blog/2004/03/a_eulogy_for_hy.html

Fauconnier, G., & Turner, M. (2003). *The way we think: Conceptual blending and the mind's hidden complexities* (1. paperback ed ed.). New York, NY: Basic Books.

Hayes, B. (1990). Thoughts on mathematica. *Pixel: the magazine of scientific visualization*, *1*(1), 28–35.

Head, A., Hohman, F., Barik, T., Drucker, S. M., & DeLine, R. (2019). Managing Messes in Computational Notebooks. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems - CHI '19* (p. 1-12). Glasgow, Scotland Uk: ACM Press. doi: 10.1145/3290605.3300500

Kay, A. (n.d.). Afterword: What Is A Dynabook? , 9.

Kay, A. (2013). The Future of Reading Depends on the Future of Learning Difficult to Learn Things. *VPRI Related Writings*.

Kay, A., & Goldberg, A. (1977, March). Personal Dynamic Media. *Computer*, *10*(3), 31-41. doi: 10.1109/C-M.1977.217672

Kay, A. C. (1993, March). The early history of Smalltalk. *ACM SIGPLAN Notices*, *28*(3), 69-95. doi: 10.1145/155360.155364

Kelleher, C., & Pausch, R. (2005, June). Lowering the barriers to programming: A taxonomy of programming environments and languages for novice programmers. *ACM Computing Surveys*, *37*(2), 83-137. doi: 10.1145/1089733.1089734

Kery, M. B., & Myers, B. A. (2018, October). Interactions for Untangling Messy History in a Computational Notebook. In *2018 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)* (p. 147-155). Lisbon: IEEE. doi: 10.1109/VLHCC.2018.8506576

Kery, M. B., Radensky, M., Arya, M., John, B. E., & Myers, B. A. (2018). The Story in the Notebook: Exploratory Data Science using a Literate Programming Tool. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems - CHI '18* (p. 1-11). Montreal QC, Canada: ACM Press. doi: 10.1145/3173574.3173748

Klokmose, C. N., Eagan, J. R., Baader, S., Mackay, W., & Beaudouin-Lafon, M. (2015). *Webstrates*: Shareable Dynamic Media. In *Proceedings of the 28th Annual ACM Symposium on User Interface Software & Technology - UIST '15* (p. 280-290). Daegu, Kyungpook, Republic of Korea: ACM

Press. doi: 10.1145/2807442.2807446

Knuth, D. E. (1992). *Literate programming* (No. no. 27). Stanford, Calif.: Center for the Study of Language and Information.

Lasar, M. (2019, May). *30-plus years of hypercard, the missing link to the web.* Retrieved from https://arstechnica.com/gadgets/2019/05/25-years-of-hypercard-the-missing-link-to-the-web/

Mathisen, A., Horak, T., Klokmose, C. N., Grønbæk, K., & Elmqvist, N. (2019). Insideinsights: Integrating data-driven reporting in collaborative visual analytics. *Computer Graphics Forum*, *38*(3).

Myers, B., Hudson, S. E., & Pausch, R. (2000, March). Past, present, and future of user interface software tools. *ACM Trans. Comput.-Hum. Interact.*, *7*(1), 3–28. doi: 10.1145/344949.344959

Papert, S. (1993). *Mindstorms: Children, computers, and powerful ideas* (2nd edition ed.). New York, NY: Basic Books. (OCLC: 28504839)

Pérez, F., & Granger, B. (2007). IPython: A System for Interactive Scientific Computing. *Computing in Science and Engieneering*, *9*(3), 21-23.

Pérez, F., & Granger, B. (2015). *Project Jupyter: Computational Narratives as the Engine of Collaborative Data Science.*

Perkel, J. M. (2018, Oct). Why jupyter is data scientists' computational notebook of choice. *Nature*, *563*(7729), 145–146. Retrieved from http://dx.doi.org/10.1038/d41586-018-07196-1 doi: 10.1038/d41586-018-07196-1

Rädle, R., Nouwens, M., Antonsen, K., Eagan, J. R., & Klokmose, C. N. (2017). Codestrates: Literate Computing with Webstrates. In *Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology - UIST '17* (p. 715-725). Québec City, QC, Canada: ACM Press. doi: 10.1145/3126594.3126642

Raskin, J. (2000). *The humane interface: New directions for designing interactive systems*. Reading, Mass: Addison Wesley.

Rule, A. (2018). Design and Use of Computational Notebooks. *UC San Diego Electronic Theses and Dissertations*.

Rule, A., Drosos, I., Tabard, A., & Hollan, J. D. (2018, November). Aiding Collaborative Reuse of Computational Notebooks with Annotated Cell Folding. *Proceedings of the ACM on Human-Computer Interaction*, *2*(CSCW), 1-12. doi: 10.1145/3274419

Rule, A., Tabard, A., & Hollan, J. D. (2018). Exploration and Explanation in Computational Notebooks. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems - CHI '18* (p. 1-12). Montreal QC, Canada: ACM Press. doi: 10.1145/3173574.3173606

Shen, H. (2014, Nov). Interactive notebooks: Sharing the code. *Nature*, *515*(7525), 151–152. Retrieved from http://dx.doi.org/10.1038/515151a doi: 10.1038/515151a

Somers, J. (2017, September). The Coming Software Apocalypse. *The Atlantic*.

Stephen wolfram blog. (2019, May). Retrieved from https://blog.stephenwolfram.com/2019/05/what-weve-built-is-a-computational-language-and-thats-very-important/

Tanimoto, S. L. (1990). Viva: A visual language for image processing. *Journal of Visual Languages & Computing*, *1*(2), 127 - 139. doi: https://doi.org/10.1016/S1045-926X(05)80012-6

Victor, B. (2011, Mar). *Explorable explanations.* Retrieved from http://worrydream.com/ExplorableExplanations/

Wood, J., Kachkaev, A., & Dykes, J. (2019, January). Design Exposition with Literate Visualization. *IEEE Transactions on Visualization and Computer Graphics*, *25*(1), 759-768. doi: 10.1109/TVCG.2018.2864836