R00002

The AGC and Computing in the 1960s
The digital computer that powered the Apollo 11 Command Module and Lunar Module was sophisticated and equipped with many leading-edge and advanced features despite its compact size and limited fixed and erasable memory capacity.¹ While the majority of computing systems in the 1960s, including the several computers used to compile and debug the AGC software, occupied large spaces in dedicated air-conditioned rooms, the main hardware that made up the AGC computer was stored in a package that measured twenty-four by twelve inches, six inches deep, and weighed a mere seventy pounds.² The complete program or software for the AGC was stored in a form of magnetic memory called rope core memory. In this resilient but quite limited storage scheme, a network of wires run through small ferrite core rings store the instructions for the computer. This was fixed, read-only memory as any changes to the instructions required extensive rewiring. The computer was hardwired, as it were, with actual wires that, when selected, signal the 1s and 0s of the binary instructions required to bootstrap or bring the computer into operational status. While the programmers wrote and edited code in their Cambridge, Massachusetts offices, a large group of mostly female workers a few miles away at Raytheon in Waltham — the company that held the NASA contract to produce the computers — programmed and installed the software by twisting and braiding thousands of wires through the ferrite rings.

The transformation of the instructions from symbolic code punched line-by-line through stacks of cards to densely packed sets of wires demonstrates the existence of the multiple shapes and forms — all of which are expressions of the language of computing. Throughout the flow of this language, from initial composition to contemporary methods of digital preservation of the AGC code, we see many such transformations taking place. The code, through many of these conversions, changes its orientation and its form. We might call these code variations versions or perhaps even editions. Calling our attention to the importance of formal features, including style and shape to the interpretation of any digital object, Dennis Tenen would have us recognize these variations as distinct formats of the code.

¹ The best guide to the design and operation of the AGC hardware and software is Frank O'Brien, The Apollo Guidance Computer: Architecture and Operation (Chichester: Springer, 2010).
“Formats,” Tenen argues, “shape the very structure of interpretation. The seemingly innocuous formatting layer contains the essence of control over the mechanisms of representation. Long a marginal concept in literary theory, formatting is therefore central to the contemporary practice of computational poetics. More than embellishment, formats govern the interface between meaning and matter, thought and page.”

But it is crucial to recognize that these formats do not necessarily operate in a progressive manner in which the appearance of a new format obsoletes the prior formats. The temporality of code authorship, in particular the code under consideration in this book, is quite complicated; some representations of the AGC code predict future, by which we mean post-processed and collated, forms and formats, while others alter the overall organization of the code and in so doing introduce different meanings to readers and interpreters.

We might understand the multiple formats of the AGC code as a form of what Jay David Bolter and Richard Grusin called remediation. For Bolter and Grusin, remediation is an attribute of media, especially but not limited to digital media, in which the cultural imperatives of immediacy and hypermediacy meet through the multiplication and erasure of media. New media borrow and remake old media in order to produce a sense of immediacy. We can frame newer presentations of the AGC code through the desire to cut out what is now considered extraneous, for example, the line numbers and page headers. In cropping the code and making these headers and numbers marginal, the code becomes more readable, but it has now lost the sense of order that structured the prior format. In producing a remediation of the printed pages in order to present the code as if it were authored in a contemporary high-level programming language, the contemporary programmers have erased the medium-specific features of the earlier code. Of course, the printed pages of code were themselves already making use of a remediated format that erased the medium-specific features of the punch card by turning each card into a line of printed code and by creating page headers and line breaks to increase the readability of the code for the programmers.

We might then apply an additional hermeneutical turn and examine yet another reformatting of the AGC code as it passed through the assembly line

---

of software production. The instructions on the punched cards were collected and processed by a set of software programs running on a conventional digital computer. The output of the assembler system was the rope wiring diagrams that were sent to Raytheon, the company holding the contract to produce the AGC hardware. The wiring diagrams reformatted the instructions and were installed or programmed through the threading, braiding, and twisting of small, thin wires through sets of rings (Figure 1). These memory ropes were used to preserve and package the software for the AGC computer. Each reformattting presents a new material shape for the instructions and incorporates another entire set of labor. That this labor, especially that of the women or “girls” who reformatted the code into twists and braids, was the product of what we might want to call “hidden figures” is a function of both the valuation structures of the 1960s managerial system that unevenly distributed credit and the remediation at the core of all reformattting.

