# EE2024 Assignment 1 Report

## Problem statement

Gradient descent algorithm is a way making use of the property of function gradient to find local minimum value for a given function and we are supposed to write the assembly language function optimize() in the file “optimize.s” to implement the gradient descent algorithm.

## Design methodology

1. Dealing with floating point numbers in assembly language is quite complicated. So casting a floating point number to integer value in C is necessary. However, simple casting may result in loss of precision and therefore we decide to preform scaling for the input of x.

We multiply x by 100 and then convert x into an integer value in C and the reasons for that are 1)we decide that it is enough for the precision to be 0.01 and 2)multiply x by 100 first and then casting will also reduce the loss of precision when perform conversion.

1. For a certain pair of a and b, the optimized solution (-b/(2a)) is not necessary an integer value. So if we scale up the optimized value and then there is better chance for us to get close to the value. For example, when the optimized value is 0.75, the function in assembly language can at best return 0 or 1; but if we scale 0.75 to be 75 and then we scale the result we get from assembly language back in C—we may get 74/100=0.74, which is more precise.
2. For the bonus part, we chose to pass in additional parameter, the address of “count”, to the assembly program. So assembly function will do a load operation from the given address and store the final value back. Then in C we just need to do print out “count” as the value in the address is updated already by assembly function.

Of course the solution described above is not perfect—since we preform scaling, there may be the risk of overflow. Also that due to the limitation of the algorithm, the value of lambda is constrained by the value of “a”, which will be covered in discussion.

## Implementation

First of all, the function signature is altered to fit in the design of bonus part. And also since that the value of “c” does not influence the calculation at all, we decided to drop it in the function declaration.

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

So we are passing in the integer value of “xi” after scaling operation, the value of “a” and “b” and the address of “cnt” which stands for count.

const int FACTOR = 100;

xsoli = optimize((int)(x \* FACTOR),a,(b \* FACTOR),&cnt);

xsol = xsoli;

printf("xsol : %f \n",(float)xsol/100);

printf("number of iterations: %d\n", cnt);

The above part is calling optimize function and printing out the result of optimize. And the following part is our optimize program.

@ This function takes in x, a, b, and address of counter, then give optimization answer:

@ x is a guess value where we start to calculate the optimization answer, and it's kept updating

@ during the whole process. When the function returns x, it will hold the position where there is

@ an optimization answer.

@ a and b is coefficients for quadratic equation (ax^2 + bx + c).

@ address of counter allows us to modify its value.

optimize:

@ R0 = Factor \* x as Integer, AKA scaling the input x

@ R1 = a,

@ R2 = b \* Factor, we also scale the answer (normally answer = -b / (2 \* a) for quadratic equation)

@ R3 = addr of cnt, we access counter's address to modify its value

@ Initialize, eg push registers, LDR etc

PUSH {R4-R6} @ back up other registers R4 ~ R7

LDR R4, [R3] @ load cnt's value into R4

MOV R5, #5 @ R5 = 1 / Lambda

loop:

@ Increment counter by One

ADD R4, #1 @ ++R4 for every loop

@ Calculating d(f) = 2 \* a \* x + 100 \* b

MUL R6, R0, R1 @ R6 = R0 \* R1, which is R6 = a \* x

ADD R6, R6 @ R6 += R6, which is R6 = 2 \* a \* x

ADD R6, R2 @ R6 += R2, which is R6 = d(f) = 2 \* a \* x + 100 \* b

SDIV R6, R5 @ R6 /= R5, which is d(f) \* lambda

@ Comparing d(f) with Zero, if so, go to DONE

CMP R6, #0 @ Compare R6 with zero

BEQ DONE @ if this branch is equal, go to DONE

@ Calculating next x

SUB R0, R6 @ else R0 -= R6, which is R0 = x` = x - d(f) \* lambda

b loop @ while(1), next loop

DONE:

@ Preparing to exit, eg STR, restore registers

STR R4, [R3] @ store R4 back to R3's address, which is \*cnt = R4

POP {R4-R6} @ restore registers R4 ~ R7

BX LR @ return R0, AKA xsoli

The comment inside explains the program clearly we believe.

## Result and discussion

1. The optimize function can produce accurate result considering it is calculating using integer values only. However, even with the scaling, the result is not as accurate as the result in C because of integer division, which will result in loss.
2. The absolute value of input value “x” should not be too large. If so, the value of “x\*FACTOR” may overflow. The same goes for “-2a/b”.
3. The value of lambda is constrained by the value of “a”. For “a”>0,“b”<0, lambda < 1/a.

## Appendix