Generating π with a Turing Machine

Morgan Saville

July 2019

1 Introduction

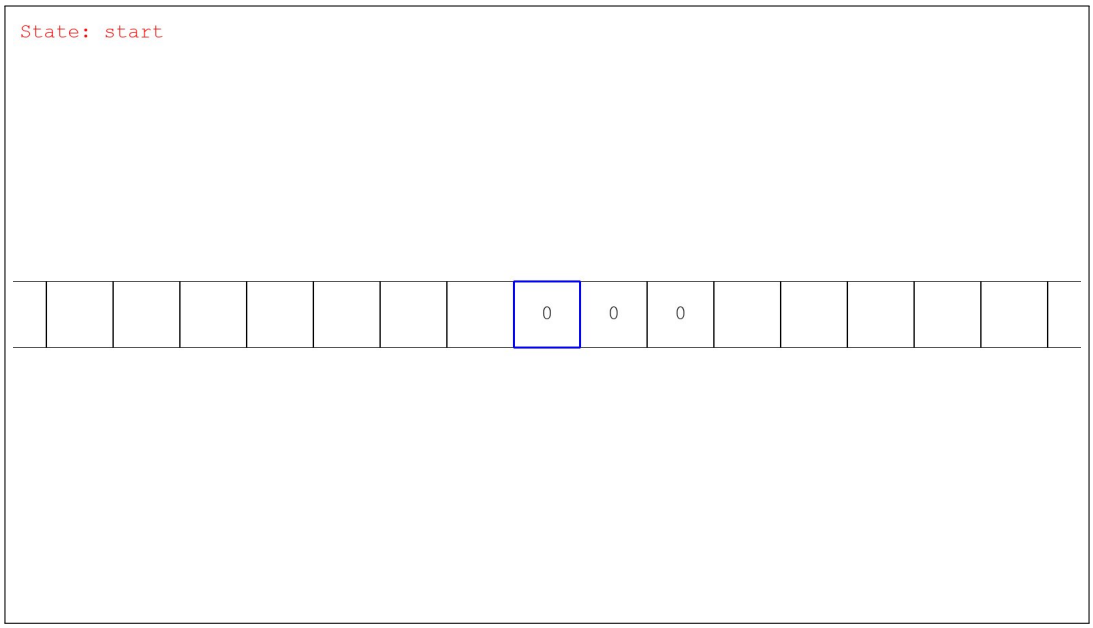
A Turing machine is a theoretical machine consisting of an infinite tape made out of cells which can contain symbols. The Turing machine also has a read/write head which points at a cell on the tape (which we will call the "current" cell), and is capable of reading what symbol is in the cell, and writing a new symbol into the cell. The machine also keeps track of its internal "state". A Turing machine is programmed with rules which tell the machine how to react to reading a certain symbol in the current cell. A rule states that if the machine is in a specific state, and the current cell contains a specific symbol, then the machine should write a new symbol to the cell (which may or may not differ from the symbol which was already there), move the read/write head one cell to the left or right, and then change the machine's state to a new state (which may or may not differ from the old state). We use function notation,

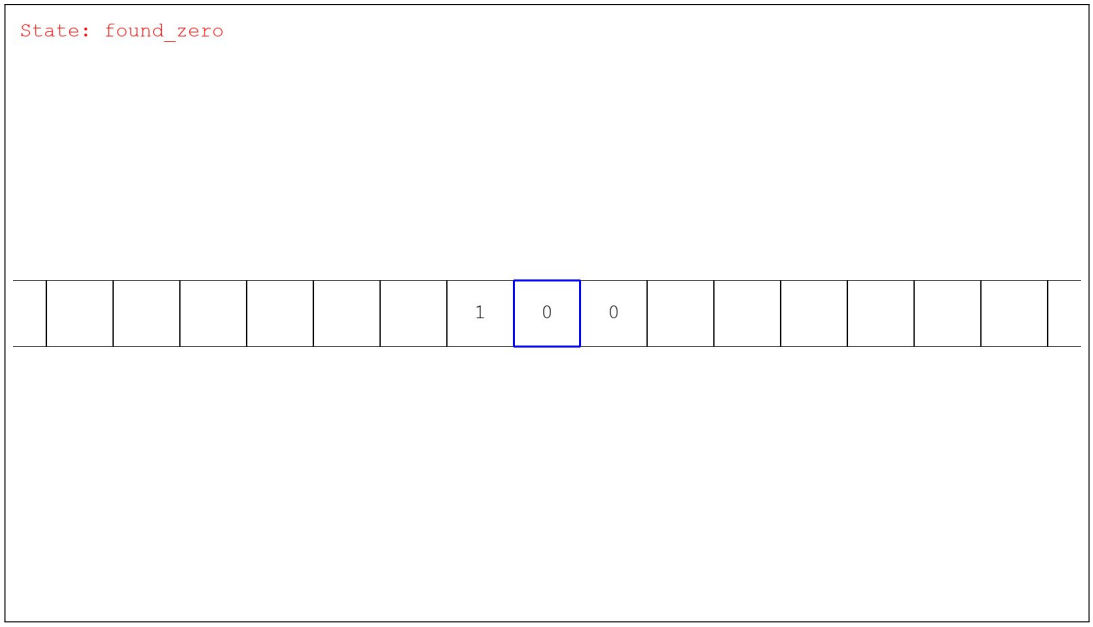
$$\Delta(state_1, symbol_1) = (symbol_2, direction, state_2) \tag{1}$$

to represent these rules, as Δ typically denotes a change. For example,

$$\Delta(start, 0) = (1, \rightarrow, found\_zero) \tag{2}$$

would represent a rule which says that if the machine is in a state called "**start**", and sees the symbol "0" in the current cell, it should replace the symbol with a "1", move the read/write head to the right, and then change to a state called "**found_zero**". The two screenshots represent one possible state of the tape before and after (respectively) applying this rule. The blue box represents the current cell.





A Turing machine program is a finite list of these rules (called a ruleset). The "input" to the program is whatever symbols are initially on the cells of the tape when the program starts, and the "output" of the program is whatever symbols are on the cells of the tape once the program has terminated. Termination occurs when there are no rules in the ruleset which apply to the current situation.

The aim of this paper is to create a Turing machine program which produces an approximation of the circle constant π. This approximation should be arbitrarily accurate, and the accuracy will depend on the input to the program. Further details of the algorithm we will implement are given in the next section.

Furthermore, we will assume that whenever a Turing machine program is executed, the machine initially begins in a state called "**start**".

2 Generation Algorithm

In order to generate approximations of π, we will program our Turing machine to calculate partial sums of the Leibniz formula for π. This is the following:

$$\frac{\pi}{4} = \frac{1}{1} - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \dots \tag{3}$$

For ease of computation, we will rearrange the sum to isolate π.

$$\pi = \frac{4}{1} - \frac{4}{3} + \frac{4}{5} - \frac{4}{7} + \frac{4}{9} - \dots \tag{4}$$

As you can see, the terms of this sum follow a simple pattern. The numerator of each is 4, and the denominators start at 1 and increment by 2 each time, and the terms alternate between being added and subtracted from the total.

This sum is a converging series, meaning that the more terms of the series you consider, the closer the result will get to π, and you can get as close to π as you like just by considering more and more terms.

Our Turing machine will take the denominator of each term in this series in turn, as well as whether it needs to be adding or subtracting. It will then divide 4 by this denominator, to create a decimal expansion to n decimal places, where n is the input to the program. It will then check whether this decimal expansion is zero. If not, it will add (or subtract as appropriate) this expansion to the total (which is also to n decimal places), and then repeat for the next term. If the expansion is indeed zero, stop the program, as the approximation can no longer get any more accurate. Please note that this does not result in the final approximation being accurate as π to n decimal places, but instead just uses n decimal places for the intermediate terms. It still holds that as n gets bigger, the approximation will be closer to π, and in the limiting case where n is arbitrarily big, the approximation is arbitrarily good.

Below is a table showing the steps of this algorithm for $n = 3$.

Denominator	Sign	Decimal Expansion	Total
1	+	4.000	4.000
3	-	1.333	2.667
5	+	0.800	3.467
7	-	0.571	2.896
9	+	0.444	3.340
11	-	0.364	2.976
13	+	0.308	3.284
15	-	0.263	3.017
⋮	⋮	⋮	⋮
7999	-	0.001	3.154
8001	+	0.000	3.154

This series converges so slowly that for any given n, the above table would have $4 \times 10^n + 1$ rows. However, the aim of completing any task with a Turing machine is never speed. All algorithms can be implemented faster on real modern hardware than they can on a Turing machine (especially when the Turing machine is simulated in Python). The aim is often instead to demonstrate that a task is possible, and to experience the wonder created by completing a complex task using nothing but simple rules. As such, I do not believe that the slow convergence of the series is a major issue.

3 Simulation and Creation of the Program

Now that we have discussed the algorithm we will use to generate approximations of π, we can talk about how we will test our program as we write it. I have written my own Turing machine simulator in Python, the code for which can be found at https://github.com/ThePythonist/TuringMachinePiGenerator. This is also the simulator I use to generate the screenshots of the tape used in this paper. This simulator expects the ruleset for the program to be written in a file (we will use a file named "*pi.tur*""). The file will contain each rule on a separate line in Δ function notation. We will create a Python script called *generate.py* which will generate the ruleset file.

First we must create some helpful elements within *generate.py*. The first one we will create is a function called `write`

```
def write(state, read, replace, direction, newState):
    with open("pi.tur", "a", encoding="utf-8") as pi:
        arrow = "\u2190" if direction == "<" else "\u2192"
        pi.write("\u0394(%s,%s) = (%s,%s,%s)\n" % (state, read, replace, arrow, newState))
```

This function allows easier writing of rules to the ruleset. This is particularly necessary because it eliminates the need for repeatedly using the unwieldy Unicode characters, `"\u 2190"` ($\leftarrow$), `"\u 2192"` ($\rightarrow$) and `"\u 0394"` (Δ). In order to write the rule shown in equation (2) to the ruleset file "*pi.tur*", we would call the `write` function as follows:

```
write("start", "0", "1", ">", "found_zero")
```

Note that the `read` and `replace` parameters are always strings, even when the symbols they represent are numerical digits. This is because the Turing machine has no concept of what numbers are mathematically, but instead it knows simply what the symbols look like. We will also write a small amount of code simply to make sure that the file "*pi.tur*" exists and is empty when the generation of the ruleset begins.

```
with open("pi.tur", "w") as pi:
    pi.write("")
```

Finally we will create a list of strings representing the numerical digits 0 to 9 in order. While this seems trivial, we will need to iterate over this list a very large number of times, and so it is helpful to have it stored in a variable. Our "*generate.py*" file now looks like this:

```
def write(state, read, replace, direction, newState):
    with open("pi.tur", "a", encoding="utf-8") as pi:
        arrow = "\u2190" if direction == "<" else "\u2192"
        pi.write("\u0394(%s,%s) = (%s,%s,%s)\n" % (state, read, replace, arrow, newState))

with open("pi.tur", "w") as pi:
```

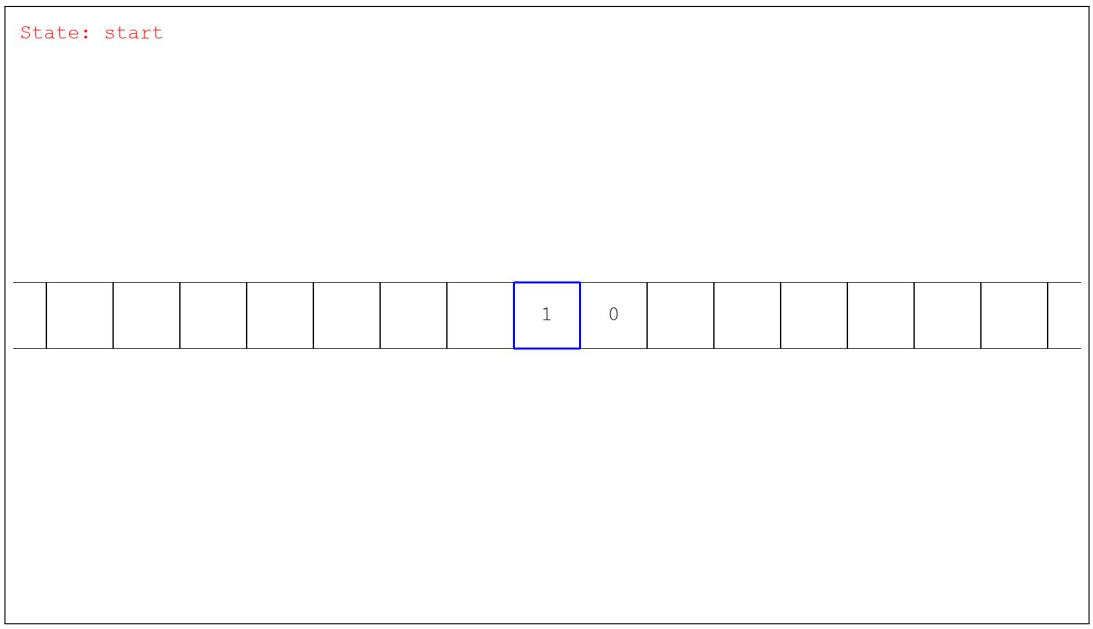
```
    pi.write("")

digits = list(str(i) for i in range(10))
```

Now that we have done this little bit of housekeeping, we can go on to write the code that creates the ruleset. In the following sections, I will refer to the Turing machine and the program we are creating for it interchangeably as "we"/"us"", as well as anthropomorphising the processes carried out by the program. This is to help with following what the program is doing, as it takes a large number of instructions to complete very basic tasks. We will now be adding code to "*generate.py*". If you are following along, do not forget to re-run this script every time we modify it, so that it can regenerate the Turing machine program.

