Computer Aided IC Design

Jonas Bertels, Hui Xu

ESAT, KUL
07/12/2020

CONTENTS

INTRODUCTION

THIS document describes the design of our Genetic Algorithm for circuit optimisation.

GENETIC ALGORITHM

The genetic algorithm used was NSGA-II [1]. We refer to the original paper for the implementation details and we will only discuss our implementation details and our stopping criterion.

Sorting

The sorting algorithm is provided by NSGA-II. This algorithm would sort our population into several fronts, and then sort population members in the same front with the goal of optimizing linear distance. (This would later have an effect on our stop criteria as this handled logarithmic functions poorly). Special caution had to be taken when the GA algorithm was used for circuits where 2 identical members of the population with slightly different objective functions (presumably caused by slight numerical errors in spice) would not dominate each other, and therefore create a new front when no new front should be created. (See the section of stop criteria).

Genetic Operation

As [1] was not perfectly clear on the generation of β, we looked through its references and found that [2] explained the generation of β nicely on page 9. After the children were generated the population was simply sorted and only the best were kept, with the amount being kept being equal to the number of parents that we used for every generation. As such, the population size was only used to generate an initial population and was for the most part irrelevant. The number of parents decided how many solutions would be used in the next generation, and returned as result, and the number of children decided how many evaluations would be performed every iteration.The parameters used are given in Table I.
We later understood that normally, the entire population is kept, with the children being added to this population and some sort of selection tournament being applied to determine the parents or the part of the population that is kept. We recognize that this is a much better approach as memory is cheap relative to executions when considering a genetic algorithm, and that a large population would have made both the stopping criterion and the model much easier to implement. Our experience was that overall NSGA-II performed excellently, and that the way it used the sort algorithm to encourage spreading was so good that there was no need to include a minimum spread criterion in the stop criteria.

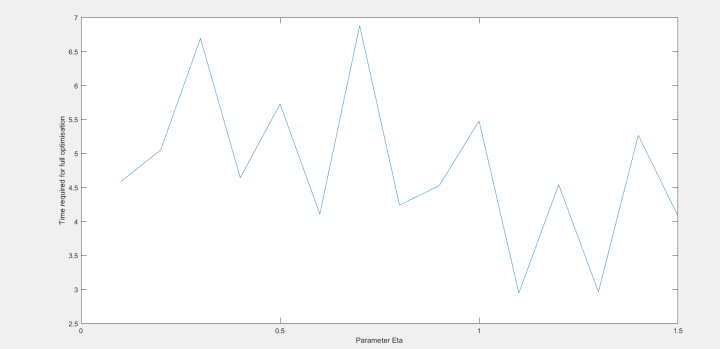

Fig. 1. The computation time of the algorithm in function of η (the graph changed on every run)

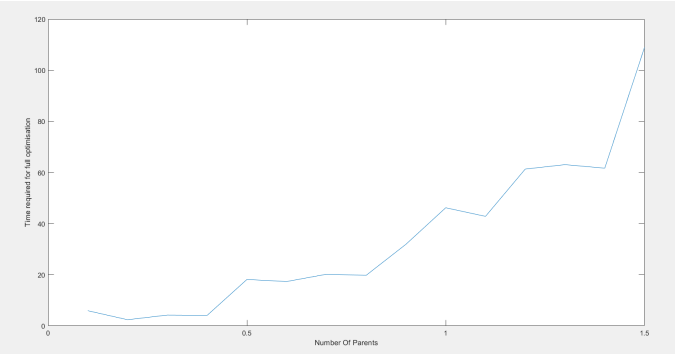

Fig. 2. The computation time of the algorithm in function of the number of parents

Parameter Sweep

We performed a standard parameter sweep over the parameters and found that the fastest optimisation times were achieved by the smallest population size. The best result came with a number of parents and a number of children both equal to 6 (figure 2). However, we wanted to return more than 6 options for the user, and designing working stop criteria became almost impossible due to the shifts along the front. We therefore increased the number of children and parents to 30. Sweeping the parameter η gave rather unclear results, as the values changed every run, so we settled on an η of 0.7 because it seemed to give the best chance of reaching the optimum. (see figure 1 for an example of one of the runs). Lastly, we swept the probability of recombination vs. mutation (see figure 3). It was later realized to do proper parameter sweeps, we should add a significant delay to every function execution to simulate the effect of spice.

