Case Report

Engineering: Open Access

ISSN: 2993-8643

Documenting Code in Research: Integrating Mathematical Notation and Process Flow

Greg Passmore*

PassmoreLab, Austin, Texas, USA

***Corresponding Author**
Greg Passmore, PassmoreLab, Austin, Texas, USA.

Submitted: 2025, Nov 24; **Accepted:** 2026, Jan 29; **Published:** 2026, Feb 27

Citation: Passmore, G. (2026). Documenting Code in Research: Integrating Mathematical Notation and Process Flow. *Eng OA, 4*(2), 01-05.

Abstract
The paper advocates for a dual approach to software documentation that amalgamates mathematical notation and flowcharts. This approach aims to enhance both accessibility and quality control. Mathematical notations allow for precise understanding, while flowcharts provide a comprehensive view of complex systems. These components collectively broaden the research's appeal to a multidisciplinary audience and impose a level of scrutiny that augments reliability. The framework further bolsters code maintainability and extendability, contending that rudimentary inline comments are inadequate for thorough documentation. An illustrative example using deconvolution is included.

1. Introduction

In computational research, code functions as both the set of methods used and the environment in which the research is conducted. However, even wellcrafted code may pose challenges in interpretation, not just for the uninitiated but also for the researchers themselves at later stages. The intrinsic complexity of code can make it difficult to understand the underlying logic and operations being executed, thereby creating a barrier to reproducibility, a cornerstone of scientific inquiry. Add to this, the plethora of languages and the reader can struggle to interpret the meaning of the specific language [1].

The proposed solution to enhance the comprehensibility of code-centric research is to couple the code with rigorous mathematical notation and flowcharts. My example will be the process of simplified deconvolution as an illustrative example. Deconvolution is a mathematical operation aimed at reversing convolution effects on recorded data. In the realm of image processing, deconvolution is utilized to reconstruct or enhance images that have been subject to various forms of distortion, such as blurring or noise. It serves as an apt example due to its ubiquity in diverse applications ranging from medical imaging to astronomy, as well as its computationally intensive nature, which necessitates rigorous internal documentation for effective implementation.

I have chosen to elaborate on deconvolution for several salient reasons. First, the operation embodies a rich mathematical framework that encompasses a variety of algorithms and techniques, thus providing a fertile ground for discussing the interplay between mathematical notation and computational execution. Secondly, deconvolution is frequently encountered in image processing tasks that are often components of larger systems in computational neuroscience, a field inherently interdisciplinary, thereby necessitating precise internal documentation for effective crossdisciplinary communication. Lastly, the complexity and nuances associated with various deconvolution techniques make it a quintessential candidate for an exercise in robust methodological documentation.

Through the lens of deconvolution, I will explore how mathematical notation can be integrated seamlessly with code to construct a comprehensive and easily comprehensible internal document. I will also illustrate how the creation of flowcharts can aid in the elucidation of the process flow, thereby creating a roadmap that navigates through the various stages of the deconvolution process from initiation to conclusion.

Thus, deconvolution serves not merely as an example but as a case study that illuminates the virtues and necessities of meticulous internal documentation in research that culminates in code. This enables both the validation of the research and the replication of results, which are cornerstone principles in the advancement of scientific knowledge.

2. Mathematical Notation

In the realm of software documentation, the incorporation of mathematical notation is seldom practiced, a notable lacuna given its potential to enhance both clarity and comprehensibility. Traditional software documentation primarily focuses on textual descriptions, inline comments, and occasionally, flowcharts to elucidate the functionality and structure of the code. While these methods serve to communicate the basic architecture and purpose of a program, they often fall short of conveying the underlying theoretical constructs or complex algorithms with sufficient granularity.

Mathematical notation serves as the lingua franca for researchers across disciplines and geographic boundaries. By converting segments of code into mathematical expressions, we utilize a universal language that conveys the operational essence of the research. This allows for easier validation and replication by other researchers, as the mathematical underpinnings provide an unambiguous basis for understanding [2]. Furthermore, mathematical notation serves as an invaluable tool for debugging. Any discrepancy between the intended operations, as expressed in mathematical terms, and the actual code can often pinpoint the location and nature of software bugs.