4 Setting Up the Calculation Environment

4.1 Counter Decrementation

When we first start the program, the tape will be blank except for the input number n. Note that if n is multiple digits long, these digits will be written one-digit-per-cell with the most significant digit in the left-most cell, and each additional digit in the following adjacent cells to the right. The read/write head will begin over the left-most digit, and the state will be "**start**". An example of this with $n = 10$ is shown below.



Note that all subsequent screenshots will also use $n = 10$ unless specified otherwise. The first task for the program is to designate a space for the running total, which should be $n + 2$ cells wide (including the space for the digit before the decimal place, and the decimal place itself). We will do this by placing a marker in the cell adjacent to the number on the tape on the right. We will then decrement the number by 1, and shift the marker one cell to the right, repeating these two steps until the counter is zero. Once the counter is indeed zero, we will place another marker two cells to the left of the right-most digit of the counter (dropping the decimal place in the appropriate spot on the way). After this, we should have cell containing a "0", a decimal point, and then n more cells containing a "0" between the two markers.

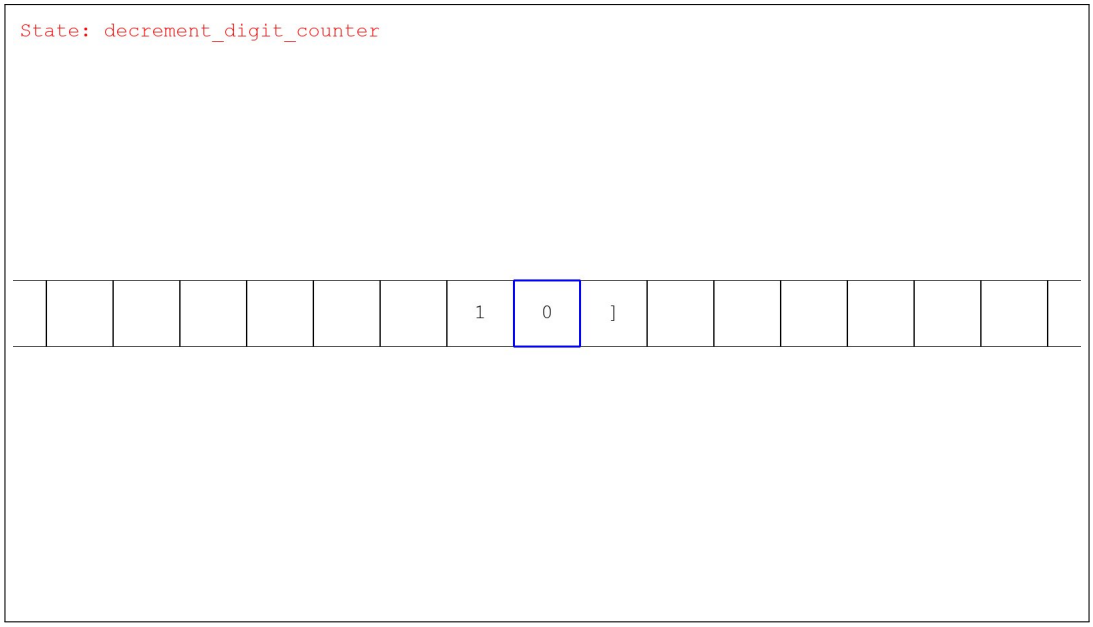
First we will drop the marker after the number on the tape. To do this we must move the read/write head to the end of the number. This is achieved by writing rules which say that if we are in the "**start**" state, and the current cell contains a digit (0-9), we will write that same digit into the cell so as not to change it, move the read/write head to the right, and stay in the "**start**" state. We therefore append the following Python code to "*generate.py*"

```
for digit in digits:
    write("start", digit, digit, ">", "start")
```

However, if we are in the "**start**" state and we encounter an empty cell, we replace it with a marker (we will use "]" as the end marker), move back to the left, and move into a new state. We will call this state "**decrement_digit_counter**".

```
write("start", "", "]", "<", "decrement_digit_counter")
```

At this point once the program has terminated it looks like this:



We now need to find a way to decrement the counter. If the digit in the current cell is non-zero, we simply replace it with the digit which is one less than it, and then move to the right, and change to a state called "**nudge_end_marker**".

```
for i, digit in enumerate(digits[1:]):
    write("decrement_digit_counter", digit, digits[i], ">", "nudge_end_marker")
```

However, if the current digit is zero, then we need to replace it with a "9", and decrement the next digit on the left. In this way, the process is self-similar, and so we can remain in the same state and rely on the logic we have just implemented.

```
write("decrement_digit_counter", "0", "9", "<", "decrement_digit_counter")
```

Now, the decrementation process is complete apart from the fact that if the counter is already zero, then it will roll over to all nines, the read/write head will move to the left onto a blank cell and the process will terminate. We will therefore implement a rule which will detect this, and then place the start marker. First we detect the empty space, and change to a state called "**digit_counter_zero**"

```
write("decrement_digit_counter", "", "", ">", "digit_counter_zero")
```

Once in this state, we find the right-most digit of the number (because if n is more than one digit long, it could have decremented to a long string of zeroes, now rolled over to a long string of nines, and so we might not already be at the right-most digit). We can also erase the number as we go, as we no longer need to use it. Since the only possible digits in the number at this point are nines, this is the only input we need to consider. If we see a 9, replace it with an empty space, move to the right, and stay in the same state. We know we have reached the end of the number if we see a space, in which case we leave it as a space, move to the left, and move to a state named "**drop_decimal_place**" (because we can fill in the appropriate blank cell with a decimal place on the way to dropping the start marker.

```
write("digit_counter_zero", "9", "", ">", "digit_counter_zero")
write("digit_counter_zero", "", "", "<", "drop_decimal_place")
```

As it turns out, once we have moved left and into the state "**drop_decimal_place**", we are in the perfect position to drop the decimal place in the blank cell we created while deleting the counter. We will then do so and then continue moving left and into a state named "**drop_start_marker**"

```
write("drop_decimal_place", "", ".", "<", "drop_start_marker")
```

Now we will introduce a shorthand which we will use for the remainder of the paper. What we want now is to move one additional cell to the left, and then drop the marker there. However, we cannot simply tell the Turing machine to move left when it sees a blank cell in this state, and then remain in the same state, because then it will do so forever.

To rectify this, we will tell it to transition into a new state with a similar name, but with an underscore at the end to signify that this new state serves a very similar purpose, but it is reached once a space has been detected whilst in the previous state. This is important to note, as the state names can become very confusing once there are a lot of states. Once in this new state, when we detect an empty space, we will drop the start marker (for which we will use "["), and then move to the right, transitioning into a state named "**write_zeroes**"

```
write("drop_start_marker", "", "", "<", "drop_start_marker_")
write("drop_start_marker_", "", "[", ">", "write_zeroes")
```

4.2 Nudging the End Marker

The protocol we have just implemented handles what happens when the counter is empty, however, we still must implement the protocol for if the counter is decremented while still being non-zero. We left off with the Turing machine in state "**nudge_end_marker**" with the read/write head somewhere in the middle of the counter. We must therefore make the read/write head move to the right until it finds the end marker. In the meantime it may encounter any digit (0-9) or an empty space, and so we must consider these two options. We can do this by iterating over the list of digits concatenated with a list containing only an empty string as follows:

```
for digit in digits+[""]:
    write("nudge_end_marker", digit, digit, ">", "nudge_end_marker")
```

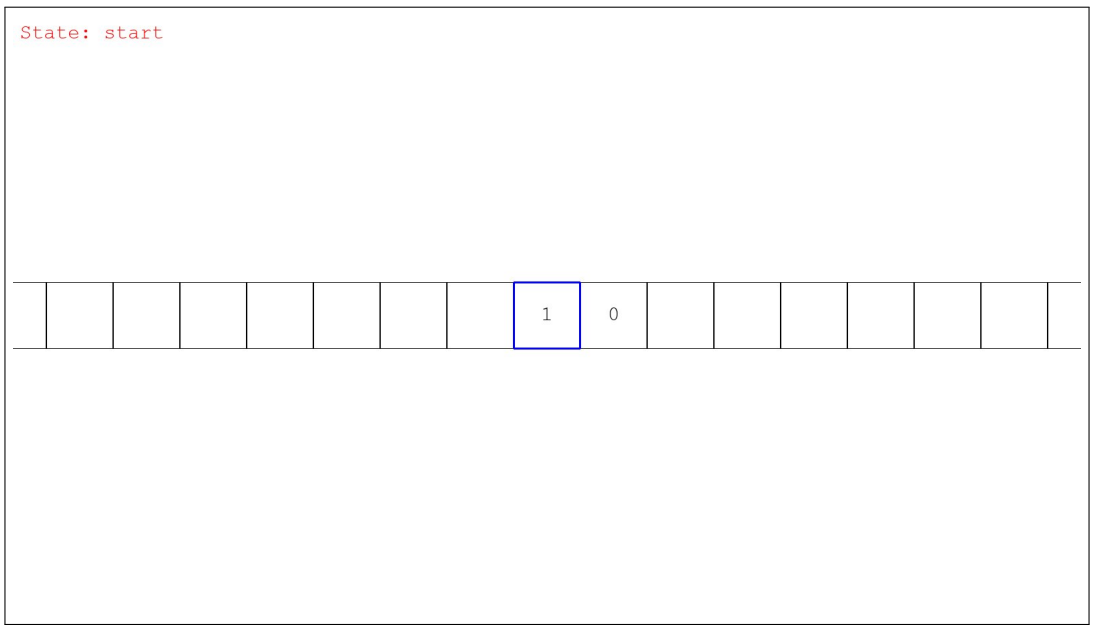
Once we have found the end marker, we need to replace it with an empty space, move again to the right, and transition to a state named "**replace_end_marker**". Once in this state, we replace the empty space with an end marker, and then move back to the left, transitioning into a new state named "**nudged_end_marker**", so that we may make our way back to the counter to decrement it again.

```
write("nudge_end_marker", "]", "", ">", "replace_end_marker")
write("replace_end_marker", "", "]", "<", "nudged_end_marker")
```