These women, like the programmers at the MIT Instrumentation Lab, worked collaboratively, passing the delicate wires back and forth as they embedded instructions into the hardware (Figure 2). If we want to study the definitive text that took the astronauts to the Moon, then the proper object of study must include the labor and products of these memory rope programmers. The software development cycle was not limited to the Instrumentation Laboratory and the code was not produced within a closed system of programmers; it required the collaborative work of thousands of people found in numerous organizations and it flowed forward and backward through these networks of people, much like an electrical current through any integrated circuit.

The majority of the code shown throughout this book was authored in one of two fairly low-level single-purpose symbolic languages, Basic (this language has no relation to the much more user-friendly interpreted BA-

---

Figure 1. A Raytheon employee creating Apollo rope memory. Screenshot from Computer for Apollo, directed by Russell Morash (Cambridge, MA: MIT Science Reporter, 1965).

Figure 2. Teamwork was required to braid the wires through the cores. Screenshot from Computer for Apollo, directed by Russell Morash (Cambridge, MA: MIT Science Reporter, 1965).
SIC, or Beginners All-Purpose Symbolic Instruction Code, language that was developed contemporaneously at Dartmouth College) and what was called Interpretive. The code contains both Basic instructions—the Basic syntax contains only forty instructions or “opcodes,” eight of the most common are found in Table 1—and the more flexible but much slower Interpretive instructions that were executed by a program called INTERPRETER. Basic was also known as “Yul.” It was named Yul by Hugh Blair-Smith because the language was developed for the original AGC Model 1A that was planned to be completed around Christmas time in 1959 (the 1A, according to Blair-Smith, was called the “Christmas Computer”), hence Yul for Yuletide. Yul was not so much a language as a system. It included testing systems and a special piece of software known as an assembler that transformed the Yul code or text, much like other high-level compiled languages like C, into a lower-level machine code and finally produced the wiring diagrams mentioned above.

The Yul system was designed to enable the programmers to quickly compose, edit, and test code before it was generated as the read-only, permanent instructions stored in the AGC’s rope memory. Like several other programming languages of the period, the two languages used in this code are fixed format languages. This means that the format of the code was imagined and printed on pages needed to take a specific form in order for conversion routines to produce the correct instructions for the digital computer that would eventually execute the instructions. The AGC code was written and edited under numerous constraints, including the rigidly fixed format required by the punch card and its associated hardware as well as the limited syntax of its major programming languages.

The AGC software was designed in an era before software as we understand it today was invented. The systems and code were imagined, designed, and edited not in a digital environment, with the array of graphical display devices, easily movable sections of text, searching mechanisms, and versioning information, but almost entirely in print and on paper. This high-tech digital computer system belonged to the world of print and it was thus imagined by the software engineers as an almost literary object.

<table>
<thead>
<tr>
<th>Code</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>TC</td>
<td>Transfer Control</td>
</tr>
<tr>
<td>CCS</td>
<td>Count, Compare, Skip</td>
</tr>
<tr>
<td>INDEX</td>
<td>Modify Next Instruction</td>
</tr>
<tr>
<td>XCH</td>
<td>Exchange</td>
</tr>
<tr>
<td>CS</td>
<td>Clear and Subtract</td>
</tr>
<tr>
<td>TS</td>
<td>Transfer to Storage</td>
</tr>
<tr>
<td>AD</td>
<td>Add and Count on Overflow</td>
</tr>
<tr>
<td>MP</td>
<td>Multiply</td>
</tr>
</tbody>
</table>

*Table 1. Major Basic or Yul Instructions*
The literary “print” metaphor drives the majority of our thinking about the prospects for interpreting the code. For while it was the depositing of the digitized code within Github, a collaborative online code repository, that initially brought the text of the Apollo Guidance Computer code to our attention, the metaphor of the printed code as an imagined and interpretable text remains our doorway into this project and into its historical moment. The programmers had to work simultaneously with at least two different formats of printed code: collated code listings and punch cards. In his memoir, *Sunburst and Luminary: An Apollo Memoir* (2018), Apollo Guidance Computer programmer Don Eyles links the writing of code to the writing of prose by reflecting on writing as a process: “Some of us wrote out our programs fully on paper forms before we sat down. Others programmed as they punched. I usually started with rough notes and wrote very much as I am writing at this moment.”

The AGC code is a highly revised, co-authored text. It was written line by line. Each line of eighty-character instructions was entered by hand, punched on an IBM 026 keypunch. But the code was imagined, always, and edited as a listing—it was collated and printed in page form, after being run through (each reading of the code was called a “pass” and several “passes” were required to fully format and process the list of instructions) different assembler programs. These assembler programs ran on the same larger general-purpose computer that processed the stack of punch cards. During the time of the Apollo 11 mission, this computer was a Honeywell 800. The final pass was known as the “wiring diagrammer” and it produced the wiring diagram tapes that were sent directly to Raytheon. The AGC code was thus produced under numerous constraints, including the rigidly fixed format required by the IBM 026 keypunch mechanism along with the Honeywell 800 card reader and the limited syntax of its major programming languages. The programmers, therefore, had to be flexible in their imagination of what the code would look like and how it would function when it was transformed into these other formats.

