“The reasoning behind this part is involved.”
— AGC Source Code
The Apollo Guidance Computer (AGC) code was primarily designed to be assembled and executed, not read and explored on the page. For those users of the several software emulators of the AGC, this is still an executable body of code. Yet this collection of code, like almost all code, is also a discursive object that registers and contains within its symbols, language, and self-understanding traces of its authorship, of its moment of production. Code, despite our ready assumptions of it as a set of concise, minimal, and utilitarian instructions, is an interpretable text. Code is a particular kind of polyvocal textual object. It is written for and addresses the particular software and hardware that define, to borrow a phrase from literary studies, what we might call its ideal reader. This reader is a particular platform with all its attendant affordances and limitations. Code, depending on the language and methods of abstraction, may very well run on other platforms without the work of porting, the translation of platform-specific code. Algorithms, of course, are generally platform-agnostic and can be reimplemented with relative ease. Code speaks, as it were, to multiple audiences and in multiple voices. There are multiple active discourses in much computer code and the AGC code provides contemporary readers with a particularly interesting site for examining the co-existence of these discourses.

But what sort of object is the AGC code? What sort of reading practices do we need to disentangle these discourses and interpret them? Should we consider code a text? Computer code, after all, is not — despite the way in which it is usually imagined by the public — constructed in ceaseless strings of 1s and 0s, but instead written using a standardized lexicon of textual signifiers, supplemented with some language-specific syntax. It is usually quite modular and organized into readable chunks with spacing and indentation used to enable comprehension. Code is almost always written and edited by humans. Almost every programming language borrows the major components of its syntax from a source “natural” language (this has been typically English) and programmers make logical and indeed creative and imaginative use of this language within both their code and their commentary.

Certainly, in the hands of cultural studies scholars, almost any object or action can be read as a discursively constructed text, from fashion to dance, from television programs to the Sony Walkman. Software, and especially computer code, can be understood as a cultural text because, as this book demonstrates, these texts are always constructed within the cultural constraints of the historical moment in which they were created and used. These constraints include, but are not limited to, the capabilities of particu-
lar hardware and supporting software libraries, major programming paradigms and languages, the so-called best practices of various programming communities, previously established methods and algorithms, the choices made by the few computer corporations that control the digital computer market, and the market available and constructed for the software product. For computational critic and theorist David Berry, code is a particularly important type of cultural text, because it simultaneously participates in several different registers. “Code,” Berry writes, “needs to be approached in its multiplicity, that is, as a literature, a mechanism, a spatial form (organisation), and as a repository of social norms, values, patterns and processes.”

Software, as the packaged and typically feature-frozen version of a selected configuration of code, touches more of these discourses and is under more of these constraints than the source code that typically remains hidden or obscured through the process by which it is compiled into machine executable software. But both software and source code register these frequently conflicting aspects of culture.

As a textually mediated mode of explanation and instruction written by a community of programmers and hackers, code shares much with other forms of textual expression, including literary texts. One powerful method by which we can examine the Apollo Guidance Computer code is through what literary scholar and theorist Caroline Levine calls the “new formalism.” Levine’s understanding of formalism is not limited just to the traditional aesthetic elements of formalism as used for decades within literary studies — the familiar practice of close reading that prompts the reader to cast her eye to language, lingering and dwelling on the appearance and significance of the words on the page — but also to a theoretically informed account of what Levine calls the “ordering principles.” She uses this notion of ordering in her gloss of this updated or “new” formalist method that examines, broadly, “an arrangement of elements — an ordering, patterning, or shaping.” Levine’s version of formalism pays close attention to the affordances of both literary, textual, and social structures — often these social structures are external to the text — and understands these various forms as not isolated phenomena but co-existing and in an informing re-

lation to each other. This is to say that the aesthetic forms used within any particular text can have political implications and that political forms may contain within them an aesthetic element.

In the theoretically informed readings of the AGC code that follow, the question of relation between social and aesthetic forms will continually reappear. In order to understand the AGC code and the multiple possible meanings produced and found within the code, we will have to shift the frame back and forth between different hermeneutical registers. This reading practice, like the code itself, might be thought of as modular and extensible.

