Neo Yu Yao Terence A0164651E, Neo Jia Wen Rachel A0176763R GA: Ng Yu Wei, Kenneth

Assemble Language Function

```
//Getting the numerator
mul r3, r1, r2        n_mm

add r2, r3, r2
lsl r2, r2, #2

ldr r2, [r0, r2]
cmp r2, #0
beq end4
push {r2}
```

Let $i = m - 1$; i goes from 0 to $(M-1)$ as m goes from 1 to M. The 1st element in row i is hence at position iM relative to the 1st element in the array. The **numerator**, n_{mm}, will hence be at the relative position of $iM + i$, addressed at $\&CM + 4(iM + i)$. To calculate the numerator, we have to first generate the index of the numerator then calculate the address offset required to get the numerator. `Ldr r2, [r0, r2]` loads the value at address `r0+r2`, into `r2`, which is the numerator.

Bonus: `Cmp r2, #0`: subtracts `r2` & `0` and discards the results while updating the conditional flags. If `r2-0` is zero, 'Z' = 1, else 'Z' = 0. An alternative way to do this is to use subs, which would update the flags as well. However, `subs` is used when you want to store the value for later use while `cmp` is used when you don't need to store the value. If 'Z' = 0, branch to end, else continue. There is no need to calculate the denominator if the numerator is 0, as the result will be 0. Hence, this would save time.

```
//Getting the denominator
lsl r3, r3, #0x2
add r1, r3, r1, lsl #0x2

add r1, r0

add r0, r3
ldr r3, [r0], #4
                      ∑_{j=1}^{M} n_mj
loop4:
    cmp r0, r1
    ittt lt
    ldrlt r2, [r0], #4
    addlt r3, r2
    blt loop4
```

The **denominator** is the sum of the row i, given by summing the elements at positions $(iM + 0)$ to $\left(iM + (M-1)\right)$ relative to the 1st element in the array. To calculate the denominator, we have to attain the amount of offset needed. To generate and begin the loop, we first do `cmp r0, r1`: this compares the two addresses by doing `r0-r1` and updates the flags. For this case, we look out for the 'N' flag. `IT` instruction for three `LT` conditions: `LT` checks for the flags 'N' and 'V'. For this case, 'V' would always be zero as there would be no overflow that would occur. Once `r0=r1`, the result from the `cmp` would be equal and the 'N' flag would be 0. For `LT`, it is looking out for 'N'! = 'V'. Once it is not negative, it would not branch into here. An alternative way of doing this is "`ittt MI`" as `MI` only looks at 'N' = 1. Next, we have to load the value at address `r0` into `r2` by post index addressing. It adds `r0` by 4 after it has loaded the value at address `r0` into `r2`. If `r0` < address of the last element of row i, loop again; else if `r0` >= last element of row i, terminate.

```
pop {r2}
ldr r1, =10000
mul r2, r1

//Output
end4:
udiv r0, r2, r3
pop {r3}
BX  LR
```

We have to multiply the numerator by 10000 to compensate for the fractions. This is because the assembly function only deals with integers and would not be able to get a float value out. For example, for 98/100, it would return 0 as the output of the operation since integer division only returns the integer portion. 98/100 is actually 0 remainder 98.

`BX LR`: This means branch to link register, and the link register has to return to call the C calling program.

<u>Extension</u>

By coding with a prudent use of registers, we used 8 registers in our Initial Code. However, that was not good enough for us, so we optimised our code to an impressive 4 registers used now. Furthermore, the registers used was halved without increasing the number of instructions executed.

To do that, we created a visualisation table (Appendix A) that helped us visualise when each register is being used. The grey boxes indicates when the registers are not being used and the white/coloured boxes indicates when the respective registers are being used or when the value in the registers are required for future instructions. 'x' denotes that the value in the register is used for that current instruction. With this visualisation table, we can then do a more efficient allocation of the registers, and can clearly see that 4 registers is the minimum registers possible for our algorithm as the bottleneck is in the loop which requires the simultaneous use of 4 registers.

C Language Function

There are three main methods to pass the array as an argument into a function. All three methods produce similar results as they pass an integer pointer pointing to the first element in the array.

	Method 1: as a pointer	Method 2: as a sized array	Method 3: as an unsized array
Defining	`float f1(int *CM){}`	`float f2(int CM[M][M]{}`	`float f3(int CM[][M]{}`
Call	`f1(CM);`	`f2(CM);`	`f3(CM);`
Accessing Elements	`CM[5]`	`CM[1][2]`	`CM[1][2]`

By passing the array as a pointer, accessing the array can be thought of as accessing a 1 dimensional (1D) array where the original array CM is arranged row wise then column wise, according to the memory addresses of the elements. On the other hand, the 2 dimensionality of the array is preserved when passing the array via methods 2 or 3; array elements can be accessed with double indexing `CM[row number][column number]`. As the idea of accessing `CM` as a 1D array will be covered in the PDM function, for simplicity and readability, we will pass the array as a sized array for the PFA function.