Let's illustrate this with the basic equation for deconvolution. Here, we let I represent the input image, K is the deconvolution kernel and O the output image. So then, $I(x,y)$, $K(i,j)$ and $O(x,y)$ represents the intensity values of the pixel at coordinates (x,y) in the input image, the deconvolution kernel and the output image, respectively. We treat the RGB channels separately, but the same kernel is applied to each channel.

Our code then, implements this equation.

$$O(x, y, \text{ channel }) = \sum_{i=0}^{\text{kernelSize }-1} \sum_{j=0}^{\text{kernelSize }-1} [I(x+i, y+j, \text{ channel }) \times K(i, j)]$$

Like so many papers, there are land mines not immediately obvious. For example, the images must be padded as such:

$$P\left(x+\frac{k}{2}, y+\frac{k}{2}\right) = I(x,y) \quad \text{for all} \quad 0 \leq x < m, \quad 0 \leq y < n$$

where P is the padded image, pixels (x, y) are copied to the next position and the remaining pixels are a set value depending on the padding type.

For padding type of zero, we denote this as:

$$P_{\text{zero}}(x,y) = \begin{cases} I\left(x-\frac{k}{2}, y-\frac{k}{2}\right) & \text{if } x, y \in \text{ the original image bounds} \\ 0 & \text{otherwise} \end{cases}$$

For edge padding, we replicate edge pixels which is more appropriate for edge detection and texture analysis. It is denoted as:

$$P_{\text{zero}}(x,y) = \begin{cases} I\left(x-\frac{k}{2}, y-\frac{k}{2}\right) & \text{if } x, y \in \text{ the original image bounds} \\ 0 & \text{otherwise} \end{cases}$$

For mirror padding, we wrap pixel values around from the other side of the image. This is useful for VR and texture analysis. We denote this as:

$$P_{\text{mirror}}(x,y) = \begin{cases} I\left(x-\frac{k}{2}, y-\frac{k}{2}\right) & \text{if } x, y \in \text{ the original image bounds:} \\ I(\text{mirrored pixel coordinates }) & \text{otherwise} \end{cases}$$

Finally, for constant value padding, we use predefined values. This is useful for speciality applications, such as cleaning up image segmentation. It is denoted as:

$$P_{\text{constant}}(x,y) = \begin{cases} I\left(x-\frac{k}{2}, y-\frac{k}{2}\right) & \text{if } x, y \in \text{ the original image bounds} \\ c & \text{otherwise} \end{cases}$$

It is also not uncommon for the author to leave the padding unspecified, definitely a bad idea. There are, of course, problems with using equations for documentation.

Translating equations into code presents an intricate challenge that goes beyond simple syntactic conversion. Several key factors contribute to the complexity of this task. Mathematical equations are often succinct and encapsulate complex operations in a way that is independent of the execution environment. However, the same equation may not inherently specify algorithmic steps, including the order of operations or conditions for branching and iteration. Deciding on how to appropriately translate mathematical semantics into code involves making explicit what is often implicit in the equation.

Mathematical equations do not naturally express concerns about computational efficiency. While an equation might elegantly represent a relationship, the straightforward implementation of that equation may result in an inefficient algorithm. Issues such as time complexity, space complexity, and hardware utilization need to be considered to produce optimized code.

Mathematical equations deal with numerical values abstractly, typically with infinite precision. Translating them into code requires mapping these to concrete data types with limited precision, such as integers, floats, or doubles. This could lead to issues like overflow, underflow, or rounding errors, which can significantly affect the accuracy of the computational results.

Modern computational environments often offer opportunities for vector operations or parallel execution, which can significantly accelerate computations. The straightforward translation of an equation into a serialized form may fail to take advantage of such features, requiring a more nuanced approach to coding to fully utilize the available computational resources.