The framework of the emergent field of critical code studies (CCS) provides, through the tacit agreement of the different possible critical perspectives, some possible methods through which we can frame and interpret the code. The close readings of code that follow will unpack and explain the purpose and aspirations of the displayed code segments. In so doing, the AGC software becomes visible as an important and readable cultural artifact and maybe even a work of art.

Computer software, cultural critic and theorist Lev Manovich tells us, is new media. Scholars working in the emergent field of software studies bring a range of critical resources, including ideological critique, formal analysis, and aesthetic criteria to bear on the design, construction, and everyday use of computer software. In several recent books, Manovich, one of the primary figures involved in the creation of software studies, asks us to take seriously the study of software, because software "mediates people's interfaces with media and other people." More and more, our everyday interaction with both local and global news, weather reports, text, audio, and video messaging, music, movies, games, directions, and access to knowledge itself is fully mediated by an array of personal digital devices and the software that presents and shapes these services and experiences. Software, in short, is culture. While there are different kinds of software, and many different ways of studying software, Manovich examines the use of media software. He defends his decision to study the mostly commercial creative media software used by cultural workers by pointing to the large and mostly anonymous user base of these packages. He argues that he wants to analyze what he calls "mainstream cultural practices" instead of the

---

4 Ibid., 31.
exception: those developing software or those involved in modifying or tinkering with existing software. This approach is roughly analogous to the arguments made by some scholars of popular culture.

While Manovich focuses on the way in which users interact with software, in particular those software packages that are used to create and access new media, other scholars have begun investigating the internals of software, the code that enables software to produce these functions and interfaces. Critical code studies (CCS) is an emergent approach to the study of software and the code that makes up this software that originates in the critical approaches offered by the field of cultural studies. Proponents of CCS argue that we can read code as an object for critical analysis; in the way in which cultural studies describes images and objects as a text, code may also be understood as a text.

David M. Berry makes an important distinction between code and software. He uses the term code to refer to the textual and social practices of source code writing, testing, and distribution. In contrast ‘software’ (as prescriptive code) will refer to the object code, that is, code that has been compiled into an executable format, which includes final software products, such as operating systems, applications or fixed products of code such as Photoshop, Word and Excel.²

Berry’s distinction depends on the division between executable, machine-readable software or compiled code and the source code that generates such software. This division is especially important to the commercial packages Berry mentions, Adobe’s Photoshop and Microsoft’s Word and Excel. These complex software packages are protected, controlled-access products. The code remains proprietary, a corporate secret, in order for the vendor—Adobe and Microsoft in the case of the packages mentioned by Berry—to sell access and, increasingly, automatically expiring subscriptions for the right to use the software products.

If, for David Berry, we should read code because code can give us insight into the software creation process, for Mark C. Marino, code is an important text in need of interrogation and critique because it offers a site for not

² Berry, The Philosophy of Software Code and Mediation in the Digital Age, 64–65.
just the analysis of software culture, but for the larger project of cultural analysis. Marino argues that code is a layer of discourse — presumably he means by this that code exists in some relation to other forms of cultural discourse — loaded with significance. It is a particular kind of cultural text, one “with connotations that are in conversation with its functioning.”

By this Marino means that the language that makes code work — the instructions, functions, and assignments — exceeds its instrumental value. Descriptive language — in his essay he highlights the naming of variables — makes something happen while also providing another type of meaning that is in excess of its functional value. While Marino’s variable names are an example of natural language — typically they encode meaning within their abstraction as pointers to data by naming the pointer itself — within code, the particular programmatic choices including spacing and even the organization of the code are subject to this form of critique. Extending the scope of CCS beyond the formal readings of source code, Marino claims that critical code studies “explores existing programming paradigms, but it also questions the choices that were made, examining among other aspects the underlying assumptions, models of the world, and constraints (whether technological or social) that helped shape the code.”

Scholars making use of CCS who work within cultural studies frame code as just another cultural, i.e., social text capable of revealing aspects of the culture that informed the writing of the code.