TABLE I
DESIGN SPECIFICATIONS

Population Size	30
Number of Children	30
Number of Parents	30
Spread parameter η	0.6
P(recombination)	0.5

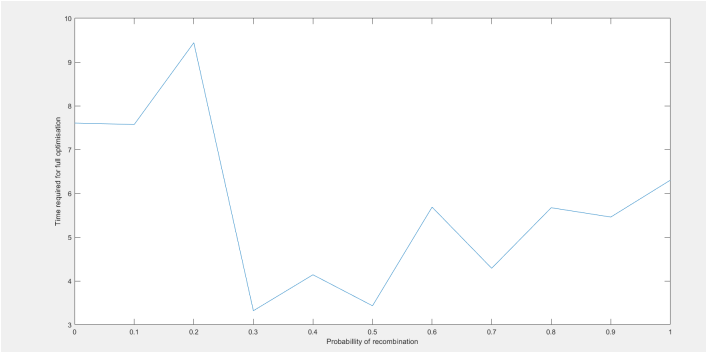

Fig. 3. The computation time of the algorithm in function of the probability of recombination

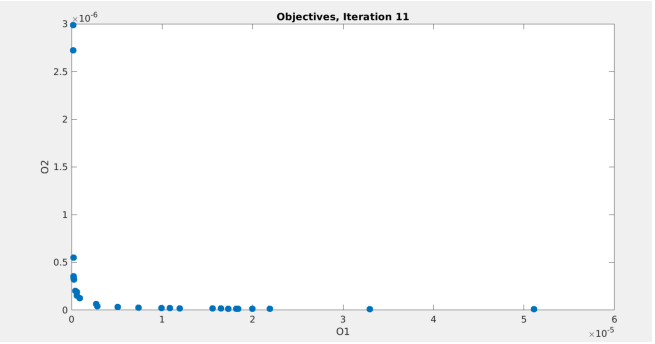

Fig. 4. Pareto curve of circuit (O1: current drain on VDD, O2: inverse GBW)

Stop Criteria

Whereas the algorithm was copied wholesale from [1], no stop criteria were given in the paper. A search through the literature [3] revealed that the following criteria were the most common:

- A set number of generations
- A set number of evaluations
- Stopping when the chance of a new change was exceedingly low

In our algorithm, there was no difference between the first and second item. Our first implementation simply stopped at a 100 cycles. However it is clear that this is a very inflexible stopping criterion. The number of iterations can of course be turned into an input parameter, but this does not provide the ”hands of” genetic algorithm that we were striving for, as the user would now have to guess how many iterations it would take to achieve the desired level of optimisation. Another method that was considered was to run, say, a hundred iterations, then provide the user with the option to continue with another hundred if he is not satisfied with the results yet. However, this requires active user input, and does not fit our scenario of ”run this GA while you go get a coffee”.

It is important to note that many of the issues that occurred with our stopping criterion were due to our oversimplified implementation of the population management. By making the parents the only part of the population that we kept, our stop criterion was much more difficult to estimate than if we had a large population to keep our integrals stable without an additional computational cost.

FrontStop: Our first implementation of the third item simply checked whether the population had belonged to the same front for the last 20/50/100 iterations, and stop if this were the case. (Originally we considered the whole population, but as only the next generation counts in GA, only these should be accounted for)

It was found that the algorithm would stop too soon if only 20 or 50 iterations were checked, but that checking 100 iterations would give good results for the ZDT4 and ZDT6 test sets. However, there were to be problems in the circuit implementation (described later). The FrontStop implementation would work for any number of objectives without any modification, not just the 2 objectives that the specifications called for.

```
integralPop = sortrows(
population(1:NP, V+1:V+M), 1);
for collapse=2:M
    integrals(collapse-1) =
    trapz(
    integralPop(:, 1),
    integralPop(:, collapse));
end
```

Algorithm 1: Integration Algorithm

```
for index=2:M do
    integrals(index-1) = trapz(integralPop(:, 1),
      integralPop(:, index));
end
```

Integration: Our next implementation used the matlab function `trapz` to integrate the second objective function over the first objective function. (To get the integration working properly, it was important to sort the objective functions according to the first objective function so that the same population would give consistent results, as `trapz` does not automatically sort in the X direction before integrating). The difference between the integration result of this generation and the previous generation was then taken and if this was less than 0.01 for a 100 generations, then we would assume that the population had stopped improving. Later, the value 0.01 was changed to a fraction of the total integrated value as the circuit objective functions are not necessarily normalized. (A more advanced version is given by Stop Criteria 1).

The infinite front change problem: When these two stopping schemes were used to stop the optimisation of the circuit, both failed. The reason for this was subtle: it was because our objective functions had a pareto front that looked like figure 4. The values seem to cluster around the center, but as the number of iterations increases, they spread out to the edges (figure **??**), because the sorting algorithm prioritizes linear distance between the objectives. However, seen on a log scale (figure 6), this means that the edges will have many values, while the center will have comparatively few.

