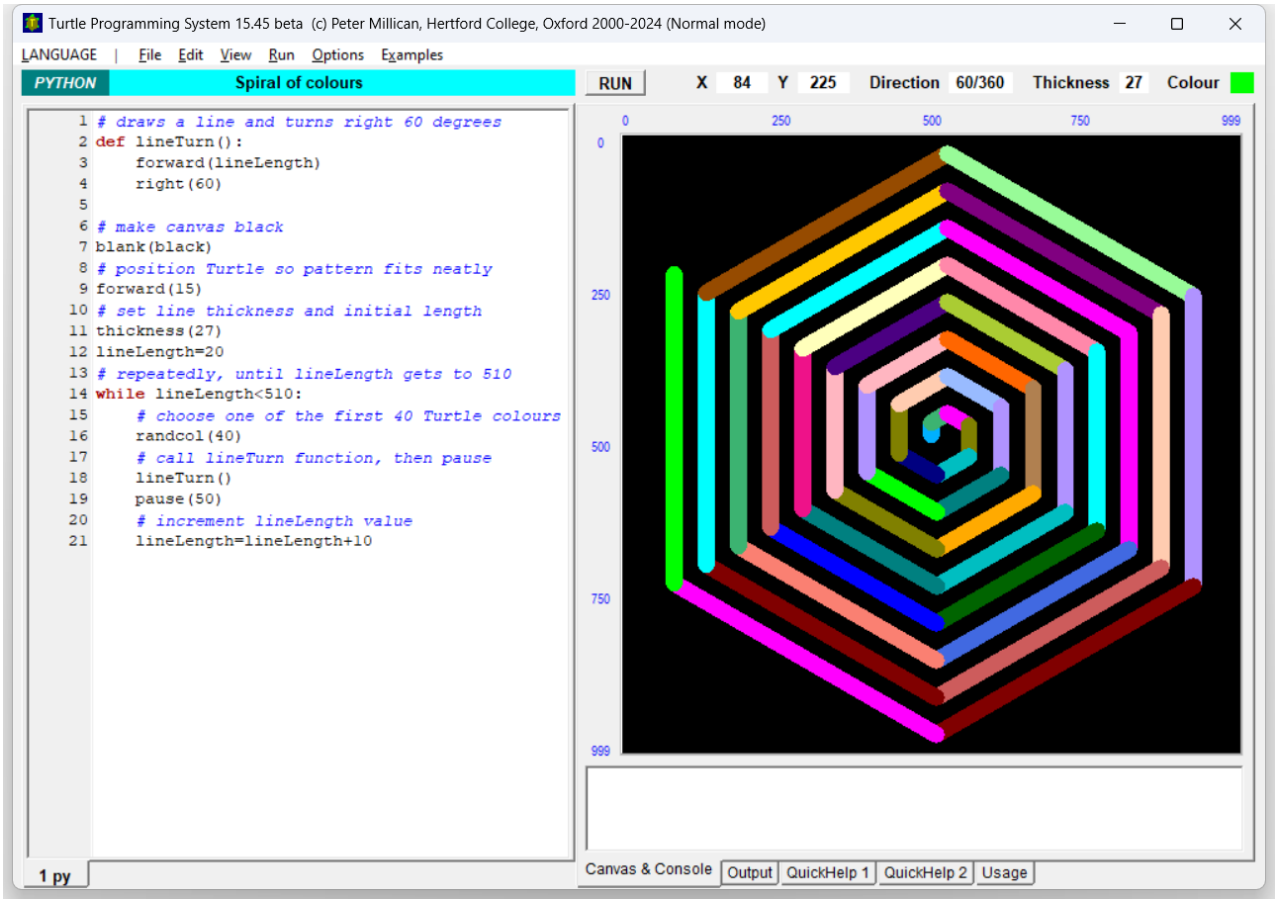
Turtle Machine 1 – PCode and the Stack

This document introduces the "Turtle Machine", on which programs written within the *Turtle System* ultimately run. You might write your programs in Turtle Python, or Turtle Pascal, or Turtle BASIC, but whichever "source" language you choose, your program is *compiled* – i.e. translated – into the "machine code" of an underlying *Turtle Machine*.