Because of the above issues involving the intersection of familiar or ordinary natural language appearing within code, the majority of debates within critical code studies and software studies has tended to discuss the philosophical nature of code and the relation between code, language, and writing. Alexander Galloway argues that code is different from writing, from language, because, in his account, code is a special type of language that he calls hyperlinguistic: “Code is a language, but a very special kind of language. Code is the only language that is executable.” Galloway provocatively describes code as “the first language that actually does what it

6 Mark C. Marino, “Why We Must Read the Code: The Science Wars, Episode IV.” In Debates in the Digital Humanities, eds. Matthew K. Gold and Lauren F. Klein (Minneapolis: University of Minnesota Press, 2016), 139.
7 Ibid., 140.
says — it is a machine for converting meaning into action.”

Language, one might argue contra Galloway’s assertion, can do things, but he wants to make a distinction within code by introducing what he calls an executable state to his understanding of language:

Code has a semantic meaning, but it also has an enactment of meaning. Thus, while natural languages such as English or Latin only have a legible state, code has both a legible and an executable state. In this way, code is the summation of language plus an executable metalayer that encapsulates that language.10

Code, of course, does not always do exactly what it says it will do — it is interpreted, by a compiler or interpreter, and the meaning of the code might not be the same meaning as the execution. Galloway concentrates mostly on compiled languages such as C and C++, in which the code is transformed into executable instructions by a compiler. Compilers (usually) create object code or bytecode, an essentially lower-level set of instructions that are optimized for system-specific hardware, including central processing units (CPUs) or virtualized systems (in the case of Java).11 The notion of code as doing what it says becomes more complicated and less and less true as we add layers of abstraction and modularity. Because of Galloway’s emphasis on compiled rather than interpreted languages, he tends to treat code as separable from its instruction. Interpreted languages are one step closer to programmers than the compiled languages critiqued by Galloway; the code is interpreted and executed by the interpreter as written, in its initial state. Interpreted languages are also subject to the critique of complex systems that will follow, but in general interpreted languages stay within what Galloway terms a legible state. Highly specialized and opaque code, as this book demonstrates, needs the supplement of natural language to make its meaning legible for human readers. This supplement renders the text of

9 Ibid., 165–66.
10 Ibid., 166.
11 The target, by which we mean an audience that must be addressed and included for compiled code. The target for most compiled “C” code on a modern Linux system is an optimized and dynamic stack of libraries. This target platform is described formally by the operating system as such: “ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically linked (uses shared libs), for GNU/Linux 2.6.18, stripped.”
the AGC code a complex configuration of writing, a space-age entanglement of meaning making that invites the full resources of critical analysis to unpack and explore.

John Cayley, who helped inaugurate critical code studies and code poetics with his essay “The Code is not the Text (unless it is the Text),” helps us to think through this complex problem of the audience for code:

If a codework text, however mutually contaminated, is read primarily as the language displayed on a screen then its address is simplified. It is addressed to a human reader who is implicitly asked to assimilate the code as part of natural language. This reading simplifies the intrinsically complex address of writing in programmable media. At the very least, for example, composed code is addressed to a processor, perhaps also addressed to specific human readers (those who are able to ‘crack’ or ‘hack’ it); while the text on the screen is simultaneously? asynchronously? addressed to human readers generally. Complexities of address should not be bracketed within a would-be creolized language of the new media utopia.¹²

Cayley is interested in a possible poetics of code and locates his investment in complicating the lines between code and text in his naming of the text of code “codework.” Cayley positions his codework as addressed simultaneously to the machine and the human reader. Doing so enables him to resist the separation between what appears on a screen or device and the code that brings this digital appearance into being. For Cayley, the audience of code must always include the possibility of a human reader.

