1. The following program outputs “0.000828” when run on a lab machine.

```
#include <stdio.h>
#define ITERATIONS 100000 /* 100 thousand */
#define INCREMENT .00003f /* 3 hundred thousandths */

int main(void) {
int i;
float x = 0.f;
for (i = 0; i < ITERATIONS; ++i) {
x = x + INCREMENT;
}
printf("%f\n", (x - 3.f)/3.f);
}
```

Assume that a `float` uses the IEEE standard Single Precision format. Why doesn’t the program print 0?

.00003 can’t be represented exactly as a floating point number (it’s a repeating “binary decimal” in base 2), so it actually represents a slightly different number. Adding this number 100 000 exaggerates this rounding error. Additionally, we can’t represent many of the intermediate results (values of x during the loop) as single precision floats, so there are additional sources of rounding error. Although in some cases we might get lucky and have these rounding errors cancel out, that clearly did not happen in this case.

If the ITERATIONS were increased to 100 000 000 and INCREMENT decreased to 0.000 000 03, would the error be larger than the original program? Why or why not?

It would have a larger error. For all single precision floating point numbers x >= 1, the closest single precision floating point number to x + 0.000 000 03 is x (since 0.000 000 03 << 2-23 and we only have 23 fraction bits). Thus, x can never exceed 1, so the error must be at least 2/3rds (in magnitude), which is considerably larger in magnitude than the original value printed.

If ITERATIONS is changed to 3 times 220 and INCREMENT is changed to 2-20, then the program outputs “0.000000” on lab machines. Why?

k * 2-20 can be represented exactly as a single precision floating point number for all k < 223, so all the calculations are exact in this case.

CALL

1. Why does the linker even exist? Why can’t the assembler just spit out
a complete program every time? Why does PC-relative addressing in branches make linking easier?

Saving time is one reason. This process allows us to compile .c files to machine code separately and stitch them together later, so we don’t have to recompile the entire program every time. We also can store pre-compiled libraries.

2. What kind of instructions need to be modified when we are linking object files together?

Pretty much anything that depends on an absolute address. So jumps for sure, but also certain luis and oris if they were loading an address into a register.

3. What metadata about the code in an object file does the linker need to correctly modify these instructions?

The linker needs to know what instructions are referencing addresses, so that the linker can rewrite them, but it also needs to know what symbols this module provides; other modules will declare some dependencies on symbols in their relocation table, and if we don’t tell the linker we are providing that symbol, and tell it where that symbol lives, the linker won’t be able to satisfy these dependencies.

4. Dynamically Linked Libraries are often called shared libraries. What do you think is the justification behind this terminology?

The same library files are shared across all the programs that use them. There’s another reason they’re called “shared”, as well, that we’ll get into later in the course: specifically, through the magic of virtual memory, they can occupy just one place in physical memory across all the programs that are using them.