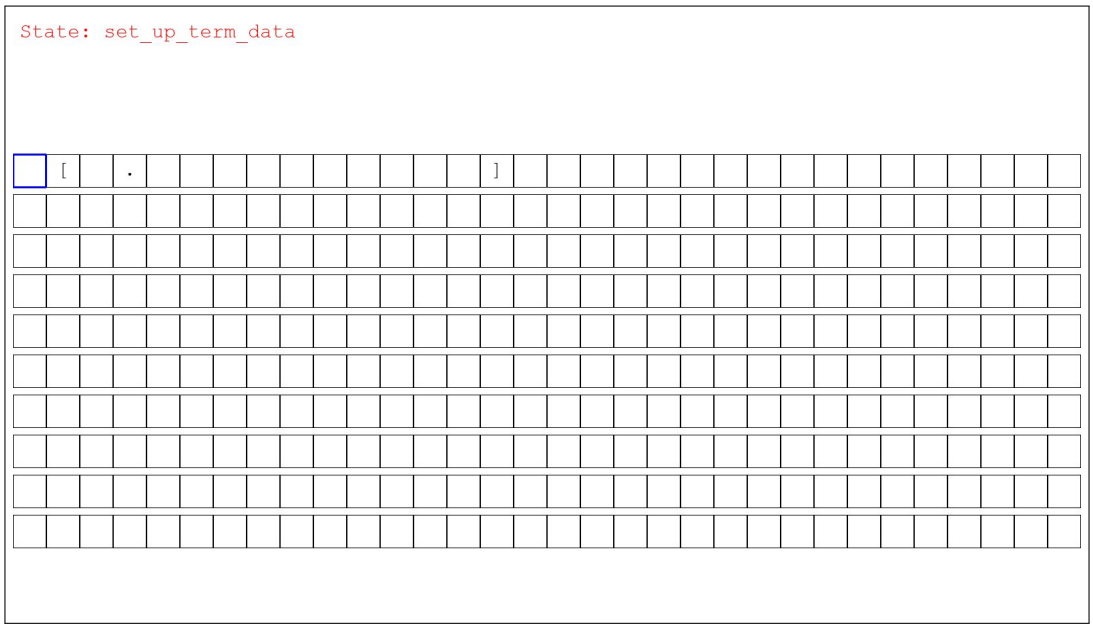
We now move back to the counter. We can tell when we have arrived because we will encounter a digit rather than a space. When this occurs, we move to the right again and into a state named "**found_counter**". We do this so that once in "**found_counter**", we can move left to the right-most bit of the counter and transition back to the state "**decrement_digit_counter**", so that the process will repeat.

```
write("nudged_end_marker", "", "", "<", "nudged_end_marker")
for digit in digits:
    write("nudged_end_marker", digit, digit, ">", "found_counter")
write("found_counter", "", "", "<", "decrement_digit_counter")
```

If we run the program now with the following input:



it will terminate in this state:

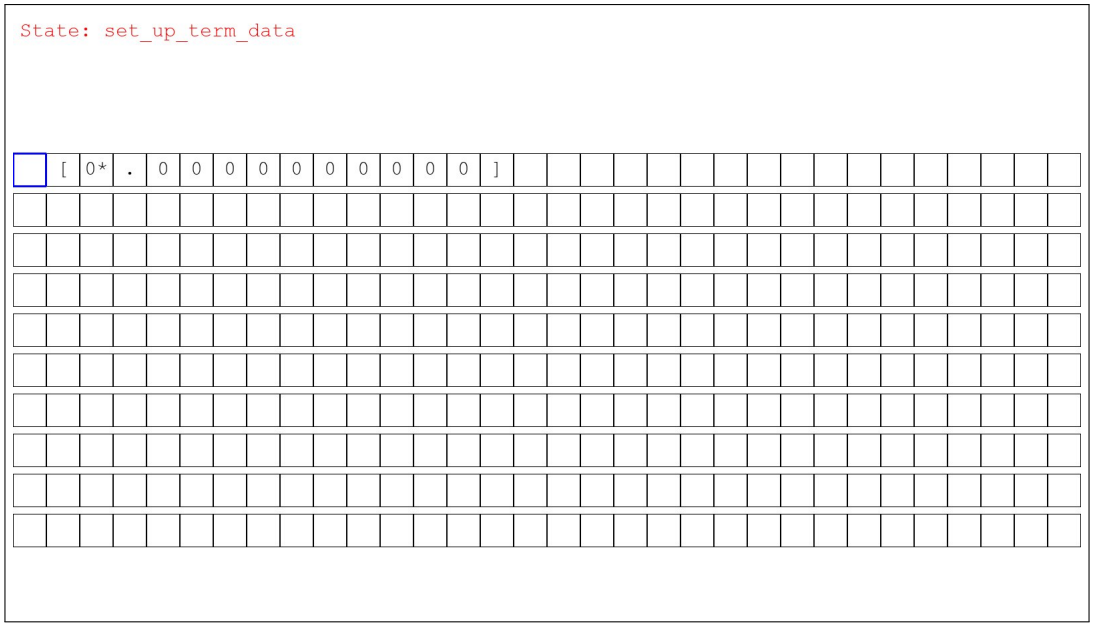


In the above screenshot, the cells shown are in reading order (left to right, then top to bottom). That is to say that the left-most cell in the second row is the cell directly to the right of the right-most cell in the first row and so on. This view is used to allow us to see more of the tape at once. Again, the blue square shows the position of the read/write head. As you can see the program we have written has taken our input number n, and has generated a cordoned-off area whose length is $n + 2$, with a decimal place after the first space. It terminates in the state "**write_zeroes**" and the read/write head is one cell to the right of the start marker. We now simply need to replace all of the empty cells in the space with more zeroes, and then move to the left-hand side of the area.

We need to write a special zero in the space before the decimal place for reasons which will become clear later. For now, just note that the first 0 will actually be a "0*" symbol (zero with an asterisk after it). It is also important to note that the asterisk does not appear in a separate cell. Instead it is simply one cell with a single "0*" symbol in it. After writing this, we will move into another state named "**write_other_zeroes**". Once in this state, if we find a decimal place, we leave it be and simply move to the right, and if we find a blank space, we replace it with a regular 0 and move to the right while staying in the same state. If we find the end marker, we then move to the left and into a state named "**wrote_zeroes**". Once in this state, we move back to the left, past zeroes and decimal places, until we find the start marker, at which point we move to the left again and transition into a state named "**set_up_term_data**".

```
write("write_zeroes", "", "0*", ">", "write_other_zeroes")
write("write_other_zeroes", "", "0", ">", "write_other_zeroes")
write("write_other_zeroes", ".", ".", ">", "write_other_zeroes")
write("write_other_zeroes", "]", "]", "<", "wrote_zeroes")

write("wrote_zeroes", "0", "0", "<", "wrote_zeroes")
write("wrote_zeroes", "0*", "0*", "<", "wrote_zeroes")
write("wrote_zeroes", ".", ".", "<", "wrote_zeroes")
write("wrote_zeroes", "[", "[", "<", "set_up_term_data")
```

Now, after running the program, the Turing machine should terminate in state "**set_up_term_data**", with the read/write head one cell to the left of the start marker, as shown below.

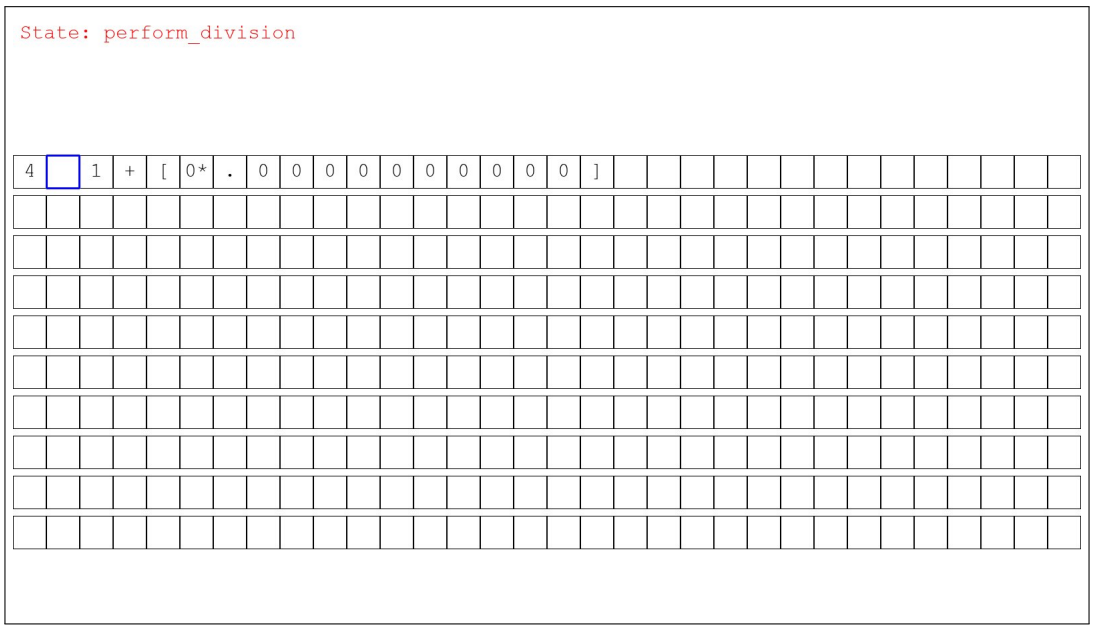


4.3 Setting Up the Term Data

"Term data" is a phrase I like to use to describe the denominator of the next term in the series, and the sign ("+" or "-" if the term needs to be added or subtracted respectively from the total). At the start, the sign of the first term is "+", and the denominator is "1". We will store the sign in the cell to the left of the marked area (which we are currently in) and then we will store the denominator to the immediate left of the sign. Although the numerator is always 4, we still need to write this to the tape. We will write it two cells to the left of the denominator, leaving one empty cell between them, and we will do so in two states named "**write_numerator**" and "**write_numerator_**".

```
write("set_up_term_data", "", "+", "<", "set_up_term_data_")
write("set_up_term_data_", "", "1", "<", "write_numerator")
write("write_numerator", "", "", "<", "write_numerator_")
write("write_numerator_", "", "4", ">", "perform_division")
```

At this point, the tape looks like this:



We are left in the state "**perform_division**" and the read/write head is in the space between the numerator and the denominator. We have now set up the calculation environment, and can move on to implementing the process of long division.

5 Division

5.1 Long Division by Repeated Subtraction

I will now briefly discuss a simplified version of the algorithm we will use to implement long division. Without modification, this specific algorithm only works if the result of the division is less than 10, both operands are natural numbers, and the denominator is non-zero. We start with a digit "0", with some marker to indicate that this is the digit we are currently working with. In this paper we use an asterisk as this marker. From now, the digit with the asterisk will be referred to as the "*working digit*". We then subtract the denominator from the numerator, and check whether the result is negative. If the result is non-negative, we replace the numerator with the result of the subtraction, and then we increment the working digit. If the result is indeed negative, we multiply the numerator by 10, and then replace the working digit with a copy of itself with no asterisk (we now call it a "*cemented digit*") and place a "0*" symbol after it as a new working digit. The process then repeats indefinitely. Below is a table showing the steps of this process when trying to divide 52 by 15:

Numerator	Denominator	Subtraction Result	Negative	Cemented Digits	Working Digit
52	15	37	No		1*
37	15	22	No		2*
22	15	7	No		3*
7	15	-8	Yes	3	0*
70	15	55	No	3	1*
55	15	40	No	3	2*
40	15	25	No	3	3*
25	15	10	No	3	4*
10	15	-5	Yes	34	0*
100	15	85	No	34	1*
85	15	70	No	34	2*
⋮	⋮	⋮	⋮	⋮	⋮

In this case over time, the cemented digits will spell out "34666666..." followed by an arbitrarily large number of sixes. Since we assume that the result is less than 10, there is an implied decimal point after the first digit, which means that the result we get is that $52 \div 15 = 3.4\overline{6}$ which is correct. The algorithm we will implement on the Turing machine is similar but with a few modifications. Specifically, instead of each working digit starting at 0 and incrementing as necessary, our working digits will be each successive digit of the running total in the cordoned off area. This is why the digit before the decimal place is marked with an asterisk before the division begins. One side effect of this is that the working digits can start with a value greater than zero, and so may roll over to the previous digit, so we will have to implement a procedure for handling this. Luckily it is trivial to prove that the running total will never reach a value of 10 or higher, so we do not need to worry about the first digit rolling over and having to expand the marked area. We must also ensure that instead of incrementing the working digit, we decrement it if the sign of the current term is negative. Another modification we must make to the algorithm is a termination case – this is to say that we do not want the division process to continue forever. Luckily we have the end marker "]" to tell us when we have reached the last digit of the division and that we should stop.

One more thing to bear in mind as we implement the division process is that we must include some way of knowing whether or not the result of the division was zero, so that we can stop adding terms. The way we will do this is by placing a marker to the right of the marked area if the working digit is incremented/decremented. Therefore once the division is over, if this marker is not there, then the result of the division was zero, as no working digits were changed.

5.2 Subtraction

The first step is to implement subtraction. To prepare for this we will travel to the right through spaces and digits until we see a sign, at which point we will move back to the left into a state named "**take_last_denom_digit**". From there the aim is to take the last digit of both the denominator and the numerator, and subtract the former from the latter, leaving the result in a cell on the left of the numerator.