Element Address	1D Indexing	2D Indexing
0x10007fa0	0	0,0
0x10007fa4	1	0,1
0x10007fa8	2	0,2
0x10007fac	3	1,0
…	…	…

Function definition:

```
1  float pfa(int CM[M][M], int i){
2        int k, j;
3        float numer=0, denom=0;
4        for (k=0; k<M; k++){
5              if (k != i){
6                    numer += CM[k][i];
7                    for  (j=0; j<M; j++)
8                          denom += CM[k][j];
9              }
10       }
11       if(denom == 0) return 0;
12       else return numer/denom;
13 }
```

$$PFA_m = \frac{\sum_{k=1,k\neq m}^{M} n_{km}}{\sum_{k=1,k\neq m}^{M} \sum_{j=1}^{M} n_{kj}}$$

The PFA Function can now be coded as a trivial translation of the given equation. Lucky for us, n_{km} refers to the element at row k, column m, in the same order as how we access the element with `CM[k][m]`.

Bonus: A denominator of zero means that the data set only contain elements of actual class m. When that happens, the PFA_m should be zero as there are no elements to be wrongly classified as m. We shall hence add a conditional (line 11) to output 0 instead of the default NaN of the zero division error.

Function call:

```
printf("%f \n", pfa(CM,index));
```

Computation and Storage Efficiency

The order of complexity for the code are as follows:

Time complexity= $O(M^2)$, Space complexity= $O(1)$

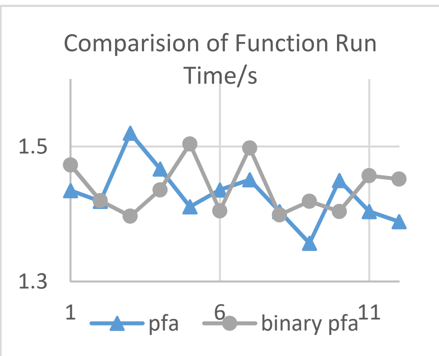
Space complexity is already optimised, but the time complexity has potential for improvement.

Extension: Maybe Better As given $M \leq 10$ is relative small, optimising the time complexity of the code will not have a big effect on run time. However, where's the fun in leaving the code as it is? We shall attempt to optimise the run time at the expense of some storage by using a binary tree summation algorithm as follows.

```
float summation(int *a, int start, int end){
      if(start == end) return a[start];
      else{
            int i = (end - start)/2;
            return summation(a,start,start+i)+summation(a,start+i+1,end);
      }}
```

Time: $O(logM)$

Space: $O(logM)$



We can then replace the 2nd for loop in pfa (lines 7-8) with:

```
int end = M-1;
denom += summation(CM[k],0,end);
```

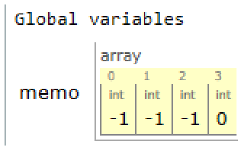
In an ideal parallel machine, the branches of the binary tree can be executed simultaneously resulting in an effective Time complexity= $O(MlogM)$. Unfortunately, the LPC board provided runs instructions sequentially resulting in a linear run time and almost the same average run time over 12 runs for a 10x10 array as illustrated in the figure.

Extension 2: Even Better

Admittedly, recursions are horrible in their high memory consumption, a cost that is not justifiable given that the LPC1769 does not run commands in parallel and will not be able to take advantage of the potential speed improvements of the binary pfa algorithm. We shall take this one step further with a memoized pfa algorithm.

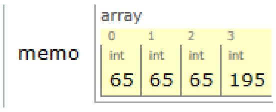
```
1  int memo[M+1];
2  float pfa_m(int CM[M][M], int i){
3          int k, j;
4          float numer=0, denom=0;
5          for (k=0; k<M; k++){
6                  if (k != i)
7                          numer += CM[k][i];
8                  if (memo[k] == -1){
9                          memo[k] = 0;
10                         for  (j=0; j<M; j++)
11                                 memo[k] += CM[k][j];
12                         memo[M] += memo[k];
13                         }
14         }
15         if(numer == 0) return 0;
16         denom = memo[M] - memo[i];
17         if(denom == 0) return 0;
18         else return numer/denom;
19 }
```

We initialise array memo as:



The first time `pfa_m` is called, we will store the sum of row k of array `CM` into `memo[k]` (line 11), and the sum of the entire array into `memo[M]` (line 12). *(Note that `CM[M][M]` has M-1 rows.)*

Subsequently when `pfa_m` is called again as we loop through index in the main function, the row sums of `CM` does not need to be recalculated and can instead be read directly from `memo`, saving significantly on the number or operations executed.



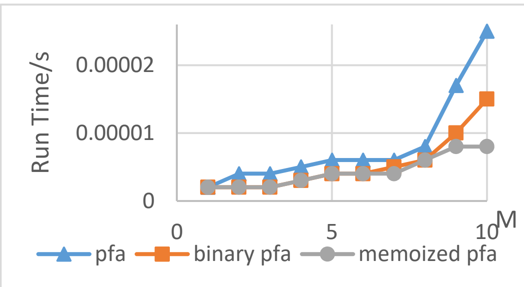
Theoretically this should reduce the run time significantly as the bulk of the arithmetic operations of `pfa` comes from the denominator, which is the sum of all rows except row m. Furthermore, the additional space required for this cache of row sums is only approximately the space a row of `CM` takes and is insignificant.

