What Is A Computer Program?

Scientific Computing
Fall, 2018
Paul Gribble

What is a computer program? What is code? What is a computer language? A computer program is simply a series of instructions that the computer executes, one after the other. An instruction is a single command. A program is a series of instructions. Code is another way of referring to a single instruction or a series of instructions (a program).

1 High-level vs low-level languages

The CPU (central processing unit) chip(s) that sit on the motherboard of your computer is the piece of hardware that actually executes instructions. A CPU only understands a relatively low-level language called machine code. Often machine code is generated automatically by translating code written in assembly language, which is a low-level programming language that has a relatively direcy relationship to machine code (but is more readable by a human). A utility program called an assembler is what translates assembly language code into machine code.

In this course we will be learning how to program in MATLAB, which is a high-level programming language. The "high-level" refers to the fact that the language has a strong abstraction from the details of the computer (the details of the machine code). A "strong abstraction" means that one can operate using high-level instructions without having to worry about the low-level details of carrying out those instructions.

An analogy is motor skill learning. A high-level language for human action might be *drive your car to the grocery store and buy apples*. A low-level version of this might be something like: (1) walk to your car; (2) open the door; (3) start the ignition; (4) put the transmission into Drive; (5) step on the gas pedal, and so on. An even lower-level description might involve instructions like: (1) activate your gastrocnemius muscle until you feel 2 kg of pressure on the underside of your right foot, maintain this pressure for 2.7 seconds, then release (stepping on the gas pedal); (2) move your left and right eyeballs 27 degrees to the left (check for oncoming cars); (3) activate your pectoralis muscle on the right side of your chest and simultaneously squeeze the steering wheel with the fingers on your right hand (steer the car to the left); and so on.

For scientific programming, we would like to deal at the highest level we can, so that we can avoid worrying about the low-level details. We might for example want to plot a line in a Figure and colour it blue. We don't want to have to program the low-level details of how each pixel on the screen is set, and how to generate each letter of the font that is used to specify the x-axis label.

As an example, here is a *hello, world* program written in a variety of languages, just to give you a sense of things. You can see the high-level languages like MATLAB, Python and R are extremely readable and understandable, even though you may not know anything about these languages (yet). The C code is less readable, there are lots of details one may not know about... and the assembly language example is a bit of a nightmare, obviously too low-level for our needs here.

MATLAB

```
disp('hello, world')
```

Python

```
print "hello, world"
```

R

```
cat("hello, world\n")
```

Javascript

```
document.write("hello, world");
```

Fortran

```
print *,"hello, world"
```

C

```
#include <stdio.h>
int main (int argc, char *argv[]) {
  printf("hello, world\n");
  return 0;
}
```

8086 Assembly language

```
; this example prints out  "hello world!" by writing directly to video memory.
; first byte is ascii character, next is character attribute (8 bit value)
; high 4 bits set background color and low 4 bits set foreground color.

org 100h

; set video mode
mov ax, 3     ; text mode 80x25, 16 colors, 8 pages (ah=0, al=3)
int 10h       ; do it!

; cancel blinking and enable all 16 colors:
mov ax, 1003h
mov bx, 0
int 10h

; set segment register:
mov     ax, 0b800h
mov     ds, ax

; print "hello world"
```

```
mov [02h], 'H'
mov [04h], 'e'
mov [06h], 'l'
mov [08h], 'l'
mov [0ah], 'o'
mov [0ch], ','
mov [0eh], 'W'
mov [10h], 'o'
mov [12h], 'r'
mov [14h], 'l'
mov [16h], 'd'
mov [18h], '!'

; color all characters:
mov cx, 12  ; number of characters.
mov di, 03h ; start from byte after 'h'

c:  mov [di], 11101100b   ; light red(1100) on yellow(1110)
    add di, 2 ; skip over next ascii code in vga memory.
    loop c

; wait for any key press:
mov ah, 0
int 16h

ret
```

2 Interpreted vs compiled languages

Some languages like C and Fortran are compiled languages, meaning that we write code in C or Fortran, and then to run the code (to have the computer execute those instructions) we first have to translate the code into machine code, and then run the machine code. The utility function that performs this translation (compilation) is called a compiler. In addition to simply translating a high-level language into machine code, modern compilers will also perform a number of optimizations to ensure that the resulting machine code runs fast, and uses little memory. Typically we write a program in C, then compile it, and if there are no errors, we then run it. We deal with the entire program as a whole. Compiled program tend to be fast since the entire program is compiled and optimized as a whole, into machine code, and then run on the CPU as a whole.