The question of audience and code legibility persists within CCS. N. Katherine Hayles follows Galloway’s understanding of code as distinct from the natural language associated with writing because of its function and its primary audience. She argues that despite the possibility of human readers, code is written primarily for machines, for a computer:

Although code originates with human writers and readers, once entered into the machine it has as its primary reader the machine itself. Before any screen display accessible to humans can be generated, the machine must first read the code and use its instructions to write messages hu-
mons can read. Regardless of what humans think of a piece of code, the machine is the final arbiter of whether the code is intelligible.\footnote{13 N. Katherine Hayles, \textit{My Mother Was a Computer: Digital Subjects and Literary Texts} (Chicago: University of Chicago Press, 2005), 50.}

This difference is what enables her to construct a successive genealogy for “the three major systems for creating signification”\footnote{14 Ibid., 39.}: “In the progression from speech to writing to code, each successor regime reinterprets the system(s) that came before, inscribing prior values into its own dynamics.”\footnote{15 Ibid.}

For Hayles, this process of reinterpretation does not necessarily obsolete the prior regime, but it does produce extensions and alterations that fundamentally exceed the capacity of the previous system to describe the new world inaugurated by the new regime. “One of Derrida’s critical points,” Hayles argues, “is that writing exceeds speech and cannot simply be conceptualized as speech’s written form. Similarly, I will argue that code exceeds both writing and speech, having characteristics that appear in neither of these legacy systems.”\footnote{16 Ibid., 40.} Hayles’s use of “legacy system” produces a shift, but it is not as dramatic of an obsoleting shift as it sounds — she calls speech and writing “vital partners on many levels of scale in the evolution of complexity.”\footnote{17 Ibid., 55.}

In a later work, Hayles doubles down on her argument that code must always be considered executable and that is always addressed to a specific interpretive community, the machine:

If the transition from handwriting to typewriting introduced a tectonic shift in discourse networks, as Friedrich Kittler (1992) has argued, the couple of human institution and machine logic leads to specificities quite different in their effects from those mobilized by print. On the human side, the requirement to write executable code means that every command must be explicitly stated in the proper form. One must therefore be very clear about what one wants the machine to do.\footnote{18 N. Katherine Hayles, \textit{How We Think: Digital Media and Contemporary Technogenesis} (Chicago: University of Chicago Press, 2012), 42.}
Despite the claims made by Galloway and Hayles, we cannot guarantee that the instructions will be executed as written because of the various levels and layers of abstraction involved in computing. The expected execution of even compiled code can be altered. Depending on the language and system used, there are multiple layers of interpretation and transformation that take place between the writer of code and the final execution of instructions. Modern computing systems are constructed from modular components, both software and hardware, and these components continually abstract any set of instructions.

This abstraction, which has been increasing throughout the past few decades, enables programmers to write shorter and simpler code—commonly used routines and procedures are frequently supplied by the operating system. Even if the programmer does not choose to use one of these supplied functions, many components of the software might be substituted by the operating system or by end users. These can be optimized for specific hardware (such as a device to offload certain operations to a Graphical Processing Unit or GPU) and software configurations. In the case of closed-source operating systems such as those supplied by Microsoft, these libraries contain well-known functions that enable software developers to write applications with a similar look and feel. Open-source platforms also make use of these types of libraries but also contain a large collection of libraries from other tools that contain these frequently used functions.

All of this is to say that the programmer cannot have any sort of guarantee that the code will be executed as written.\(^{19}\) Code resembles more of wish than a command. Wendy Chun has provided one of the most pointed critiques of Galloway and Hayles’s position. She takes issue with the reduction of software “to a recipe, a set of instructions” and argues that code is devious and crafty.\(^{20}\) Chun demonstrates this by pointing to the layering involved in complex computer systems and the fact that because of the it-

---

19 Rita Raley complicates this understanding by asking us to consider the difference between code and computation. She does so by analyzing code that is not nor can never be executed and raises questions about the “function” of code specifically designed to fail or crash, in which its failure becomes precisely its successful function. See Rita Raley, “Code.surface || Code.depth,” Dichtung Digital 36 (2006), http://www.dichtung-digital.org/2006/01/Raley/index.htm.