Consider the now iconic image of Margaret Hamilton with the stock of AGC code almost reaching her own height (**Figure 3**). This image referenc-

---

Figure 3. Margaret Hamilton standing next to stack of Apollo Guidance Computer code. Courtesy of the MIT Museum.
es and reworks other depictions of programmers, especially women, with the material embodiment of code. Computer company advertising, for many years of its early existence, used images of women appearing next to stacks of punch cards, storage devices, and other equipment. This photograph of Hamilton references and reconfigures the advertising image to position her and her body as the signature that authorizes the presented code. The code is “embodied” both in the sense of the presentation of the complete body of the text, as well as the reference image, the human body, that serves to measure the length of the code. The concept of software and the engineering of software were essentially being invented at this moment. Comparing the code to the body made it concrete by presenting it in a familiar form and scene.

That code that we see represented as stacks of printed pages or displayed as modular functions and routines stored in separate files within the Github repository was initially authored in short eighty-column segments on 3 1/4 x 7 1/2 IBM paper punch-cards (referred to simply as “cards” with the body of the code). A card reader sorted and compiled the individual cards into the text of the complete code for the AGC and it was then printed on wide pages on a Honeywell printer. The code authors produced small sets of instructions and commentary on the code on individual punch cards, but imagined the collaboratively constructed code as a numerically ordered set of cards, with each card forming a line of code and eventually printed on pages. The code presents and understands itself in paginated terms. There are numerous times in which the code references other “paragraphs” and pages of the code — there were 1,743 pages of code in the “LUMINARY 1A” portion of the AGC code of July 14, 1969. In the imaged scans of the printed code that are available, we can see that the text of the code was printed on a continuous stream of paper with alternating colored lines. Each printed page (see Figure 4) contained a short header that provided metadata about the assembly and printing of the code, including the revision, the current date and time, and the page number.

The code and language used throughout the AGC project is simple and rather utilitarian. As mentioned above, this was due to the limited memory of the AGC computer and the constraints of the coding environment. The following table reproduces information from the Programmer’s Manual and
Figure 4. AGC Source Code, page 392.
illustrates the prescribed content for each column or punch card position for each line of the AGC code.  

<table>
<thead>
<tr>
<th>Columns</th>
<th>1–7</th>
<th>Card Number and Card Content Control</th>
</tr>
</thead>
<tbody>
<tr>
<td>Columns</td>
<td>8</td>
<td>Vertical Spacing Control</td>
</tr>
<tr>
<td>Columns</td>
<td>9–16</td>
<td>Location Field</td>
</tr>
<tr>
<td>Columns</td>
<td>18–23</td>
<td>Operation Field</td>
</tr>
<tr>
<td>Columns</td>
<td>25–40</td>
<td>Address Field</td>
</tr>
<tr>
<td>Columns</td>
<td>41–80</td>
<td>Remarks Field</td>
</tr>
</tbody>
</table>

Table 2.1: Punch card organization

The card numbers, columns 2–7, indicated what would become the line number of the card when it was collated and printed. Like the line numbers used in the popular BASIC programming language, these card and line numbers were used to organize code and to enable some basic editing and revision. These were incrementing numbers and each card inserted into the card reader was required to be a larger number than the previous card. It appears that the programmers planned to use four- or five-digit numbers (the majority are five digit numbers). If blank, the value of empty columns was equivalent to zero, enabling the proper sorting of any number of cards. The first card of the Luminary 1A program was numbered R00001 and was followed by R00002 and then R000025. The addition of the sixth column for the third card demonstrates an important feature of the code: it was designed to enable the addition of new code and the minor revision of existing code without renumbering and thus repunching the entire body of the code. This is enabled projective thinking – the imagination of future revisions by leaving possible empty space, an area of expansion and breathing room for the existing code. If new code was required to add a feature or extend particular instructions, these additional cards using six-column numbers...

---

9 Ibid., 54.
could be inserted between existing cards that used five-column numbers. With this scheme, nine lines could be added (XXXXX1–XXXXX9) without introducing a major revision. To correct a minor error, the programmer would just need to revise that single card.