Often, developers must operate within the constraints of specific programming languages, libraries, and computational ecosystems, each with its own idioms, best practices, and limitations. The ability to effectively translate an equation into code often requires a deep understanding of these environments, and the available functions and data structures may not perfectly align with the mathematical entities involved in the equation.

3. Flowcharts

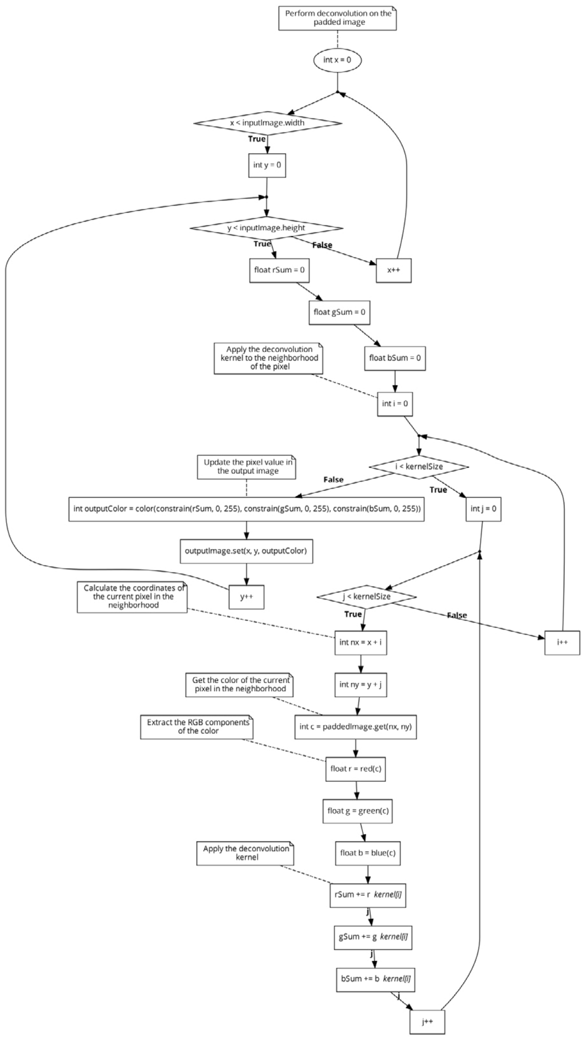
Flowcharts contribute a visual layer of understanding, supplementing the logical rigor provided by mathematical notation. A well-designed flowchart offers a high-level overview of the process flow, thereby revealing the architecture of the system or algorithm. This is particularly helpful for those who may find mathematical notation too granular or abstract. Moreover, flow charts foster interdisciplinary collaboration. In teams comprised of individuals with varying degrees of coding or mathematical proficiency, a flowchart can serve as a common reference point for discussions and collective understanding.

For our deconvolution example, our flowchart appears as below. Comparing the flowchart to the grace and succinctness of the equation illustrates the power of mathematical notation. This becomes especially clear when we combine all the above operations into a single equation as follows:

$$O^C_{x,y} = \text{clamp}\left(\sum_{i=0}^{P-1}\sum_{j=0}^{Q-1}\text{Pad}\left(I^C_{x+i-P/2,y+j-Q/2}\right)\times K_{i,j}, 0, 255\right)$$

There is beauty to the combined equation. The equation's succinctness aids in conveying the core logic of the algorithm to individuals who may not have the specialized knowledge to decipher source code. This quality is particularly beneficial for pedagogical purposes, where the equation can serve as a focal point for explaining the fundamental principles behind image deconvolution.

Moreover, the equation serves as a universal language that unifies multiple disciplines — from computer science and engineering to mathematics and signal processing. It opens the door for interdisciplinary research and innovation.