ervative development cycle of software, “source code only becomes a source after the fact.”\textsuperscript{21} The “fact” of computation, in Chun’s argument, requires the successful execution and testing of code. Execution makes and names the code that was executed “the source” for the executed code. The source code then might be said to retroactively become a re-source. Chun breaks with the normative understanding of code to expose what she calls the fetish logic of code:

code as fetish thus underscores code as thing: code as a “dirty window pane,” rather than as a window that leads us to the “source.” Code as fetish emphasizes code as a set of relations, rather than as an enclosed object, and it highlights both the ambiguity and the specificity of code. Code points to, it indicates, something both specific and nebulous, both defined and indefinable. Code, again, is an abstraction that is haunted, a source that is a re-source, a source that renders the machinic — with its annoying specificities or “bugs” ghostly.\textsuperscript{22}

Chun calls the belief that the only meaning of code could be what it does a form of “sourcery” that is in fact a fetish covering over the deviations between execution and code. The retroactive process that makes code a source after its “correct” execution leaves marks, leaves traces within the code — both within the functions and commands and within the natural language found within code comments.

Friedrich Kittler refers to the above referenced hierarchical layering of languages and instructions as a “postmodern Tower of Babel” that has produced a fog of interpretive confusion that covers over the gaps between instruction and execution — so much so that he argues that “we can simply no longer know what our writing is doing, and least of all when we are programming.”\textsuperscript{23} While some might take this confusing stack of instructions as provocation to examine computation, to turn to the task of translating the particularities of a language or machine-specific instructions into a common code, one for a universal computer, code (at least successfully executed code) is inscribed with the signs of being run through a configura-

\textsuperscript{21} Ibid., 24.
\textsuperscript{22} Ibid., 54.
tion of hardware and software and these signs bear the traces of culture, of the programmer’s membership within communities of practice. This is one that that we can be sure of when we talk about that type of writing called programming: when one writes code, one works with conventions. There might be only iterations of utterly conventional code to be found or perhaps when reading code we discover a range of imaginative and creative extensions, elaborations, and elegant appropriations. Programming might attempt to present itself as a form of wizardry or sorcery but it is ultimately the use of a communal language used by a certain type of desiring machine that is humon, all too humon.

The interpretive practices outlined above make the AGC code available to a wide range of contemporary readers. Potential readings include an antiquarian desire to take what might call a software archeological dig into this historical code or a culture critique that seeks to unpack the ways in which the functions, commands, and comments register the conditions that made the creation of this particular body of code possible. The esoteric and the aesthetic are combined and interleaved throughout the lines of this code and this combination invites reading with and against the grain. Historicizing, critiquing, and appreciating the language structuring the earlier years of programming and digital computers makes it possible to shift and ultimately shuttle our attention back and forth through the long history of computing, adding insight to both the past and the present of digital culture.

**Reading Code**

In order to help frame and make more concrete some of the objections and questions raised by the above arguments and their claims for the interpretation of the AGC code, we can turn to some contemporary and highly simplified examples of computer code. The following are lines of code written in a high-level interpreted programming language called Python. Python programs remain (generally) in textual or “source” form. These instructions are “read” and interpreted by the Python interpreter, itself written in the C programming language and compiled for a specific computing platform (for example, macOS running on the x86_64 CPU). This fragment of a program defines (def) a function named euclidean_distance. The function operates on two supplied input parameters (input1 and input2). Functions are the building blocks or components of well-designed larger programs. They
enable more efficient and readable code by bundling together instructions that might be used multiple times within a single program. Functions, in Python and other programming languages, can be thought of as the addition of new instructions to the existing language resources. The euclidean_distance function calculates the “distance” between the two supplied parameters by taking the square root of the summed squared differences between the input objects supplied as the parameters.

```python
def euclidean_distance(input1, input2):
    d = 0
    for i in range(len(input1)):
        d += (input1[i] - input2[i])**2
    return d**(.5)
```

Within the function, we first set the value of a new variable `d` (for distance) to 0. Following this, the function will loop (for `i`) through each component or “item” of the supplied input objects, adding to the variable `d` the squared differences between the input items. Once the loop is completed and we’ve reached the end of the supplied input, we return back to the calling function the square root of the summed values stored as `d`.