The cards beginning with the character R were known as “remarks cards.” While the Yul language specifications defines specific column markers, for free-form explanatory or other forms of commentary in each card, these remarks cards mark the entire card space as remarks. These cards, like the contents of columns 41–80, lack any explicit requirements or standards. Remarks cards, such as the first three cards invoked above, were used for various functions. At the most simple level, most code is marked by two voices: the code and embedded remarks or commentary. Commentary is a supplement; not necessary for execution but essential for its comprehension and future modification. Code commentary speaks to the past, this is how this works and why we did this, but primarily it addresses future readers — reminders, warnings, justifications. The cards that make up the page of code in Figure 4 provide explanatory notes about the code that appears in the following cards. Cards R0072 and R0073 explain the general purpose of the remarks cards throughout the code: REMARKS CARDS PRECEDE THE REFERENCED SYMBOL DEFINITION. SEE SYMBOL TABLE TO FIND APPROPRIATE PAGE NUMBERS. With these statements the programmers make reference to two of the major forms of the code: the cards that make up the individual lines and the transformed and paginated text printed on 11 × 15-inch paper. These remarks address future readers of the code and provide them with an introduction or preface to the text that will follow. Remarks cards and the remarks columns of the cards are outside of the code — they are assembled and printed but never executed, never transformed into wiring diagrams — but still contained within the text of the code. The reference or pointer to a specialized index or table of contents points to the way in which the programmers understand their code to be a printed text. The addition of new cards, even those making use of six-column card numbers, would alter page numbers, thus there are references to abstract pages rather than specific page numbers.

If we are to understand references to the appearance of the code as abstract, we should read the signs of authorship as even more obscure. Determining the authorship of any collaboratively edited text is difficult. Because of the sparseness of the syntax and diction available and the heavily edited and revised nature of large projects such as the AGC, code remains especially impenetrable to determining authorship. The authorship of code might best be theorized in terms of a function or collaborative group. The code was written and edited collaboratively, but that did not lend much coherency to the organization and form of the code. Fred Martin explains: "[We] had no standards. We had no programming standards. Each group or each little entity would have a style. I think when we got into project management, I did try, to some extent, to get some standardization. But it was hard. I think people used different expressions for constants in their programs." At several locations within the code there are self-referential comments referring to "the authors" as originators for the commentary and code. I will thus follow their lead and unless there are specific markers, we refer to the AGC code as authored by "the programmers" throughout our interpretation and analysis.

The following code segment tells us that Margaret Hamilton is the author of this particular program, which itself appears to be four different programs:

```
# PROGRAM NAME: PREREAD, READACCS, SERVICER, AVERAGE G.
# MOD NO. 00 BY M. HAMILTON     DEC. 12, 1966
#
# FUNCTIONAL DESCRIPTION
```

The code displayed here in these lines has been transformed from the paginated number lines into a format that corresponds to contemporary coding practices. These are lines of code as they appear in the Github repository for the Apollo 11 AGC code. Instead of numbered remarks cards, commentary appears in lines or sections of lines beginning with the # character. This conforms to coding norms in a number of more contemporary programming

languages. The text or other characters following the # are not interpreted. Each of the above lines would have been a separate remarks card. The first line glosses the purpose of this particular section or program within the body of the AGC code. We know that this is the first version or modification of the code, but the numbering scheme here includes double-digit zeros, a variation of the modification scheme used in other sections and subroutines.

The following lines provide another example of what was a set of remarks cards, a set of cards introducing code with a more complex revision history:

```
# SUBROUTINE NAME: TFFCONIC             DATE:  01.29.67
# MOD NO: 0                             LOG SECTION: TIME OF FREE FALL
# MOD BY: RR BAIRNSFATHER
# MOD NO: 1 MOD BY: RR BAIRNSFATHER DATE: 11 APR 67
# MOD NO: 2 MOD BY: RR BAIRNSFATHER DATE: 21 NOV 67 ADD MOON MU.
# MOD NO: 3 MOD BY: RR BAIRNSFATHER DATE: 21 MAR 68 ACCEPT DIFFERENT EARTH/ MOON SCALES
```

In the above, we see several modifications or revisions of the code. Each of these “mods” is numbered in sequential order, beginning with 0 for the first modification (unlike the double-digit mod in the previous example) and incremented by one for each major revision. What constitutes enough change to introduce a new “mod” is not exactly clear from the code or the manuals. How much of the code should change for the mod counter to be incremented? Any modification of the code at all? Major changes?