Perform deconvolution on the padded image
int x = 0
x < inputImage.width
True
int y = 0
y < inputImage.height
True
False
float rSum = 0
x++
float gSum = 0
Apply the deconvolution kernel to the neighborhood of the pixel
float bSum = 0
int i = 0
Update the pixel value in the output image
i < kernelSize
False
True
int outputColor = color(constrain(rSum, 0, 255), constrain(gSum, 0, 255), constrain(bSum, 0, 255))
int j = 0
outputImage.set(x, y, outputColor)
Calculate the coordinates of the current pixel in the neighborhood
y++
j < kernelSize
True
False
int nx = x + i
i++
Get the color of the current pixel in the neighborhood
int ny = y + j
Extract the RGB components of the color
int c = paddedImage.get(nx, ny)
float r = red(c)
float g = green(c)
Apply the deconvolution kernel
float b = blue(c)
rSum += r kernel[i]
j
gSum += g kernel[i]
j
bSum += b kernel[i]
j
j++

While flowcharts are helpful, they have their own limitations, especially in expressing the intricacies of more elaborate operations. When applied to complex systems involving numerous steps, branches, and loops, flowcharts may become unwieldy and challenging to interpret. The lack of depth is another significant drawback; although they offer a high-level overview of a given process, flowcharts cannot capture the specificities of implementation details like data structures and memory allocation.

Further complicating their usage is the issue of ambiguous interpretation. Even with standardized symbols, poorly designed flowcharts or those lacking in detailed annotations can lead to misunderstandings. Creating and updating these charts also demands considerable effort. The static nature of a flowchart makes them time-intensive to develop and revise, particularly when the process it represents undergoes changes. Unlike coded algorithms that can be debugged or tested, flowcharts do not offer a built-in mechanism for validation, requiring careful manual review for assurance of their accuracy.

Loops, by their very nature, involve repetition and conditional branching, which can become intricate to convey effectively in a two-dimensional flowchart. While it is possible to indicate a loop using a typical decision or process block, the resulting diagram can become cumbersome and confusing when the loop has multiple exit points, nested loops, or a variable number of iterations.
In a flowchart, loops are often represented with decision blocks that funnel back into earlier stages of the process, creating a cyclical pattern. While this can illustrate the basic concept of a loop, it does not capture the nuanced behaviors or the intricate conditions under which the loop will continue or terminate. A simple 'for' or 'while' loop might be easy to portray, but as the logic within the loop becomes more complex, the flowchart can become cluttered and harder to follow.

Additionally, flowcharts struggle to efficiently represent the concept of state changes within a loop. In programming, loops often include variables whose values change with each iteration. These variables can influence the behavior of subsequent iterations or even other loops. In a flowchart, demonstrating the continuous change of such variables can be laborious and can lead to the proliferation of additional blocks and arrows, which diminish the clarity and simplicity that flowcharts are intended to provide.

Moreover, when it comes to nested loops or loops that contain branching decision structures, a flowchart can quickly become overwhelming and complicated. Not only does this reduce the efficacy of the flowchart as a quick visual guide, but it also increases the likelihood of errors or oversights in its construction. Thus, while flowcharts remain useful for illustrating straightforward procedures or algorithms, they are often not the best tool for capturing the complexities inherent in loops.

The representation of multithreaded or multiprocessor code presents a unique set of challenges that make flowcharts an inadequate tool for thoroughly capturing the underlying complexity. Flowcharts are traditionally geared towards representing linear or conditional flow of control, making them well-suited for single-threaded applications where operations occur in a predictable sequence. However, multithreading and multiprocessing introduce concurrent or parallel execution into the equation, which is difficult to visualize in the inherently sequential and two-dimensional medium that is a flowchart.

The notion of time, or timing, is a pivotal aspect of concurrent programming that is not well-represented in flowcharts. Threads or processes in a concurrent environment may execute in an interleaved manner, with the operating system scheduler determining their execution order. This non-deterministic behavior makes it impossible to accurately depict the sequence of operations in a flowchart, which relies on a deterministic flow of control.