When the `euclidean_distance` function is correctly called with the appropriate parameters, it returns the distance between these parameters in Euclidean space. Euclidean distance is defined as the shortest straight path between two points in a common, uniform geometrical space. As an example, first imagine a simple one-dimensional space, a line, with two points. One point on the line is 8 and the other 64. To calculate the Euclidean distance between these two points, we subtract the second point from the first and square the result and then take the square root. Using the `x**y` notation in Python to calculate `x` raised to the power of `y`, we can find this result with: `((8-64)**2)**.5`. Using our `euclidean_distance` function, we can print these results with:

```python
euclidean_distance([8],[64])
```

The basic Python system provides a set of functions embedded within a package called “math” that handles some of these calculations with a little more grace and enables greater readability. Instead of calculating a square root with `x**.5` we can ask Python to make the math package available (im-
import math
def euclidean_distance(input1, input2):
    d = 0
    for i in range(len(input1)):
        d += pow(input1[i] - input2[i], 2)
    return sqrt(d)

This trivial example demonstrates that there are many different ways to solve the same problem, some more comprehensible and elegant than others. Elegance in this case includes using the affordances and norms of the programming language — for Python language programs, that means writing code in a manner playfully termed "Pythonic." The choice to use pow and sqrt signals the author's participation in a writing and interpretive community organized around the use of these Pythonic norms.

We can now use our same euclidean_distance function with two-dimensional data. To calculate the shortest distance between two points in a simple x,y coordinate system, we would simply call the function as such: euclidean_distance([-2, 2], [2, -1]). The function returns "5.0" as the Euclidean distance between these two points. Higher-dimension data can be supplied in a similar manner. For example, we can take the measurements in centimeters of two Iris flowers that are part of Ronald Fisher's 1936 Iris dataset. For each flower, we have the length and width of the sepal (5.1 and 3.5 for the first flower) and petal (1.4 and 0.2 for the first flower). To calculate the distance between these two flowers in this four-dimensional common, uniform geometrical space, we would simply call our function as such: euclidean_distance([5.1, 3.5, 1.4, 0.2], [4.9, 3.0, 1.4, 0.2]). The "distance" in this shared space between these two flowers is returned as "0.5385164807134502." But what does this distance mean? The possible meanings this distance might have depend on the rest of the dataset. Are these two measurements representative of the phenomena that we wish to

measure (i.e., a natural distribution)? Did we choose the correct parameters (sepal and petal) and measurement metrics (length and width) to make meaningful comparisons?

These simple lines of Python show just some of the possibilities and constraints of programming languages. We have two functions that accomplished the same task but used different methods to reach this result. The revised function is better and yet can be improved in numerous ways. These few lines of code contain within them many assumptions about the input parameters. Taken together, this function encapsulates an understanding of a geometrical space that is in many ways only an ideal. Formally, the function produces the results requested but it operates in concert with its data. This idealized geometric space is created only through data and thus the function cannot be isolated from the “assumptions” held by both the formula it renders as code and its data. This function would typically be used by another that makes use of the returned distances. Euclidean distance, for example, is often used with classification algorithms, including k-nearest neighbor, an algorithm that uses the distances between data labeled as members of existing classes of objects of a similar kind to determine the membership of previously unseen and unlabeled data. The meaning of this code fragment within the implementation of k-nearest neighbor would raise new questions. Is Euclidean distance the appropriate distance metric for this algorithm? What are we attempting to classify? Do these objects all belong to the same space? What might that mean?

Code is created to solve problems. The problem space is cultivated and constrained by understandings of how these problems will present themselves or be presented, especially in the form of data. We can also add the included and instrument-sampled data as another discourse to those mentioned above. Much of these data were ephemeral. They are no longer available, detected and processed in the moment. Some were anticipated and part of the exhaustive testing procedures and others were entirely unpredicted.

When examining code, we read and interpret the instructions, imagine or attempt execution. A critical account of the source code of the AGC needs


to examine the problem space that gave shape to the code. What were the technical and social constraints? How did these limit the functions and possibilities of the AGC hardware and software? In examining the code, we need to at least attempt to historicize and bring into understanding the execution environment, the computational and cultural situation, that retroactively named this particular text the source code that brought humans to the Moon.