What we would now call the programming environment for writing and editing code was entirely paper-based. Because of the nature of the input devices, the punch card system, and the use of other programs to pass through and assemble the code, it needed to undergo numerous runs through a transformation that moved and organized stacks of individual cards into paginated, ordered form. The printed pages that provide the historical record of the AGC code demonstrate the extent to which the programmers needed to keep these different forms active in their imagination of the code at all times. The AGC code provides a palimpsestic record of this process; it bears the traces of its composition and revision. These lines of code offer up a snapshot, a frozen image of a collaboratively edited and dynamically changing text.
Tracing Memory and Executing Code

The Apollo Guidance Computer code is complex and frequently difficult to understand. It is hard to follow for several reasons. The limited number of instructions and the lack of abstracted or higher-level libraries providing commonly used subroutines means that the code needed to be as compact and minimal as possible. When reading the code, we need to trace the “line” of execution and follow instructions and data through numerous obscure and abstruse subroutines. In following these instructions, we need to keep in mind the current state of the computer and memory and in particular the present state of a special location or register known as the accumulator. The accumulator was used by the programmers to store the current value of the last arithmetic or logical operation.

The small sections of code shown in this section were compiled, executed, and inspected using two tools from the open-source Virtual AGC environment: yaYUL, the code compiler that generates “core-ropes” objects and yaAGC, the AGC emulator and debugger that executes compiled core-ropes. These tools are provided as part of the fantastic resource that is the open-source Virtual AGC environment. The code for these two tools (they were written in C and can be compiled on several different platforms) can be found with the rest of the environment at: https://github.com/virtualagc/.

The following lines of a fragment of a Basic program for the AGC demonstrates how one would write a program to add together two simple decimal numbers and save the result to a section of erasable memory:

```
BLOCK  2
SUM   EQUALS  10
A   EQUALS  0
CA  VALUE1
AD  VALUE2
TS  SUM
TC  EXIT

VALUE1   DEC 5
VALUE2   DEC 7
```
The first line contains an instruction to tell the computer where to store the code, which block of memory to use. The next two lines assign names to specific memory locations. The name \texttt{A} is shorthand for the accumulator. The memory location name \texttt{SUM} is used, in this code fragment, as the storage location using a 10-bit memory address, to which we will transfer the output from the accumulator. The two numbers to be added are stored as variables. These variables, \texttt{VALUE1} and \texttt{VALUE2}, are defined as a particular type of number, decimals. The \texttt{DEC} instruction tells the compiler that the variable name appearing to the left will contain a decimal and assigns to this variable the value on the right. Decimals, with either single, double, or triple precision, are one datatype used by the Basic/Yul programming language, and others include tables and vectors. Within the debugger provided by the AGC emulator, we can access a list of these variables with the “info variables” command:

File \texttt{sum.agc}:
\begin{verbatim}
var VALUE1;
var VALUE2;
\end{verbatim}

To display the stored or current values of these variables, we can use the print command:

\begin{verbatim}
print/d VALUE1
$1 = +5
\end{verbatim}

The +5 indicates that \texttt{VALUE1} was stored as a decimal value with a positive value. To add these two variables, we first “Clear and Add” (CA) the value of \texttt{VALUE1} to the accumulator. Executing this code instruction by instruction, we watch the value of the accumulator change:

\begin{verbatim}
print/d A
$1 = +0
next
print/d A
$1 = +5
\end{verbatim}
Once the accumulator contains the value of VALUE1, we can Add (AD) the value of VALUE2 to the accumulator:

\[
\begin{align*}
\text{next} \\
\text{print/d A} \\
$1 = +12 \\
\end{align*}
\]

With the value we want available in the accumulator register, we can Transfer to Storage (TS) this value to the storage location SUM and then Transfer Control (TC) to a subroutine named EXIT that performs no function:

\[
\begin{align*}
\text{TS SUM} \\
\text{TC EXIT} \\
\end{align*}
\]

We can print the contents of the memory location referenced as SUM and find the correct result:

\[
\begin{align*}
\text{print/d SUM} \\
$1 = +12 \\
\end{align*}
\]

Building on the above set of instructions, we can see how we might begin to implement the Euclidean distance metric mentioned in the previous chapter using a simple set of Basic primitives. In the extended list, the language provides an instruction to calculate the square of a number store in the accumulator called SQUARE. Using a temporary storage location, we can subtract two numbers and then square the result. This code fragment adds the instruction to Subtract (SU), Square (SQUARE), and exchanges the output of the L register used to store the result by the SQUARE instruction with another erasable memory location (OUTPUT):
BLOCK    2
OUTP    EQUALS   10
A    EQUALS   0
TEMP    EQUALS   11
CA   VALUE2
TS   TEMP
CA   VALUE1
EXTEND
SU   TEMP
EXTEND
SQUARE
LXCH OUTP
TC   EXIT
VALUE1   DEC   8
VALUE2   DEC   64
EXIT