**CG2028 Assignment Report**

**Explain the console window output that you see below the “ARM ASM & Integer version:” heading.**

The output is “xsol : -6.000000”.

The C program calls the optimize function with the values of xsol, a, b and c. Since the optimize function signature indicates that it accepts only integers, the value -6.78 is truncated to -6 and passed to the optimize function as the first parameter. The first parameter is placed in R0, second parameter in R1, third parameter in R2 and last parameter in R3.

The C program expects the return value to be in R0. Since the optimize function does nothing, the value in R0 remains intact which is then interpreted as the return value of -6. The value is then assigned to a float variable resulting in the output value of -6.000000.

**Explain what can be done in the C program to convert the floating point value x0 to its integer version xi0 so that the decimal part of x0 is not simply discarded.**

The graph can be scaled along the x-axis by a factor such as 10,000 so that the decimal part is not simply discarded. The return value of optimize can then be divided by the scaling factor to obtain the non-scaled version with some decimal information.

**Experiment Results with our program**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **a** | **b** | **c** |  | **ASM x\* when**  **(no. of iterations)** | **C x\* when**  **(no. of iterations)** | **True minimum** |
| 3 | 4 | -3 | -6.78 | -0.666800 (13) | -0.666667 (21) | -2/3 |
| 3 | 4 | -3 | 6.78 | -0.666600 (14) | -0.666667 (22) | -2/3 |
| 1 | -6 | 14 | 4.23 | 3.000400 (39) | 3.000000 (66) | 3 |
| 1 | -6 | 14 | 3.00 | 3.000000 (1) | 3.000000 (1) | 3 |
| 1 | -6 | 14 | -2.12 | 2.999600 (46) | 3.000000 (73) | 3 |
| 2 | 48 | 289 | 0.01 | -11.999800 (23) | -11.999999 (33) | -12 |
| 2 | 48 | 289 | -0.001 | -11.999800 (23) | -11.999999 (33) | -12 |
| 5 | 0 | -9 | 1.00 | 0.000000 (2) | 0.000000 (2) | 0 |

**Compare these two solutions with the true solution obtained through mathematical analysis in your report and discuss why they are the same or different.**

Both C and ASM versions are close to the true solution most of the times but the C version has better precision. This is because in the ASM version, precision is lost through data scaling process and integer division. However, these two sources of precision loss are not present in the C program. As can be seen from the row of the table above (a = c, b = 0, c = -9, x\_i = 1.00), the values of both C and ASM are the same when there are no decimal parts involved in any part of the calculation.

**Compare the number of iterations needed to converge to the optimal value x\* and the accuracy of the solution in both the assembly language and C cases.**

The number of iterations needed to converge in ASM is lower than or equal to that of C, and the accuracy of the solution is better in C than in ASM. Since the ASM version terminates at a x\* with lower accuracy, fewer iterations are required.

**Modify the step size parameter λ to 0.5 and rerun the program. Describe what you observe in terms of the number of iterations needed to converge and the accuracy of the solution in both the assembly language and C cases.**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **a** | **b** | **c** |  | **ASM x\* when**  **(no. of iterations)** | **C x\* when**  **(no. of iterations)** | **True minimum** |
| 3 | 4 | -3 | -6.78 | -143166.250000 (29) | No convergence | -2/3 |
| 3 | 4 | -3 | 6.78 | 71582.117187 (29) | No convergence | -2/3 |
| 1 | -6 | 14 | 4.23 | 3.000000 (2) | 3.000000 (2) | 3 |
| 1 | -6 | 14 | 3.00 | 3.000000 (1) | 3.000000 (1) | 3 |
| 1 | -6 | 14 | -2.12 | 3.000000 (2) | 3.000000 (2) | 3 |
| 2 | 48 | 289 | 0.01 | No convergence | No convergence | -12 |
| 2 | 48 | 289 | -0.001 | No convergence | No convergence | -12 |
| 5 | 0 | -9 | 1.00 | -214748.359375 (15) | No convergence | 0 |

For certain input, the C and ASM versions of gradient descent do not converge or converge incorrectly as the learning parameter **λ** is too high at 0.5. However, when it does converge, the programs converge with fewer iterations with **λ** of 0.5 than with a lower **λ** of 0.1. Interestingly, we obtained a more accurate minimum with a higher learning rate although we would expect the solution to be less accurate with a higher learning rate.

**Implementation**

The values along the x-axis are scaled by a factor of 10,000 in our C program before being passed to the assembly program. The assembly program repeatedly performs a step of gradient descent using a hardcoded learning rate. Unlike the C program, signed integer division is used in our assembly which results in some accuracy loss. The assembly loop ends when the newly calculated x is the same as the previous value.

To count the number of iterations, a pointer to a variable is passed in place of the variable c since c is not used. Just before returning, the number of iterations recorded is stored in the location pointed by the pointer.

Here is the relevent part of our C program.

#define MULTIPLIER 10000

extern int optimize(int xi, int a, int b, int\* counter);

// In the main function

int counter;

printf("ARM ASM & Integer version:\n");

xsoli = optimize((int)(x \* MULTIPLIER), a, b \* MULTIPLIER, &counter);

xsol = (float) xsoli / MULTIPLIER;

printf("xsol : %f, no. of iterations : %d \n\n", xsol, counter);