1. A Simple Example Program

To start, go to the "Examples" menu, highlight "Examples 2 – functions and simple recursion", and click on "Spiral of colours". The Python version of this program (which is also available in Pascal and Basic if you now switch to those using the "LANGUAGE" menu) is shown here in the program editor on the left of the *Turtle* window. If you click on "RUN" (near the top centre of the window), then a coloured pattern will appear on the Canvas at the right of the window (though the colours will be random):



This program is only intended to illustrate PCode, and should not be taken as a model of good programming technique! For example, it begins with an almost trivially simple *function* called "lineTurn" which just moves the *Turtle* forward (by the value of the variable "lineLength", whatever that happens to be) and turns it right by 60 degrees; but in a working program, this would not deserve a function to itself, because this makes the program needlessly more complicated, and adds three lines of code while only saving one. For our purposes, however, the simplicity of the function will be helpful for focusing on the main structural points.

Before proceeding further, check that you understand how the program works. It starts by blanking the Canvas to black, then moves the *Turtle* forward 15 units, sets its *thickness* to 27 (so as to produce a thick line as it moves) and the initial value of the (integer) variable "lineLength" to 20. It then enters a "while"

loop which continues until the value of "lineLength" reaches 510 or above. Within that loop, it first sets the *Turtle*'s colour to one of its first 40 "native" colours (these can be seen by clicking on "Quickhelp 1" at the bottom of the window, and then on "Operators/Colours"). It then calls the "lineTurn" function, which as we have seen moves the *Turtle* forward by the current value of "lineLength" and turns its direction right by 60 degrees. Then there is a pause of 50 milliseconds (i.e. one twentieth of a second), which makes the spiral take time to grow. And finally within the loop, "lineLength" is incremented by 10, so its value will increase from the initial 20 to 30, then 40, then 50, and so on all the way up to 510, at which point the loop will end. (Again, this program does not pretend to be a model – a loop like this would more naturally be implemented as a "for" rather than "while" loop, but the latter produces much more simple PCode.)

2. Selecting Machine Mode

The *Turtle System* standardly starts up in "Normal mode" (as can be seen from the title bar of the software window). To see the PCode, you are going to need to change into "Machine mode", which you do by clicking on the bottom entry in the "View" menu. On doing this, another five tabs will appear at the bottom right of the software window (we call this right side the "tabs section"). Here is a brief summary of the facilities provided by all ten tabs:

"Canvas": Labelled "Canvas & Console" in Normal mode (as in the previous screenshot). This displays the Canvas and also the "Console", which is a small area below the Canvas where printed output appears – it can also be used for text input within programs that provide for that.

"Output": This displays the "Output" text box, which preserves the text that appears in the Console. Enables extensive text display from relevant programs, as well as scrolling through it.

"Help1": Labelled "QuickHelp 1" in Normal mode (as in the previous screenshot). This provides a sequence of eight panels summarising the basic features of programming in the currently chosen language, along with display of the available colours, fonts, and cursors.

"Help2": Labelled "QuickHelp 2" in Normal mode (as in the previous screenshot). This displays a user-configurable table, summarising the "Native Turtle Commands" (such as "blank", "forward" and "thickness"). This is extremely useful for quick reference when programming, and it's worth familiarising yourself with the various display options. When experienced, you will probably prefer to deselect "Include all groups", and pick the relevant group individually.

"Usage": This summarises the various structures (e.g. "while") and native commands (e.g. "forward") that occur in your program, together with line numbers. It can be useful for finding key parts within a long program, and also for ensuring that you have conformed with any coursework requirements (e.g. if you are required to demonstrate use of some range of structures).