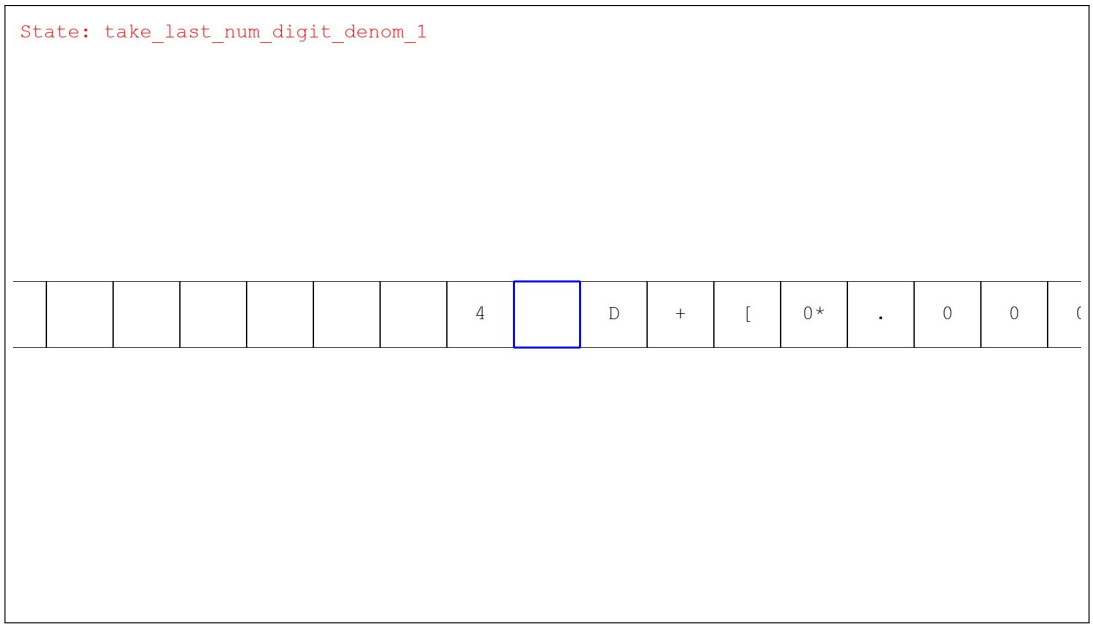
```
for digit in digits + [""]:
    write("perform_division", digit, digit, ">", "perform_division")
write("perform_division", "+", "+", "<", "take_last_denom_digit")
write("perform_division", "-", "-", "<", "take_last_denom_digit")
```

Here is where the state naming can get quite complicated. We are going to perform subtraction digit by digit – first taking the right-most digit of the numerator and denominator. As such, we need to be in a different state after taking each different digit, i.e. if we are in state "**take_last_denom_digit**" and we see a "1", then we will replace it with a

"D" marker and move to a state named "**take_last_num_digit_denom_1**" (because we need to take the last digit of the numerator, and the denominator digit we are remembering is 1), but if we saw a "5", we would need to move to state "**take_last_num_digit_denom_5**" and so on. Our rules are therefore generated by the following Python code:

```
for digit in digits:
    write("take_last_denom_digit", digit, "D", "<", "take_last_num_digit_denom_%s" % digit)
```

Now that we have taken the denominator digit, we need to find the last digit of the numerator. At the moment, the tape is in this position:



As you can see, we are already in the space between the numerator and the denominator. However, if the denominator were more than a single digit long, we would still be somewhere in the middle of it. We therefore need to move left, staying in the same state until we encounter a space. There are 10 different states we could be in right now, and each of them need to be able to respond to 10 different digits and a space. We therefore need the following nested `for` loop:

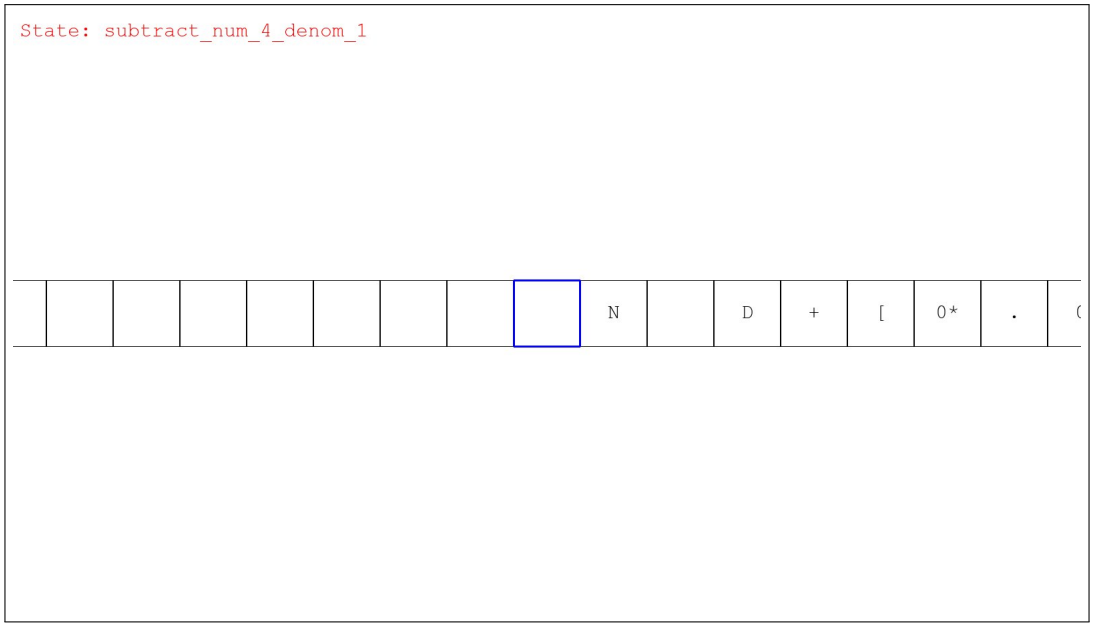
```
for denomDigit in digits:
    for encounteredDigit in digits:
        write("take_last_num_digit_denom_%s" % denomDigit,
              encounteredDigit,
              encounteredDigit,
              "<",
              "take_last_num_digit_denom_%s" % denomDigit)
    write("take_last_num_digit_denom_%s" % denomDigit,
          "",
          "",
          "<",
          "take_last_num_digit_now_denom_%s" % denomDigit)
```

Once the space has been found, we move to the left and transition to a state named "**take_last_num_digit_now_denom_{digit}**" The reason we have added "now" to the state name is to signify that at this point the read/write head is directly on the digit we need to take. We will replace it with an "N" marker and continue to move to the left. The new state name will now have to encapsulate both remembered digits.

```
for denomDigit in digits:
    for numDigit in digits:
        write("take_last_num_digit_now_denom_%s" % denomDigit,
              numDigit,
              "N",
              "<",
              "subtract_num_%s_denom_%s" % (numDigit, denomDigit))
```

Now the state name will be in the form "**subtract_num_{digit}_denom_{digit}**". For example if the remembered

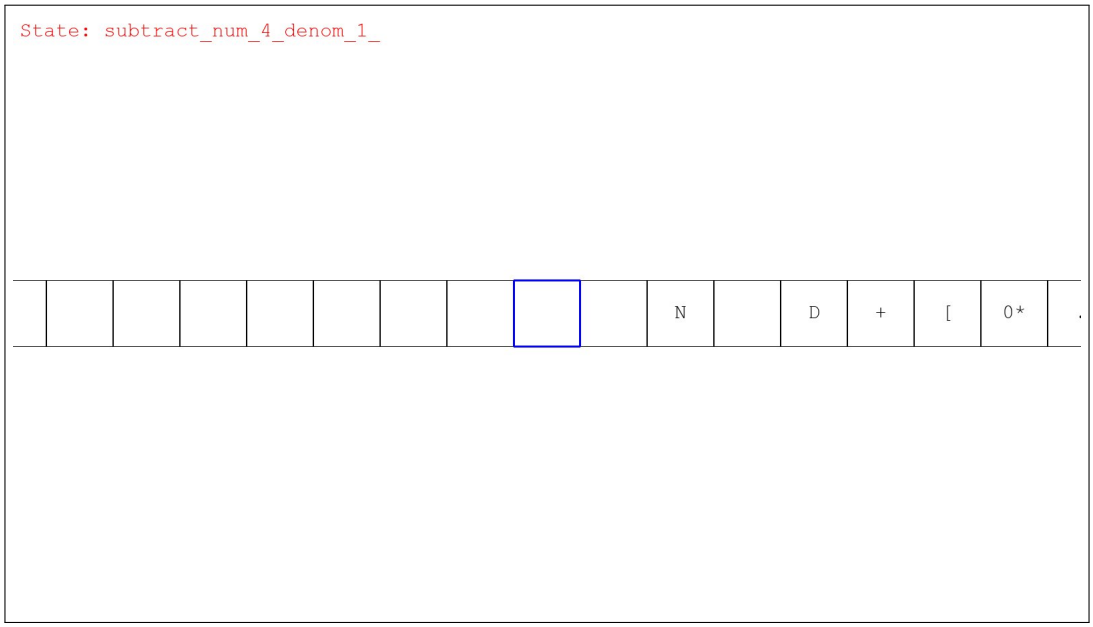
denominator digit was 1 and the numerator digit was 4 (as is the case the first time around) then the state would be "**subtract_num_4_denom_1**" as shown below.



Again, if the numerator was multiple digits long, we may still be in the middle of it, and so we need to move to the left until we find a space. For reasons which will become clear later, the remaining digits of the numerator may still include an "N" marker. Now our state name encapsulates two digits, and in each state we need to react to 10 digits and an "N" marker. This means we need an additional nested `for` loop as follows:

```
for denomDigit in digits:
    for numDigit in digits:
        for encounteredDigit in digits + ["N"]:
            write("subtract_num_%s_denom_%s" % (numDigit, denomDigit),
                  encounteredDigit,
                  encounteredDigit,
                  "<",
                  "subtract_num_%s_denom_%s" % (numDigit, denomDigit))
        write("subtract_num_%s_denom_%s" % (numDigit, denomDigit),
              "",
              "",
              "<",
              "subtract_num_%s_denom_%s_" % (numDigit, denomDigit))
```

Once a space has been found the state name is appended with an underscore to signify that we are still performing the same process but a space has been found. This is so that we can leave a blank cell between the numerator and the subtraction result. The tape now looks like this:



Right now, this is the space in which we want to place the result of the subtraction. If the numerator digit subtract the denominator digit is non-negative, we would write the difference in this cell and move on. Otherwise, we would write the difference plus ten, and write a "C" in the cell to the immediate left to indicate that we carry the -1. For example if the numerator digit was 1 and the denominator digit was 5, the difference is -4, so we would write a 6 in this cell and then a "C" in the one on the left. However, this means that on the subtraction of the next digits, this space might not be empty, and so we need to move to the left until we find either a blank space or a "C". Here there are four cases to consider.

1. If we find a blank space and the difference is non-negative, write the difference to the cell and move to the right into a state named "**subtracted_num_{digit}_denom_{digit}**".
2. If we find a blank space but instead the difference is negative, write the difference plus ten into the cell, and move left into a state named "**drop_carry_marker_num_{digit}_denom_{digit}**".
3. If we find a "C" (meaning we have an extra -1 to consider) and the difference is greater than or equal to 1, then we write the difference subtract 1 into the cell, and move right into a state named "**subtracted_num_{digit}_denom_{digit}**".
4. Finally, if we find a "C" and the difference is less than 1, we write the difference plus nine into the cell, and move left into a state named "**drop_carry_marker_num_{digit}_denom_{digit}**".

Below is an implementation of these rules.