Multithreaded and multiprocessor systems often involve intricate synchronization mechanisms like locks, semaphores, or barriers to manage resource sharing and data consistency. Representing these mechanisms accurately in a flowchart can be exceptionally challenging. The flowchart would require multiple layers or dimensions to capture the dependencies and potential deadlocks that can occur in a multithreaded environment, which are far beyond the capabilities of a traditional two-dimensional diagram.

Additionally, the concurrency in multithreaded systems introduces a higher likelihood of race conditions, where the final state may be dependent on the timing of thread execution. Capturing the full range of possible race conditions would lead to an explosion in the complexity of the flowchart, making it impractical as a documentation or design tool.

Moreover, in a multiprocessor environment, where tasks may be distributed across multiple CPU cores or even different physical machines, flowcharts become increasingly inadequate for capturing data flow and dependencies between different components of the system. They cannot easily represent the layers of abstraction and complexity introduced by the concurrent execution of different tasks on different processors.

While their language-independent nature is generally considered advantageous, it can also serve as a limitation. Specifically, when features unique to certain programming languages or platforms are essential for understanding a process, the universality of a flowchart may become a hindrance. Additionally, flowcharts may lack the capacity to efficiently represent complex conditional or multi-threading processes, which often require auxiliary methods or notations to be employed.

4. Method Fusion

The synergy of mathematical notation and flowcharts creates a comprehensive documentation framework. Mathematical expressions delve into the specifics, providing the minutiae required for precise understanding and replication. Flowcharts, on the other hand, offer a bird's-eye view, simplifying complex systems into intuitive, visual components.

The integration of mathematical notation and flowcharts alongside code serves as a dual-pronged approach to effectively document research. This methodology brings several key advantages, notably in making the research accessible to a broader audience and acting as a self-imposed quality control mechanism [3].

Accessibility is a critical aspect of research, and relying solely on code to convey the nuances of an investigation can isolate individuals who may not be proficient in a particular programming language. Mathematical notation and flowcharts, being more universally understood, bridge this gap. They make the research findings more amenable to scrutiny and interpretation by professionals from various disciplines, thereby increasing the potential for collaborative advancements and cross-disciplinary applications.

The act of translating code into mathematical notations and flowcharts serves an equally essential purpose in quality control. In order to accurately represent the code in these alternate formats, the researcher is compelled to delve deep into the mechanics and assumptions underpinning the investigation. This scrutiny often uncovers oversights, ambiguities, or errors that may have otherwise gone unnoticed [4]. Moreover, this exercise imposes a level of rigor that enhances the reliability of the research outcomes. It necessitates that the code aligns with the mathematical models and process flows it is intended to represent, thereby ensuring consistency and accuracy across multiple representations.

Additionally, by encouraging this level of detailed understanding, the approach also makes the code more maintainable and extendable throughout the life cycle [5]. Should the need for modifications or upgrades arise, the documentation will provide an invaluable resource for any researcher or developer tasked with that responsibility. This is particularly significant in long-term projects or research that may involve multiple iterations or phases.

The purpose of this paper is to encourage a more robust documentation of research code. Long gone are the days of a few clever inline comments sufficing to hope that some unfortunate future coder will understand what is intended.

References

1. Fehr, J., Heiland, J., Himpe, C., & Saak, J. (2016). Best practices for replicability, reproducibility and reusability of computer-based experiments exemplified by model reduction software. *arXiv preprint arXiv:1607.01191*.
2. Uptegrove, E. B. (2015). Shared communication in building mathematical ideas: A longitudinal study. *The Journal of Mathematical Behavior, 40*, 106-130.
3. De Graaf, K. A., Tang, A., Liang, P., & Van Vliet, H. (2012, August). Ontology-based software architecture documentation. In *2012 Joint Working IEEE/IFIP Conference on Software Architecture and European Conference on Software Architecture* (pp. 121-130). IEEE.
4. Ozkaya, I. (2021). Can we really achieve software quality?. *IEEE software, 38*(3), 3-6.
5. Kumar, M., & Rashid, E. (2018). An efficient software development life cycle model for developing software project. *International Journal of Education and Management Engineering, 8*(6), 59-68.