"Comments": This collects together all of the comments in the program, which again can be useful for finding key parts if your program has been annotated appropriately.

"Syntax": This shows the program (not including comments) divided into "lexemes" – i.e. individual items of text – and displays information about how they have been analysed. This can be helpful if you are trying to understand the process of *compilation*.

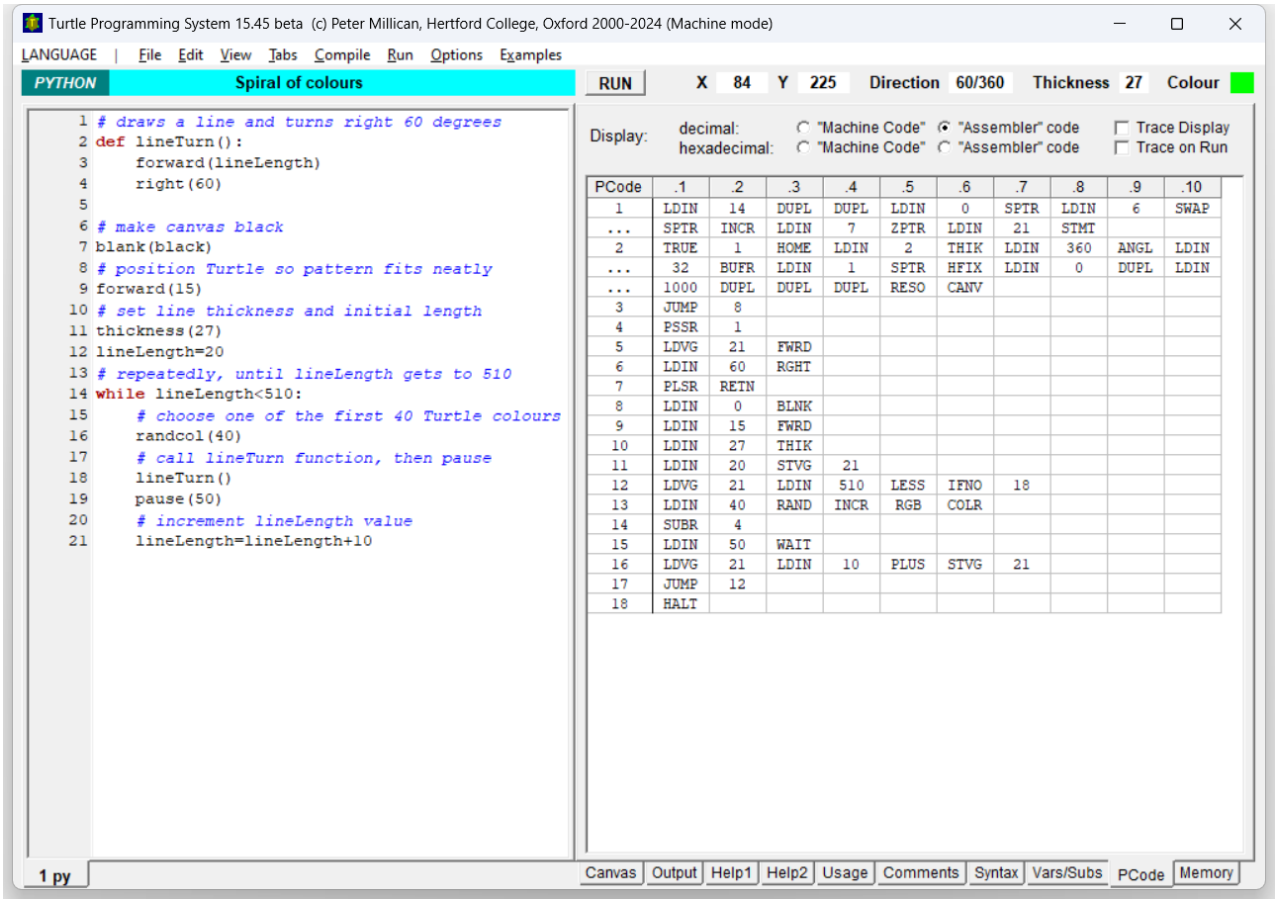
"Vars/Subs": Displays information about the variables and subroutines that occur in the program. Note that for the current program, it shows not only "lineLength" but also the six *Turtle attributes* "turtx", "turty", "turtd", "turta", "turtt", and "turtc", which keep track respectively of the *Turtle*'s x- and y-coordinates, direction, angle factor, pen thickness and colour.