Other languages, like MATLAB, Python and R, are interpreted languages, meaning that we write code which is then translated, command by command, into machine language instructions which are run one after another. This is done using a utility called an interpreter. We don't have to compile the whole program all together in order to run it. Instead we can run it one instruction at a time. Typically we do this in an interactive programming environment where we can type in a command, and observe the result, and then type a next command, etc. This is known as the read-eval-print (REPL) loop. This is advantageous for scientific programming, where we typically spend a lot of time exploring our data in an interactive way. One can of course run a program such as this in a batch mode, all at once, without the interactive REPL environment... but this doesn't change the fact that the translation to machine code still happens one line at a time, each in isolation. Interpreted languages tend to be slow, because every single command is taken in isolation, one after the other, and in real time translated into machine code which is then executed in a piecemeal fashion.

For interactive programming, when we are exploring our data, interpreted languages like MATLAB, Python

and R shine. They may be slow but it (typically) doesn't matter, because what's many orders of magnitude slower, is the firing of the neurons in our brain as we consider the output of each command and decide what to do next, how to analyse our data differently, what to plot next, etc. For batch programming (for example fMRI processing pipelines, or electrophysiological recording signal processing, or numerical optimizations, or statistical bootstrapping operations), where we want to run a large set of instructions all at once, without looking at the result of each step along the way, compiled languages really shine. They are much faster than interpreted languages, often several orders of magnitude faster. It's not unusual for even a simple program written in C to run 100x or even 1000x faster than the same program written in MATLAB, Python or R.

A 1000x speedup may not be very important when the program runs in 5 seconds (versus 5 milliseconds) but when a program takes 60 seconds to run in MATLAB, for example, things can start to get problematic.

Imagine you write some MATLAB code to read in data from one subject, process that data, and write the result to a file, and that operation takes 60 seconds. Is that so bad? Not if you only have to run it once. Now let's imagine you have 15 subjects in your group. Now 60 seconds is 15 minutes. Now let's say you have 4 groups. Now 15 minutes is one hour. You run your program, go have lunch, and come back an hour later and you find there was an error. You fix the error and re-run. Another hour. Even if you get it right, now imagine your supervisor asks you to re-run the analysis 5 different ways, varying some parameter of the analysis (maybe filtering the data at a different frequency, for example). Now you need 5 hours to see the result. It doesn't take a huge amount of data to run into this sort of situation.

Now imagine if you could program this data processing pipeline in C instead, and you could achieve a 500x speedup (not unusual), now those 5 hours turn into 36 seconds (you could run your analysis twice and it would still take less time than listening to Stairway to Heaven a dozen times). All of a sudden it's the difference between an overnight operation and a 30 second operation. That makes a big difference to the kind of work you can do, and the kinds of questions you can pursue.

MATLAB is pretty good about using optimized, compiled subroutines for operations that it knows it can farm out (e.g. many matrix algebra operations), so in many cases the difference between MATLAB and C performance isn't as great as it is for others. MATLAB also has a toolbox (called the MATLAB Coder) that will allow you to generate C code from your MATLAB code, so in principle you can take slow MATLAB code and generate faster, compiled C code. In practice this can be tricky though.

My own approach is to use interpreted languages like Python, R, MATLAB, etc, for prototyping: exploring small amounts of data, for developing an approach, and algorithms, for analysing data, and for generating graphics. When I have a computation, or a simulation, or a series of operations that are time-consuming, I think about implementing them in C. Interpreted languages for *prototyping* and *exploration*, and C for *performance*.