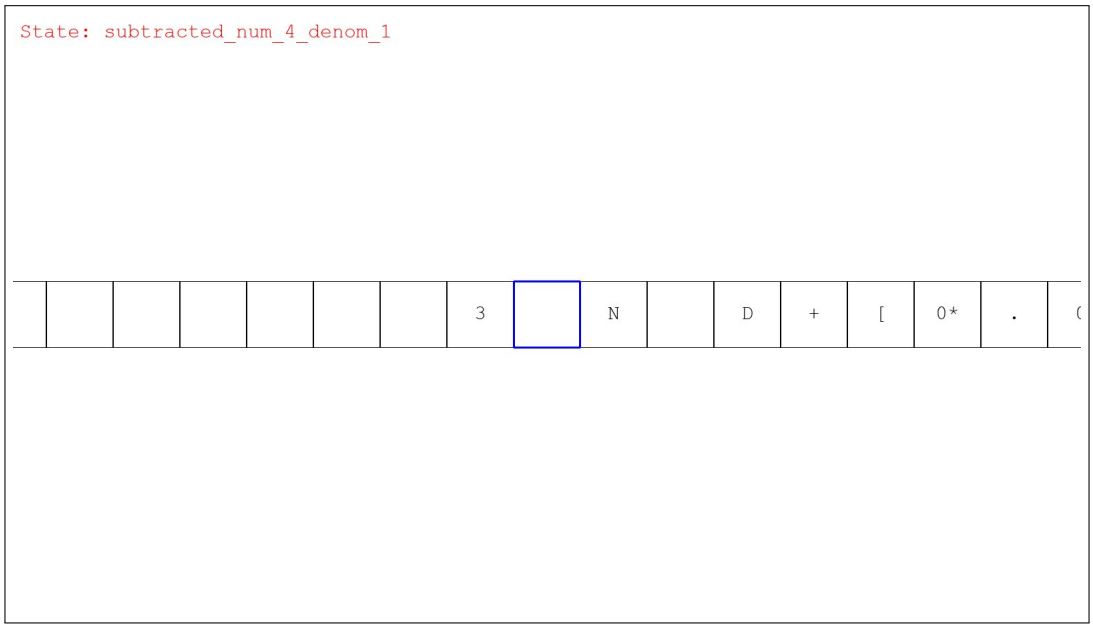
```
for denomDigit in digits:
    for numDigit in digits:
        for encounteredDigit in digits:
            write("subtract_num_%s_denom_%s_" % (numDigit, denomDigit),
                  encounteredDigit,
                  encounteredDigit,
                  "<",
                  "subtract_num_%s_denom_%s_" % (numDigit, denomDigit))

        difference = int(numDigit) - int(denomDigit)
        if difference >= 0:
            write("subtract_num_%s_denom_%s_" % (numDigit, denomDigit),
                  "",
                  str(difference),
                  ">",
                  "subtracted_num_%s_denom_%s" % (numDigit, denomDigit))
        else:
            write("subtract_num_%s_denom_%s_" % (numDigit, denomDigit),
                  "",
                  str(difference+10),
                  "<",
                  "drop_carry_marker_num_%s_denom_%s" % (numDigit, denomDigit))
        if difference >= 1:
            write("subtract_num_%s_denom_%s_" % (numDigit, denomDigit),
                  "C",
                  str(difference-1),
                  ">",
                  "subtracted_num_%s_denom_%s" % (numDigit, denomDigit))
        else:
            write("subtract_num_%s_denom_%s_" % (numDigit, denomDigit),
                  "C",
                  str(difference+9),
                  "<",
                  "drop_carry_marker_num_%s_denom_%s" % (numDigit, denomDigit))
```

If we had to drop a carry marker, we have now ended up in a state named "**drop_carry_marker_num_{digit}_denom_{digit}**". From there, we replace the blank cell with a "C" and move right the state "**subtracted_num_{digit}_denom_{digit}**".

```
for denomDigit in digits:
    for numDigit in digits:
```

```
        write("drop_carry_marker_num_%s_denom_%s" % (numDigit, denomDigit),
              "",
              "C",
              ">",
              "subtracted_num_%s_denom_%s" % (numDigit, denomDigit))
```

Now when the program terminates the tape will look like this:



At this point you might be wondering why the state name still needs to include the digits we've taken from the numerator and the denominator, seeing as how we've already used them to do the subtraction. However, we replaced these digits with an "N" marker for the numerator and a "D" marker for the denominator, and so we need to remember the digits we took so that we can put them back once we've done the subtraction. We needed to use these markers because the Turing machine needs to know which digits it just subtracted from each number, so that on the next iteration it knows where to find the next digits from each number. We now therefore need to move back to the right until we find the "N" marker. This may be through digits or empty spaces. Once we have found the "N" marker, we replace it with the digit we were remembering for the numerator, move one cell to the left, and into a state named "**fetch_num_denom_{digit}**". Note that we no longer need to keep track of the numerator digit. Once in this state, if we see a digit, then we take that digit to be our next numerator number and replace it with the "N" marker, moving to the right and moving into a state named "**find_denom_num_{digit}_denom_{digit}**" (where the first digit is that which we have just taken from the numerator, and the second digit is the one we need to return to the denominator). However, if instead of a digit we find an empty space, it means that the "N" marker was on the left-most digit of the numerator, and so we have already taken every digit from the numerator. Since there may still be more digits to take from the denominator, we nonetheless need to continue the subtraction, taking the new numerator digit to be zero. As such, in this case we move back to the right and into a state named "**find_denom_num_0_denom_{digit}**". Notice that in this case we do not drop another "N" marker. This means that while searching for the "N" marker on the next iteration, there is a chance that the read/write head will instead encounter the "D" marker first. In this case, it would be in the state "**subtracted_num_0_denom_{digit}**", and should replace the "D" marker with the digit being remembered for the denominator, and then move left into the state "**fetch_denom_num_0**" (where now the 0 represents the next numerator digit to subtract, as there are no more digits to take from the numerator itself).

```
for denomDigit in digits:
    for numDigit in digits:
        for encounteredDigit in digits + [""]:
            write("subtracted_num_%s_denom_%s" % (numDigit, denomDigit),
                  encounteredDigit,
                  encounteredDigit,
                  ">",
                  "subtracted_num_%s_denom_%s" % (numDigit, denomDigit))

        write("subtracted_num_%s_denom_%s" % (numDigit, denomDigit),
              "N",
              numDigit,
              "<",
              "fetch_num_denom_%s" % denomDigit)
```

```
    write("subtracted_num_0_denom_%s" % denomDigit,
          "D",
          denomDigit,
          "<",
          "fetch_denom_num_0")

for denomDigit in digits:
    for numDigit in digits:
        write("fetch_num_denom_%s" % denomDigit,
              numDigit,
              "N",
              ">",
              "find_denom_num_%s_denom_%s" % (numDigit, denomDigit))
    write("fetch_num_denom_%s" % denomDigit,
          "",
          "",
          ">",
          "find_denom_num_0_denom_%s" % denomDigit)
```

If a new numerator digit was indeed found, we would now be in the state "**find_denom_num_{digit}_denom_{digit}**". In this case we would need to continue moving to the right until we find the "D" marker (through digits and empty spaces). As with before, when we find the "D" marker, we replace it with the digit we have been remembering for the denominator and move left into the state "**fetch_denom_num_{digit}**" where now the digit is that which we have just picked up from the numerator. Once in this state, if we see a digit, we replace it with the "D" marker, and move to the left, transitioning to a new state named "**fetched_num_{digit}_denom_{digit}**". However, as with before, if we have exhausted all of the digits in the denominator, we may find a blank space where we expect a digit. Again, since the denominator may contain fewer digits than the numerator, we cannot cease subtraction. Instead, we leave the space empty, and move back to the left, but this time into a state we have already used, named "**subtract_num_{digit}_denom_0**". This is the state that allows us to move through the numerator to the result area and continue the calculation, and this is why when implementing this state we accounted for the fact that we may need to move through an "N" marker. As with the numerator, it is therefore possible that the read/write pointer may come to a sign whilst looking for the "D" marker if it is carrying a zero for the denominator digit. In this case, we take the denominator digit to be 0, and move left into the state "**fetched_num_{digit}_denom_0**".

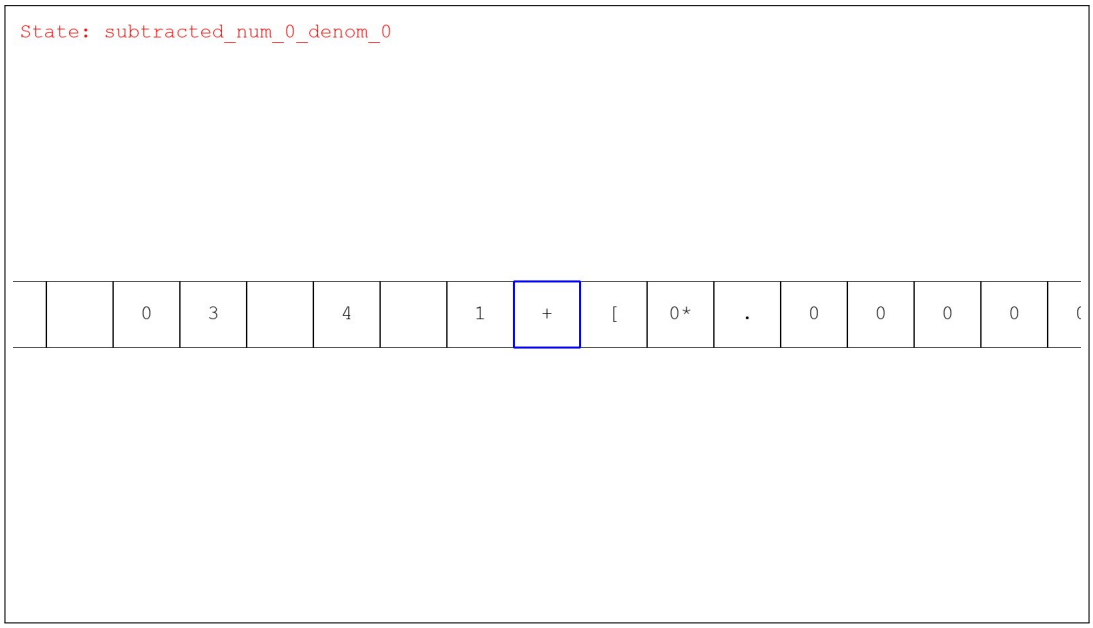
```
for denomDigit in digits:
    for numDigit in digits:
        for encounteredDigit in digits + [""]:
            write("find_denom_num_%s_denom_%s" % (numDigit, denomDigit),
                  encounteredDigit,
                  encounteredDigit,
                  ">",
                  "find_denom_num_%s_denom_%s" % (numDigit, denomDigit))

        write("find_denom_num_%s_denom_%s" % (numDigit, denomDigit),
              "D",
              denomDigit,
              "<",
              "fetch_denom_num_%s" % numDigit)

for numDigit in digits:
    for sign in "+-":
        write("find_denom_num_%s_denom_0" % numDigit,
              sign,
              sign,
              "<",
              "fetched_num_%s_denom_0" % numDigit)

for numDigit in digits:
    for denomDigit in digits:
        write("fetch_denom_num_%s" % numDigit,
              denomDigit,
              "D",
```

```
            "<",
            "fetched_num_%s_denom_%s" % (numDigit, denomDigit))

    write("fetch_denom_num_%s" % numDigit,
          "",
          "",
          "<",
          "subtract_num_%s_denom_0" % numDigit)
```

Now, once we have fetched the denominator digit, we will be somewhere in the middle of the denominator in the state "**fetched_num_{digit}_denom_{digit}**" and we need to move back to the numerator before moving to the "**subtract_num_{digit}_denom_{digit}**" state. This is because that state looks for a space in order to know where the subtraction result should go, but it would encounter a space in between the numerator and the denominator if we moved to that state straight away. Luckiy the solution is simple. When in the state "**fetched_num_{digit}_denom_{digit}**", move to the left through any digit, and then once we encounter a space, move again to the left and transition to the state "**subtract_num_{digit}_denom_{digit}**".

```
for denomDigit in digits:
    for numDigit in digits:
        for encounteredDigit in digits:
            write("fetched_num_%s_denom_%s" % (numDigit, denomDigit),
                  encounteredDigit,
                  encounteredDigit,
                  "<",
                  "fetched_num_%s_denom_%s" % (numDigit, denomDigit))

        write("fetched_num_%s_denom_%s" % (numDigit, denomDigit),
              "",
              "",
              "<",
              "subtract_num_%s_denom_%s" % (numDigit, denomDigit))
```

From now, the process will repeat until the subtraction is complete. Upon termination, the tape should look like this:



While this may seem like a long-winded way to subtract 1 from 4, it is important not to lose sight of exactly what we have just implemented. The Turing machine can now perform any subtraction of any two numbers, no matter how many digits are in either of them. This is, in my opinion, by far the most complicated part of the entire program.

Before continuing to the next step of the program, we should first examine why the program terminates when the subtraction is completed in the first place. This may be a surprising fact, considering that we never explicitly told the program how to check when the subtraction has finished. We know that once we have run out of digits to take from either the numerator or the denominator, we stop using the "N" or "D" markers altogether, and simply take the digit to be zero. We had to take this into account when considering the fact that when searching for an "N" marker, the read/write head

may instead encounter a "D" marker, as the "N" marker no longer exists. However, once we have exhausted the digits from both the numerator *and* the denominator (i.e. the subtraction is complete), when searching for an "N" marker in the "**subtracted_num_0_denom_0**" state, the read/write head will find neither an "N" marker not a "D" marker, as both will have been removed in previous steps. Therefore, the first symbol it will encounter which is neither a digit nor a space will be a sign ("+" or "-"). We have not written rules which dictate how to react to either of these characters in that state, and so the program terminates. This is one of the two reasons why when developing this environment we have decided to do the subtraction to the left of the marked area, rather than the right – such a setup comes with an inbuilt mechanism by which the end of a subtraction operation is detected.