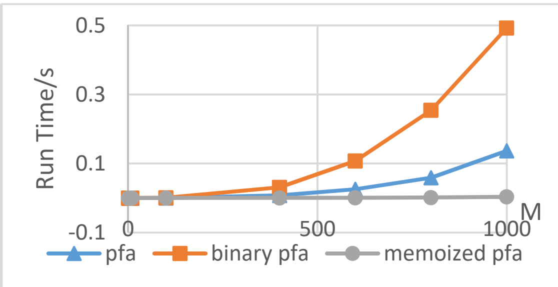
Speed Comparisons and Conclusion

Running the functions on a multi-core computer over 1,000 iterations, we get the following average run times:

```
M = 10
PFA Elapsed: 0.000025 seconds
Binary PFA Elapsed: 0.000015 seconds
Memoized PFA Elapsed: 0.000008 seconds
```

```
M = 1000
PFA Elapsed: 0.136801 seconds
Binary PFA Elapsed: 0.492432 seconds
Memoized PFA Elapsed: 0.003495 seconds
```





Before the number of cores become a bottleneck, `binary pfa` outperforms `pfa`. `memoized pfa` is the fastest, reaching an impressive ~2.5% run time of `pfa` at M=1,000.

Questions

Explain the console window output that you see.

Initial Console Window output:

```
26846.815200
26846.815200
26846.815200

0.000000
0.000000
0.000000
```

The first 3 numbers is the address of the memory location containing the first element of CM. These 3 numbers were previously loaded into R0 in the PDM function which was output as there is no code in the PDM function initially.

The next 3 numbers are the output of **`float pfa()`** which returns null.

Final Console Window output:

```
PDM values:
Class m = 1: 0.923000
Class m = 2: 0.723000
Class m = 3: 0.369200

PFA values:
Class m = 1: 0.292308
Class m = 2: 0.123077
Class m = 3: 0.076923
```

The first M numbers in the console window output are the output values of the PDM function as index increases from 0 to M-1.

These numbers are rounded off to 4 decimal places.

The next M numbers in the console window output are the output values of the PFA function as index increases from 0 to M-1.

What is the address of the memory location containing element (1,1) of the CM matrix? How do you determine this?

Method 1: From C programme

Execute the command `printf("%d\n",&CM);` Output: 268468128

Method 2: From ASM programme

Read the value off register R0 upon entering the ASM programme.

```
Name : r0
    Hex:0x10007fa0
    Decimal:268468128
    Octal:02000077640
    Binary:0b00010000000000000111111110100000
    Default:0x10007FA0
```

Appendix A (Assembly Language Function Visualisation Table)

Initial Code	R0	R1	R2	R3	R4	R5	R6	R10	Comments
Initial	CM	M	i						
`push {r3}`									Store r3 to a stack
`ldr r10, =10000`									
`mul r3, r1, r2`		x	x	iM					Relative addr of i/4, r3 to be used for denom, can't use MLA
`add r4, r3, r2`			iM+i	x	iM+i				Relative addr of numer/4
`lsl r4, r4, #2`			(iM+i)*4		(iM+i)*4				Relative addr of numer
`ldr r4, [r0, r4]`	x		numer		num				**Numerator**
`cmp r4, #0`			x		x				Return 0 if numer=0
`beq end`									Jump to given addr if Z flag is set
`push {r2}`			x						Store r2 o a stack
`lsl r3, r3, #0x2`				4iM					Relative addr of i
`add r1, r3, r1, lsl #0x2`		4iM + 4M		x					Relative addr of past element in i, iM + (M-1) gives last element
`add r1, r0`	x	4iM + 4M + CM							Addr of last element in i
`add r0, r3`	4iM + CM			x					Addr of i
`ldr r5, [r0], #4`	+=4			4iM + CM		4iM + CM			1st element in i, increment r0 to next element
`LOOP:`									
`cmp r0, r1`	x	x							**Comparison**
`ittt lt`									
`ldrlt r6, [r0], #4`	+=4		nxt element in row i				nxt element in row i		**LOOP** needs 4 regs
`addlt r5, r6`				denom		denom			**Denominator**
`blt loop`									If r0 < addr of last element of i, loop again; else if r0 >= last element of i, terminate
`pop {r2}`			numer						Restore r2 from stack
`ldr r10, =10000`		10000						10000	Cannot use #
`mul r4, r10`		x	[CM + 4(iM+i)] * 10000		[CM + 4(iM+i)] * 10000			x	Numer x10000 to compensate for fractions
`END:`									
`udiv r0, r4, r5`	10000 *num/ denom		x	x	x	x			Unsigned division
`pop {r3}`									Restore r3 from stack
`BX LR`									Return to main function