"PCode": Shows the compiled PCode of the current program. If you select "Trace on Run" and then RUN the program again, the table that appears below will show all of the PCode instructions that have been executed in the program (here over 1200!).

"Memory": Displays the memory of the (virtual) *Turtle Machine*, divided into the Memory Stack (distinct from the Evaluation Stack described below) and the Memory Heap, together with a copy of the Variables table (from the "Vars/Subs" tab). To show the current memory, click on "Show Current State". Notice that the *Turtle* attribute values are stored in memory at locations 15 to 20, immediately before the "lineLength" value which is stored at location 21.

3. Understanding PCode

If you now click on the "PCode" tab, the Canvas will be replaced by the PCode display, as shown here:



When your program is run, the "source code" (e.g. in Python, read from the program editor) has first to be *compiled* (i.e. *translated*) into a form of "machine code" which actually consists of a sequence of numbers. These numbers can be seen if you click on the "Machine Code" button above the PCode table, but it is easier to understand what's going on if we focus on the "Assembler" code display as shown above, in which many of the numbers are replaced with mnemonic symbols (e.g. 160 is replaced with "LDIN", 2 with "DUPL"). *When the program is executed, it is actually these PCode instructions that are being carried out*.

Turtle Machine PCode runs on a "virtual machine" which is implemented in software (i.e. the *Turtle System* software simulates an imagined hardware chip which has been designed to respond appropriately to these numeric commands). Many modern systems (including Java and Python) use virtual machines in this sort of way – it has the advantage that the software they produce can be made to run on a variety of hardware, simply by implementing a version of the virtual machine on that hardware (so in this sense the code is "portable", hence the name "PCode"). Another feature that the *Turtle Machine* shares with many other virtual machines is that it has a *Stack-based* architecture, which means that most of its operations involve an "Evaluation Stack" which is a sort of structured short-term memory, holding relevant values in order with new values being added in sequence at the top of the Stack (thus pushing other values down in the Stack), and values also being taken off in turn from the top (thus operating "last in – first out"). This

kind of operation can be seen in the current program, as explained below, but it's perhaps clearest in the case of numerical calculations, as illustrated in the next section.

Unlike other virtual machines, however, the *Turtle Machine* has been designed specifically for *ease of student understanding*, and for that reason, its PCode is conveniently divided up into lines, as we can see in the last screenshot. The first two lines are long, consisting of 18 and 26 PCode values respectively – these are concerned with such things as setting up the *Turtle* attributes in memory, setting up a keyboard buffer (for keyboard input), and fixing the initial Canvas dimensions and settings. Here we'll ignore those two lines, and focus on lines 3 to 18. The following table will help to explain what is going on here, bearing in mind that – as we saw in the "Memory" display" – variable "lineLength" is stored at location 21:

	3 JUMP 8	Jump to start of main program – line 8
def lineTurn()	4 PSSR 1	"Push" subroutine number = 1
forward(lineLength)	5 LDVG 21 FWRD	Load global location 21, move forward
right(60)	6 LDIN 60 RGHT	Load integer 60, turn right
	7 PLSR RETN	"Pull" subroutine number, and return
blank(black)	8 LDIN 0 BLNK	Load integer 0 (Black), and blank Canvas
forward(15)	9 LDIN 15 FWRD	Load integer 15, move forward
thickness(27)	10 LDIN 27 THIK	Load integer 27, thickness
lineLength = 20	11 LDIN 20 STVG 21	Load integer 20, store in global location 21
while lineLenth<510:	12 LDVG 21 LDIN 510	Load global location 21, load integer 510 …
	LESS	Is the former less than the latter?
	IFNO 18	If not, jump ahead to line 18
randcol(40)	13 LDIN 40 RAND	Load 40, replace with random number 0-39
	INCR	Increment number (to 1-40)
	RGB COLR	Convert to native RGB code, set *Turtle* colour
lineTurn()	14 SUBR 4	Call subroutine at line 4
pause(50)	15 LDIN 50 WAIT	Load integer 50, pause 50 milliseconds
lineLength = lineLength+10	16 LDVG 21 LDIN 10	Load global location 21, load integer 10
	PLUS STVG 21	Add the two values, store in location 21
	17 JUMP 12	Jump back to line 12, to form *while* loop
	18 HALT	

Note in particular the stack operations at PCode lines 12 and 16. To take line 16 first, LDVG 21 and LDIN 10 both load a value onto the Stack. LDVG 21 loads the contents of global memory location 21 (i.e. the "lineLength" variable location); LDIN 10 the integer 10. PLUS then adds the two top values on the Stack, removing them and leaving the result there. STVG 21 then stores the new result into location 21, thus updating the value of "lineLength". Line 12 is more complicated, though it starts similarly by loading the contents of global memory location 21 and an integer – this time 510 – onto the Stack. Next, LESS tests whether the former value (now placed *second* on the Stack) is *less than* the latter value (i.e. the *top* value on the Stack), removing them and leaving the value 1 on the Stack if the result is *true*, 0 if the result is *false*. And then IFNO 18 jumps to line 18 (exiting the while loop and halting the program) if the result was 0.

4. Arithmetic on the Stack

Stack arithmetic involves treating numerical operators in an unfamiliar order, as we can see if we enter the very simple program:

```
n = 5 + (2 * 3 - 1) * 4 + 2 * 6
print(n)
```

This gives the result 37, because multiplication is given precedence over addition or subtraction, hence the sum is calculated as:

```
n = 5 + ((6 - 1) * 4) + (2 * 6)
```

and hence:

```
n = 5 + 20 + 12 =  37
```

If we look at the PCode produced, however, we see the Stack being used to perform the arithemetic, as follows. After each instruction is shown the contents of the Evaluation Stack that results, with the top of the Stack at the left (after the "|" symbol). This sequence can be inspected directly if you choose "Trace on Run" from the PCode tab, and then RUN the program.

```
LDIN 5          | 5
LDIN 2          | 2 5
LDIN 3          | 3 2 5
MULT            | 6 5
LDIN 1          | 1 6 5
SUBT            | 5 5
LDIN 4          | 4 5 5
MULT            | 20 5
PLUS            | 25
LDIN 2          | 2 25
LDIN 6          | 6 2 25
MULT            | 12 25
PLUS            | 37
```

You might have come across "postfix" notation (i.e. operators coming *after* the operand values), which is a way of representing arithmetic expressions in a way that corresponds to their Stack-based processing, and avoids the need for brackets. Thus expressed, the sum above would be:

```
5 2 3 * 1 - 4 * + 2 6 * +
```

which is more efficient than the "infix" alternative (i.e. operators coming *between* the operands):

```
5 + (2 * 3 - 1) * 4 + 2 * 6
```

especially if we spell out explicitly the precedence of operations applying to the latter:

```
(5 + (((2 * 3) - 1) * 4)) + (2 * 6)
```

Most computer languages, of course, use our familiar infix notation (and take account of the standard rules of precedence). One of the important operations that has to be performed when compiling programs containing such notation is to translate it into a sequence of Stack operations, as illustrated above, and thus implicitly into a postfix structure.