Similarly, doing subtraction on the left of the marked area is useful because we work with the least significant digits of the operands first, and then work our way up to the most significant digits, meaning that when we do the subtraction, we form the less significant digits of the result first. Because of this, it is nice that we form the result from right to left (whereas if we were doing the subtraction to the right of the marked area it would be more natural to create the subtraction result from left to right), as this means that the subtraction result will be written with the most significant digit on the left, and the least significant digit on the right (as we write numbers in everyday life).

5.3 Updating the Working Digit

The next step of the program is to check whether the result of the subtraction was negative or not. The program terminated in the state "**subtracted_num_0_denom_0**" on a sign, and so upon seeing a sign we can move to the left and into a state named "**check_subtraction_result**".

```
write("subtracted_num_0_denom_0", "+", "+", "<", "check_subtraction_result")
write("subtracted_num_0_denom_0", "-", "-", "<", "check_subtraction_result")
```

Once in this state we need to continue going left until either we find a "C" marker (which would show that the result was negative) or two empty cells in a row (which would show that the result was non-negative). To do this, we will write rules such that when we are in the state "**check_subtraction_result**", if we find a digit, then we move to the left and stay in the same state. If we see a "C" marker, we replace it with an empty cell and move to the right in a state named "**subtraction_negative**". However if we find a space, then we move left but into the state "**check_subtraction_result_**". Similarly, if we are in state "**check_subtraction_result_**" and we see a digit, we move back to "**check_subtraction_result**", but if we see another space, then we move left into the state "**subtraction_positive**". We do not need to worry about encountering a "C" marker in the state "**check_subtraction_result_**" because a "C" marker will never come before a space.

```
for digit in digits:
    write("check_subtraction_result", digit, digit, "<", "check_subtraction_result")
write("check_subtraction_result", "C", "", ">", "subtraction_negative")
write("check_subtraction_result", "", "", "<", "check_subtraction_result_")

for digit in digits:
    write("check_subtraction_result_", digit, digit, "<", "check_subtraction_result")
write("check_subtraction_result_", "", "", ">", "subtraction_positive")
```

First, we will handle what to do if the subtraction result is negative. We can now remove the subtraction result from the tape, as it is no longer relevant. We do this by moving to the right, and replacing all digits with empty spaces. When we encounter an empty space, we continue to move left onto the numerator and into a state named "**multiply_num_by_ten**", as we now need to multiply the numerator by ten.

```
for digit in digits:
    write("subtraction_negative", digit, "", ">", "subtraction_negative")
write("subtraction_negative", "", "", ">", "multiply_num_by_ten")
```

We now need to find the right-hand side of the numerator so that we can add a zero to the end, so we move right through the digits until we find a space. When we do find this space, we will move back left onto the right-most digit of the numerator and transition into a state named "**multiply_num_0**". This is because we now need to push each digit of the numerator one cell to the left, to make room for the new digit. We do so by copying the digit in each cell into the cell on its left. While doing this we will be in a state called "**multiply_num_{digit}**", where the digit in the state name represents which digit to place in the next cell. Since we want to push a "0" into the right-most cell, we start in state "**multiply_num_0**".

```
for digit in digits:
    write("multiply_num_by_ten", digit ,digit, ">", "multiply_num_by_ten")
write("multiply_num_by_ten", "", "", "<", "multiply_num_0")
```

We now say that when in a state whose name is in the form "**multiply_num_{digit}**", and we see another digit, we write the digit in the state name into the cell, and transition into the state "**multiply_num_{digit}**" where the digit is now the digit which was found in the cell, and move to the left. If we find a blank space, replace it with the digit in the state name, and then move right into a state named "**cement_working_digit**"

```
for copyingDigit in digits:
    for encounteredDigit in digits:
        write("multiply_num_%s" % copyingDigit,
              encounteredDigit,
              copyingDigit,
              "<",
              "multiply_num_%s" % encounteredDigit)

    write ("multiply_num_%s" % copyingDigit,
           "",
           copyingDigit,
           ">",
           "cement_working_digit")
```

Now we need to travel to the right, all the way to the working digit, so that we can turn it into a cemented digit, and then turn the digit on its right (if you ignore the decimal place) into the working digit. We therefore need to move right through digits, spaces, the start marker, the signs, and potentially the decimal place. Once we encounter a digit with an asterisk (marking that it is the working digit, we replace it with the same digit but without an asterisk (cementing it) and then move again to the right in the state "**create_working_digit**". If at this point we are looking at a decimal place, leave it be and move one cell further to the right.

```
for symbol in digits + ["", "[", "+", "-", "."]:
    write("cement_working_digit", symbol, symbol, ">", "cement_working_digit")

for digit in digits:
    write("cement_working_digit", digit+"*", digit, ">", "create_working_digit")

write("create_working_digit", ".", ".", ">", "create_working_digit")
```

Now in most cases, the read/write head would be over another digit for us to turn into a working digit. If so, we replace the digit with the same digit followed by an asterisk, and move left into a state named "**return_to_subtraction**". However, we may have reached the end of the marked area and so we would be directly over the end marker, but this does not mean that we have completed the division. We actually need to use this space for a temporary working digit, just to see whether or not we need to round the previous cemented digit up or not. Because of this, we can replace the end marker with a "0*", and move to the left and into the state "**return_to_subtraction**". Now that we have removed the end marker, we may instead be directly on top of the cell which would be immediately to the right of the end marker in the state "**create_working_digit**". This cell may contain a blank space, or a marker to indicate that the division result was not zero (we will use "!" as this marker). If the first of these cases is true, then we need to evaluate whether the temporary working digit is greater than or equal to 5. To do this we move left into a state named "**check_temp_digit**". Once in this state, if we see a digit 4 or below then we replace it with an empty space and move left into a state named "**check_for_addition**". If we see a digit 6 or above, we replace it with the end marker "]", and then move left into a state named "**round**". However if we see a digit 5, then we move left into a state named "**round_and_continue**". This is because when subtracting, if the temporary working digit is 5 and the last cemented digit rounds upwards, then the division had no effect on the running total, but we still want to continue, as the next term is added, and when the temporary working digit is 5 on an addition, the last cemented digit rounds up which *is* a change. If when in the state "**create_working_digit**" we instead encounter a "!" marker, we replace it with a blank space, move to the left and transition into the state "**check_temp_digit_and_continue**". This state is very similar to "**check_temp_digit**" except if we see a digit which is 4 or below, we replace it with the end marker, "]" and move to the left into the state "**division_complete**", whereas if we see 5 or above, we replace it with the end marker and move left into the state "**round_and_continue**"

```
for digit in digits:
    write("create_working_digit", digit, digit+"*", "<", "return_to_subtraction")
```

```
write("create_working_digit", "]", "0*", "<", "return_to_subtraction")
write("create_working_digit", "", "", "<", "check_temp_digit")
write("create_working_digit", "!", "", "<", "check_temp_digit_and_continue")

for lowDigit in digits[:5]:
    write("check_temp_digit", lowDigit, "]", "<", "check_for_addition")
write("check_temp_digit", "5", "]", "<", "round_and_continue")
for highDigit in digits[6:]:
    write("check_temp_digit", highDigit, "]", "<", "round")

for lowDigit in digits[:5]:
    write("check_temp_digit_and_continue", lowDigit, "]", "<", "division_complete")
for highDigit in digits[5:]:
    write("check_temp_digit_and_continue", highDigit, "]", "<", "round_and_continue")
```

We now have several loose ends to tie up:

1. "**return_to_subtraction**": This is the state in which we find ourselves when we have cemented the old working digit, created a new working digit, and now need to move back to the left to perform another subtraction.
2. "**round**": This is the state which is used for when we have finished one entire division and we need to go on to do more divisions if and only if the last term was added rather than subtracted from the running total, and the last digit of the total needs to be rounded up.
3. "**round_and_continue**": This is the state which is used for when we have finished one entire division and we definitely need to go on to do more divisions, but the last digit of the total needs to be rounded up first.
4. "**division_complete**": This is the state which is used when we have finished one entire division and we need to go on to do more divisions, but the last digit of the total does not need to be rounded up first. We will implement this state later.
5. "**check_for_addition**": This state means that the division we have just completed resulted in the temporary working digit rounding down rather than up, and that no other changes were made to the running total. If the current term was being added, then this implies that the current term made no difference to the total, and so the approximation is complete.

First let's tackle "**return_to_subtraction**". This one is quite simple, because all we need to do is move to the left through digits, the decimal place and the start marker (and potentially the working digit, as this state will also be used by the procedure for when the subtraction result was positive) until we find a sign, at which point we move left again and into a state we have already used before - "**take_last_denom_digit**". The logic we implemented earlier will then take care of making sure that the next subtraction takes place.

```
for digit in digits:
    write("return_to_subtraction", digit+"*", digit+"*", "<", "return_to_subtraction")
for symbol in digits + [".", "["]:
    write("return_to_subtraction", symbol, symbol, "<", "return_to_subtraction")
write("return_to_subtraction", "+", "+", "<", "take_last_denom_digit")
write("return_to_subtraction", "-", "-", "<", "take_last_denom_digit")
```

Now let's deal with the states "**round**" and "**round_and_continue**". If the digit we see is between 0 and 8, we replace it with the digit which is one greater than it, and then move left into the state "**check_for_subtraction**". If instead the digit is a 9, then replace it with a 0 and move left, staying in the state "**round**". If we find the decimal point instead, simply move to the left and stay in the state "**round**". Luckily we do not have to consider the possibility of overflow, as the running total will never reach or exceed a value of 10 as previously stated. The same applies for the state "**round_and_continue**" except that we do not need to transition to "**check_for_subtraction**", but can instead transition directly to "**division_complete**" as we know that we must go back and divide more terms.

```
for i, digit in enumerate(digits[:-1]):
    write("round", digit, digits[i+1], "<", "check_for_subtraction")
write("round", "9", "0", "<", "round")
write("round", ".", ".", "<", "round")

for i, digit in enumerate(digits[:-1]):
```

```
    write("round_and_continue", digit, digits[i+1], "<", "division_complete")
write("round_and_continue", "9", "0", "<", "round_and_continue")
write("round_and_continue", ".", ".", "<", "round_and_continue")
```

We will now implement the states "**check_for_addition**" and "**check_for_subtraction**". These are the states that are used when either the sign of the current term being "+" or "-" respectively would imply that the approximation is finished, and otherwise imply that the individual division is finished. In either state we need to move backwards through digits, a decimal place and the starting marker until we reach a sign. If in state "**check_for_addition**" the sign is "+", then we move right into the state "**found**", or else move right into the state "**not_found**" and similar rules apply for "**check_for_subtraction**".

```
for digit in digits + [".", "["]:
    write("check_for_addition", digit, digit, "<", "check_for_addition")
write("check_for_addition", "+", "+", ">", "found")
write("check_for_addition", "-", "-", ">", "not_found")

for digit in digits + [".", "["]:
    write("check_for_subtraction", digit, digit, "<", "check_for_subtraction")
write("check_for_subtraction", "+", "+", ">", "not_found")
write("check_for_subtraction", "-", "-", ">", "found")
```

If in state "**not_found**", we can move once more to the right and into a state named "**reset_working_digit**" which we will implement later. Otherwise if in state "**found**" we continue to move right through the marked area through the start marker, digits and decimal place until we reach the end marker, at which point we can erase it, move left, and transition into a state named "**approximation_complete**", which we will implement later.

```
write("not_found", "[", "[", ">", "reset_working_digit")
for symbol in digits + ["[", "."]:
    write("found", symbol, symbol, ">", "found")
write("found", "]", "", "<", "approximation_complete")
```

We can now go on to deal with what happens when the result of the subtraction is positive. As a reminder, if this is the case we need to replace the numerator with the result of the subtraction, and then either increment or decrement the working digit as appropriate. The first step is to move right once, transitioning into a state named "**remove_leading_zeroes**", then remove the leading zeroes from the subtraction result and replace them with a marker, "#", to signify the start of the number. If we come across a non-zero digit, move right into a state named "**find_num**". However, there is a possibility that the result of the subtraction will be exactly 0, meaning that every digit is a leading zero. If this is the case, we will find ourselves in the "**remove_leading_zeroes**" with the read/write head over a blank space. In such a case, we can simple move to the left again into a new state named "**replace_leading_zero**". Once in this state, replace one "#" marker with a 0 (this is safe to do because if the subtraction result is zero, then there will always be at least two zeroes in the result, and so replacing one of the markers with a zero is guaranteed to leave at least one marker remaining), and then move right into the state "**find_num**"

```
write("subtraction_positive", "", "", ">", "remove_leading_zeroes")
write("remove_leading_zeroes", "0", "#", ">", "remove_leading_zeroes")
for digit in digits[1:]:
    write("remove_leading_zeroes", digit, digit, ">", "find_num")
write("remove_leading_zeroes", "", "", "<", "replace_leading_zero")
write("replace_leading_zero", "#", "0", ">", "find_num")
```

Now we need to remove the numerator and in the cell where its right-most digit would have been, we will place a marker "@" to show the position where the next digit of the subtraction result should go. To do this we first move right to the end of the subtraction result, bypassing all digits until we find an empty space, at which point we transition to a state named "**remove_num**".

```
for digit in digits:
    write("find_num", digit, digit, ">", "find_num")
write("find_num", "", "", ">", "remove_num")
```

Once in this state, we move to the right, removing all digits as we find them. Once we hit an empty space, move back to the left and transition into a state named "**drop_num_marker**". In this state, replace the empty space where the right-most digit of the numerator used to be with the "@" marker, transitioning into a state named "**copy_subtraction_result**".

```
for digit in digits:
    write("remove_num", digit, "", ">", "remove_num")
write("remove_num", "", "", "<", "drop_num_marker")
write("drop_num_marker", "", "@", "<", "copy_subtraction_result")
```

From here, we need to move left through the empty spaces until we find a digit. The first digit we encounter will be the one we need to copy, as we will remove them as we copy them. Upon finding a digit, move to the right into the state "**copying_result_{digit}**" where the digit in the state name is the digit we encountered. However, if instead of encountering a digit, we encounter the "#" marker which signifies that we have run out of digits to copy, we replace it with an empty space, move to the left, and transition to the state "**copied_subtraction_result**".

```
write("copy_subtraction_result", "", "", "<", "copy_subtraction_result")
write("copy_subtraction_result", "#", "", "<", "copied_subtraction_result")
for digit in digits:
    write("copy_subtraction_result", digit, "", ">", "copying_result_%s" % digit)
```

In this new state, we must move right through empty spaces until we find the "@" marker, at which point we replace it with the digit we are copying and move left, back into the state "**drop_num_marker**". This will cause the process to repeat until the entire subtraction result has been copied.

```
for digit in digits:
    write("copying_result_%s" % digit, "", "", ">", "copying_result_%s" % digit)
    write("copying_result_%s" % digit, "@", digit, "<", "drop_num_marker")
```

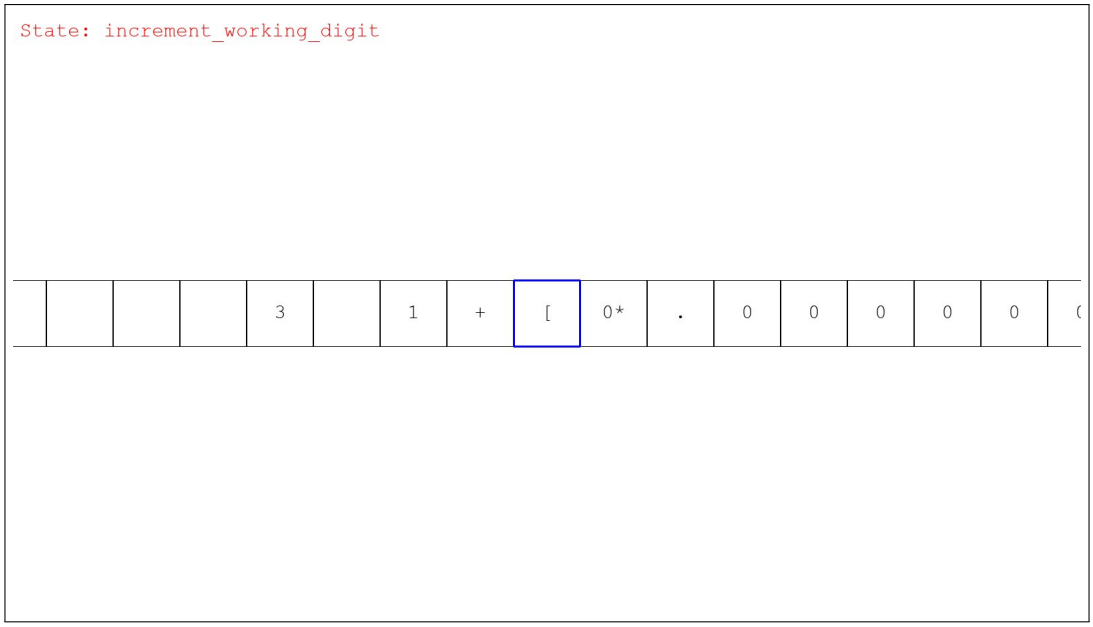
Once the copying has finished, we may need to continue moving left, deleting "#" markers (as there may have been more than one leading zero). Once we encounter an empty space, we move back to the right and into a state named "**check_sign**"

```
write("copied_subtraction_result", "#", "", "<", "copied_subtraction_result")
write("copied_subtraction_result", "", "", ">", "check_sign")
```

Now we need to check whether the sign is "+" or "-" so we know whether we need to increment or decrement the working digit. This means that we can simply move right through digits and spaces until we hit either a "+" (in which case we move right and transition to the state "**increment_working_digit**") or a "-" (in which case we move right and transition to the state "**decrement_working_digit**"), removing the "@" sign when we see it.

```
for digit in digits + [""]:
    write("check_sign", digit, digit, ">", "check_sign")
write("check_sign", "@", "", ">", "check_sign")
write("check_sign", "+", "+", ">", "increment_working_digit")
write("check_sign", "-", "-", ">", "decrement_working_digit")
```

At this point the tape looks like this:



We now need to move to the right, through digits, the start marker and potentially the decimal place, until we find a digit with an asterisk after it. Once we find the working digit there are four options:

1. We are in state "**increment_working_digit**" and the working digit is between 0 and 8 (inclusive). In this case, replace it with the digit one greater than its value, followed by an asterisk. Then, move right into the state "**place_change_marker**".
2. We are in state "**increment_working_digit**" and the working digit is 9. In this case, replace it with a "0*" and move to the left into the state "**overflow**".
3. We are in state "**decrement_working_digit**" and the working digit is between 1 and 9 (inclusive). In this case, replace it with the digit one less than its value, followed by an asterisk. Then, move right into the state "**place_change_marker**".
4. We are in state "**decrement_working_digit**" and the working digit is 0. In this case, replace it with a "9*" and move to the left into the state "**underflow**".

```
for symbol in digits + ["[", "."]:
    write("increment_working_digit", symbol, symbol, ">", "increment_working_digit")
    for i, digit in enumerate(digits[:-1]):
        write("increment_working_digit", digit+"*", digits[i+1]+"*", ">", "place_change_marker")
    write("increment_working_digit", "9*", "0*", "<", "overflow")

for symbol in digits + ["[", "."]:
    write("decrement_working_digit", symbol, symbol, ">", "decrement_working_digit")
    write("decrement_working_digit", "0*", "9*", "<", "underflow")
    for i, digit in enumerate(digits[1:]):
        write("decrement_working_digit", digit+"*", digits[i]+"*", ">", "place_change_marker")
```

We will now implement the "**underflow**" state. If we are in this state, and we see a digit between 1 and 9 (inclusive), replace it with the digit one less than it and then move right into the state "**place_change_marker**". If we see a 0, replace it with a 9, move to the left, and stay in the state "**underflow**". If we see a decimal point, ignore it and move to the left.

```
write("underflow", "0", "9", "<", "underflow")
for i, digit in enumerate(digits[1:]):
    write("underflow", digit, digits[i], ">", "place_change_marker")
write("underflow", ".", ".", "<", "underflow")
```

We will also implement the "**overflow**" state (which is very similar to the "**round**") state. If we are in this state and we see a digit between 0 and 8 (inclusive), replace it with the digit one greater than it and then move right into the state "**place_change_marker**". If we see a 9, replace it with a 0, move to the left, and stay in the state "**overflow**". If we

see a decimal point, ignore it and move to the left.

```
for i, digit in enumerate(digits[:-1]):
    write("overflow", digit, digits[i+1], ">", "place_change_marker")
write("overflow", "9", "0", "<", "overflow")
write("overflow", ".", ".", "<", "overflow")
```

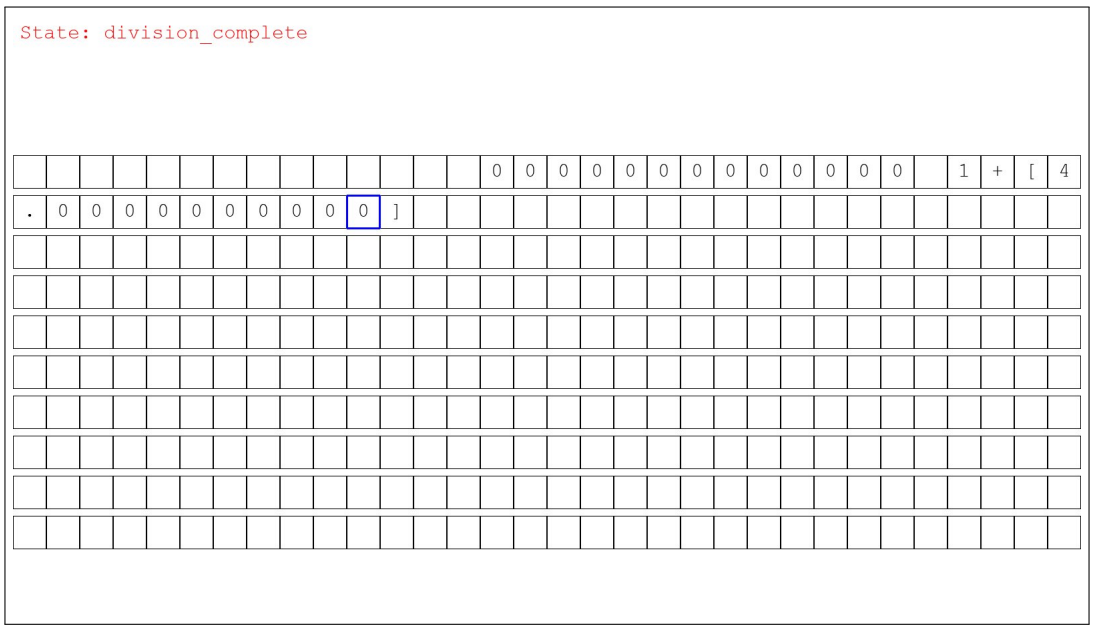
Now we need to implement the state "**place_change_marker**". This is because we need to leave the "!" marker on the right of the end marker to signify that the running total has changed, and so more terms need to be calculated. To do this we need to move right through digits and potentially a working digit or a decimal place. Once we reach the end marker "]", we move right into the state "**drop_change_marker**". Once here, we will either find an empty space or the "!" marker already there. In either case we write the "!" marker and move to the left into a state named "**placed_change_marker**". If we find ourselves in the state "**place_change_marker**" and the current cell contains either an empty space or the "!" marker, this means that the working digit is a temporary one. We can therefore forget about placing the marker, as we will only need to do so depending on how the temporary working digit rounds. We can therefore simply move left and transition to the "**return_to_subtraction**" state instead.

```
for digit in digits:
    write("place_change_marker", digit+"*", digit+"*", ">", "place_change_marker")
for symbol in digits+["."]:
    write("place_change_marker", symbol, symbol, ">", "place_change_marker")
write("place_change_marker", "", "", "<", "return_to_subtraction")
write("place_change_marker", "!", "!", "<", "return_to_subtraction")
write("place_change_marker", "]", "]", ">", "drop_change_marker")

write("drop_change_marker", "", "!", "<", "placed_change_marker")
write("drop_change_marker", "!", "!", "<", "placed_change_marker")
```

Once in the state "**placed_change_marker**", we need to continue to move left, transitioning into the state "**return_to_subtraction**" which we have already created.

```
write("placed_change_marker", "]", "]", "<", "return_to_subtraction")
```

If we now run the program, we will see the following output upon termination:



We have now finished implementing the division algorithm. As you can see, the Turing machine successfully divided 4 by 1 to ten decimal places, leaving 4.0000000000 in the space between the "[" and "]" markers. While this may seem like an uninteresting example, we have actually programmed the logic for dividing any natural number by any positive integer to any number of decimal places, and adding or subtracting the result to whatever is in the running total box, as well as keeping track of whether the division came out to zero. There is only one real function step left to create, and the rest is just housekeeping.

6 Moving to the Next Term

When we started up the program, we created the instructions to set up the calculation environment. This included writing the numerator (always 4), the denominator (initially 1) and the sign (initially "+") to the tape on the left hand side of the marked area. We now need to implement the procedure to flip the sign, increase the denominator by 2, and put the 4 back in the numerator's spot. We have terminated in the "**division_complete**" state with the read/write head somewhere within the running total. If we are to the right of the decimal place, we therefore need to keep travelling left, through digits to get back to the decimal place. However, if we were to the left of the decimal place, we will instead encounter the start marker, "[" first. If this happens, we can move right and transition into a state named "**reset_working_digit**" so that we can set the first digit to be the working digit again. If we do indeed reach the decimal place, we need to move left again and transition into the state "**reset_working_digit**" . In this state, we replace whatever digit we see with the same digit but with an asterisk concatenated onto the end (We can actually make an optimisation here because we know that the first digit can only ever be 2, 3, or 4). We than move left again onto the start marker, and we can even stay in the same state, so long as we write a rule which says that if we are in the state "**reset_working_digit**" and we see a start marker "[", we move left into a state named "**flip_sign**".

```
for digit in digits:
    write("division_complete", digit, digit, "<", "division_complete")
write("division_complete", ".", ".", "<", "reset_working_digit")
write("division_complete", "[", "[", ">", "reset_working_digit")

for digit in "234":
    write("reset_working_digit", digit, digit+"*", "<", "reset_working_digit")
write("reset_working_digit", "[", "[", "<", "flip_sign")
```

The "**flip_sign**" state is quite simple. If we see a "+", replace it with a "-" and vice versa. Either way, move to the left and into the state "**double_increment_denominator**".

```
write("flip_sign", "+", "-", "<", "double_increment_denominator")
write("flip_sign", "-", "+", "<", "double_increment_denominator")
```

We now need to implement the double-incrementation. When in the "**double_increment_denominator**" state, we will be directly over the least significant digit of the denominator. Since this will always be odd, we only need to consider the odd digits. If the digit in the cell is less than 9, then replace it with the digit which is two greater than it, move left, and transition into a state named "**reset_numerator**". Otherwise, replace the digit with a 1, and move left into the state "**increment_denominator**". Here there are three possibilities:

1. The digit in the current cell is between 0 and 8 (inclusive) in which case replace it with the digit which is one greater than it, and move to the left into the state "**reset_numerator**"
2. The digit in the current cell is a 9, in which case replace it with a 0 and move to the left, staying in the state "**increment_denominator**"
3. The current cell is blank (meaning that the denominator needs to become one digit longer), in which case replace it with a 1, and move left into the state "**denominator_overflow**"

```
odds = "13579"
for i, digit in enumerate(odds[:-1]):
    write("double_increment_denominator", digit, odds[i+1], "<", "reset_numerator")
write("double_increment_denominator", "9", "1", "<", "increment_denominator")

for i, digit in enumerate(digits[:-1]):
    write("increment_denominator", digit, digits[i+1], "<", "reset_numerator")
write("increment_denominator", "9", "0", "<", "increment_denominator")
write("increment_denominator", "", "1", "<", "denominator_overflow")
```

If we are in the state "**denominator_overflow**", then we are currently in a cell which contains a digit, but should instead now be the space between the numerator and the denominator. We can therefore replace it with an empty space, and then move left into the state "**place_four**".

```
for digit in digits:
    write("denominator_overflow", digit, "", "<", "place_four")
```

If instead we are in the "**reset_numerator**" state, that means that the read/write head is currently somewhere in the middle of the denominator, or in the space to the left of it. Either way, we need to keep moving left through any digits until we find that space, at which point we move left again into the state "**place_four**".

```
for digit in digits:
    write("reset_numerator", digit, digit, "<", "reset_numerator")
write("reset_numerator", "", "", "<", "place_four")
```

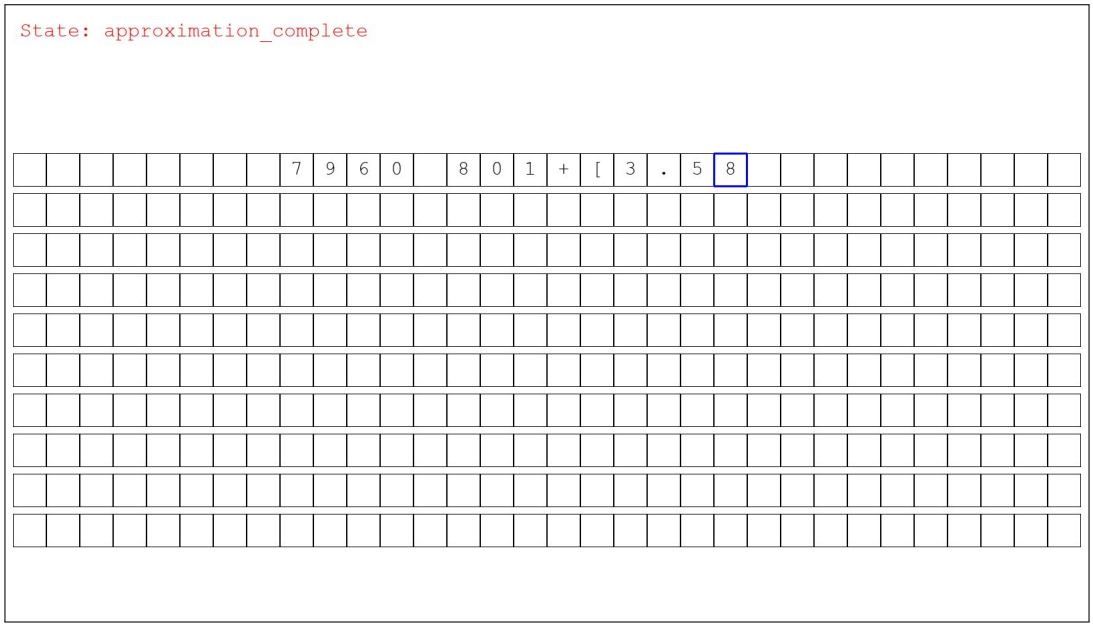
Once in the "**place_four**" state, we need to replace whatever is in the current cell (whether it is a digit or an empty space) with a 4. We then move left into a state named "**wipe_numerator**". In this state, we need to continue to move left, deleting all digits we find until we come across an empty space, at which point we move right again into the state "**restart_division**".

```
for digit in digits + [""]:
    write("place_four", digit, "4", "<", "wipe_numerator")

for digit in digits:
    write("wipe_numerator", digit, "", "<", "wipe_numerator")
write("wipe_numerator", "", "", ">", "restart_division")
```

Once we have gotten into the "**restart_division**" state, we simply need to move left through digits and spaces until we find the sign. Once we find the sign, we move back to the left, and transition into the state called "**take_last_denom_digit**" to kick-start the next division operation.

```
for digit in digits + [""]:
    write("restart_division", digit, digit, ">", "restart_division")
write("restart_division", "+", "+", "<", "take_last_denom_digit")
write("restart_division", "-", "-", "<", "take_last_denom_digit")
```

We have now completed the entirety of the mathematical portion of this program. If we leave the program running, when it finally terminates, the tape will look like this (note that in all following screenshots I have used $n = 2$ – starting the program with a symbol "2" on the tape – so that the program will terminate in a reasonable amount of time, however if left for long enough the program will indeed terminate regardless of your choice of n, provided the computer on which you are running the simulator does not run out of memory):



The final process we need to implement is cleaning up the tape, and making the result look nice and presentable.

7 Cleaning the Tape

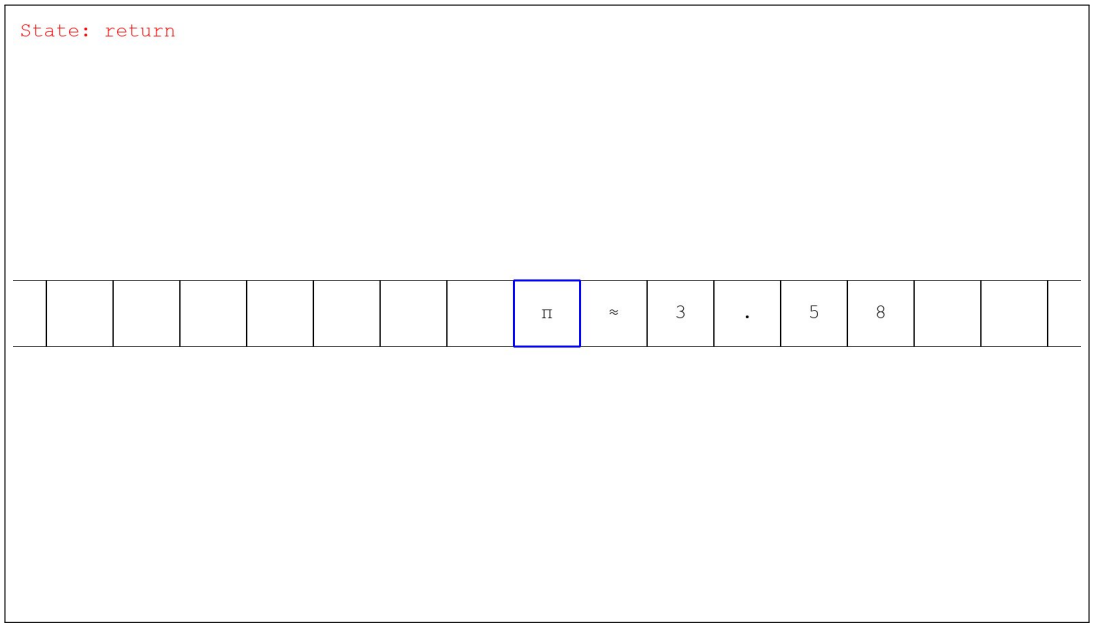
Once the approximation has been generated, the program terminates in the state "**approximation_complete**" with the read/write head over the last digit of the approximation. The end marker "]" and the temporary working digit have already been removed. We can now move to the left through the digits and the decimal place until we reach the start marker "[". Upon finding it, we can replace it with an "≈" sign (`"\u 2248"`), and move left, transitioning to a state named "**drop_pi**". Then, we can replace the "+" or "-" with a "π" symbol (`"\u 03C0"`), move left, and transition into a state named "**clean**".

```
for digit in digits + ["."]:
    write("approximation_complete", digit, digit, "<", "approximation_complete")
write("approximation_complete", "[", "\u2248", "<", "drop_pi")

write("drop_pi", "+", "\u03C0", "<", "clean")
write("drop_pi", "-", "\u03C0", "<", "clean")
```

Once in the "**clean**" state, we simply need to erase every digit to the left, and then return to the "π" symbol. To do this, we implement rules such that if we encounter a digit, we replace it with an empty space, move to the left, and remain in the "**clean**" state. If instead we encounter a space, we move to the left, and move to the "**clean_**" state. When in the "**clean_**" state if we see a digit, replace it with an empty space, move left, and transition back to the "**clean**" state, whereas if we see another space, we know we have reached the end, and so can move right, and transition to a state named "**return**". In return, we move right through each space, remaining in the state "**return**". Once we reach the "π" symbol, there will be no rules which apply and the program will terminate gracefully.

```
for digit in digits:
    write("clean", digit, "", "<", "clean")
write("clean", "", "", "<", "clean_")

for digit in digits:
    write("clean_", digit, "", "<", "clean")
write("clean_", "", "", ">", "return")

write("return", "", "", ">", "return")
```

Once the program terminates, the tape will look like this:



8 Conclusion

We have now created a Turing machine program which can generate approximations of π which are arbitrarily accurate. The ruleset contains 7375 rules, utilising 725 states. While the program is incredibly slow (in fact its time complexity is $\mathcal{O}(10^n)$, taking tens of minutes to even create an approximation to 2 decimal places – especially considering the fact that

it takes a much larger approximation to be accurate as π to 2 decimal places), I believe that there is some importance to this program. In particular, I think that there is profound beauty in the way in which a complex task can be performed on such a simple machine. For me, Turing machine programs are a wonderful demonstration of how complexity can arise from simplicity, and designing them provides a very powerful experience of what abstraction feels like on such a low level. We start with very basic functionality – the machine has an internal state and can read a symbol. The machine then writes a new symbol, shifts one cell to the left or right, and updates its state. We can then use that architecture to design a more complex process like subtraction, and in doing so we abstract away the simplicity of the machine. Next, when implementing division, we no longer need to think about how we do this using Turing machine instructions, but instead how we achieve the task with the more complex functionality we've already designed. Having said this, the intricacy of the tape setup means we never truly lose sight of the low-level processes. To me, this interplay between high and low level thinking which is necessary when writing Turing machine programs really gives an insight into the very nature of computation itself.

This program which we have written demonstrates how π fits in to this abstract world of computability, which is why I believe it to be of value.