The following problem could then occur over 3 generations:

1) A circuitA with $Obj1 = A1$ and $Obj2 = A2$ would be displaced by a circuitB with (for example) $A1 << B1 =$

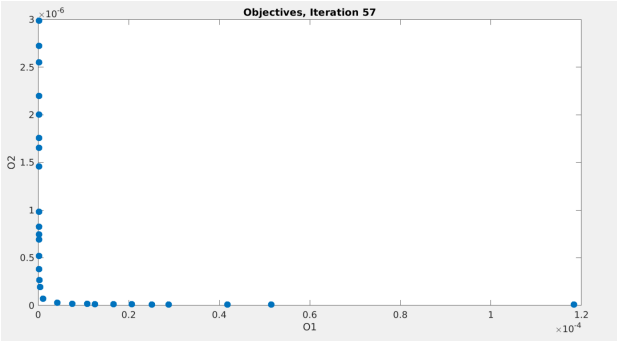

Fig. 5. Pareto curve of circuit after 57 iterations (O1: current drain on VDD, O2: inverse GBW)

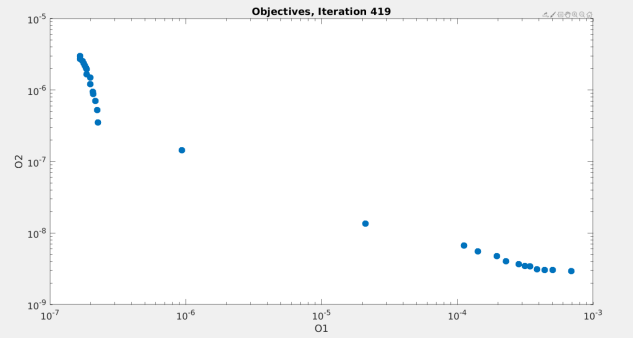

Fig. 6. Pareto Log curve (O1: current drain on VDD, O2: inverse GBW)

$Obj1, Obj2 = B2 < A2$, but with circuitB being further away from the rest of the population and therefore being favored over A as they were technically in the same front.

2) The circuitB could then be displaced by a circuitC with a $C1$ so that $A1 < C1 < B1$ and $B2 < A2 < C2$ but a distance that is also further away from the population than circuitB. CircuitC is straight up worse than circuitA since both objective values for circuitA are less than the objective values for circuitC.
3) CircuitA could then displace circuitC, as circuitA dominates circuitC. This would also trigger a front change and therefore prevent the FrontStop from working. Additionally, because the curve in figure 5 hugs the axes so closely, even a small change in the value at the center would have a large impact on the integration value, making it impossible to tell whether a change was caused by a legitimate improvement or by this process causing fluctuation in the final values.

Improving the integration stop criteria: First we improved the maximum difference between the integration of this generation and the previous generation by scaling it with the size of the population, so that smaller populations would be more forgiving of large changes in integration and vice versa.
Secondly, we improved by implementing the following algorithm: (algorithm Stop Criteria 1)
This algorithm had as great advantage that it was much less affected by shifts in population in the same front as these would result in relatively small changes in the integratedDiff

Algorithm 2: Stop Criteria 1

```
runFlag = True;
while runFlag do
    Genetic Algorithm;
    integratedDiff += (previousIntegral - integral);
    if integratedDiff < integral * (NumberOfChildren)^2/9000) then
        if counter == 100 then
            runFlag = false;
        else
            counter += 1;
        end
    else
        counter = 0;
        integratedDiff = 0;
    end
end
```

and would cancel out before integratedDiff rises above our limit that resets the counter.
However, while this worked well for the benchmark functions, it still didn't stop our circuit optimisation, where the changes in integration were so large that they would exceed our integration limit in one bound so to speak. There was effectively no quantitative difference between the changes of the integration value during optimisation and during the infinite loop that the algorithm would get stuck in at the end. This can be seen just by looking at the curve on figure 4 and figure 5, which hug the Objective axes closely. If the values in the center change by even minimally, it has a large impact on the integration result.

Combining all options: the limit-multiplier: After we realized this last point, the only feasible solution was thus to use stopping criteria that would take into account the number of generations/evaluations that had occurred, but that was flexible enough to work for both circuits and the benchmark function. This was Stop Criteria 2 (given below). This algorithm allows the limitMultiplier to grow exponentially every time the algorithm determines that the integral is decreasing too quickly to stop. Eventually, no matter how large the integral value changes are, the algorithm will stop.

Caveats: Just like with the set number of generations, it is possible to stop too early, especially if one has to go through many fronts and therefore many large integration changes before reaching the optimum. Its large advantage lies in the fact that in situations where the population becomes "stuck" in a non-optimal front, it will not stop until we have gone through 100 iterations without a large integration change. This makes it work well for ZDT4. Secondly, it is resistant to very large integration changes, but it does build up a tolerance if these changes are normal and should therefore be disregarded. Lastly, because we do not take the absolute value of $(previousIntegral - integral)$, small changes back and forth have no impact on the accumulator integratedDiff, and if they are large changes, the limit $integral * (NumberOfChildren)^2/100000) * limitMultiplier$ will in-

Algorithm 3: Stop Criteria 2

```
runFlag = True;
limitMultiplier = 1;
while runFlag do
    Genetic Algorithm;
    integratedDiff += (previousIntegral - integral);
    if integratedDiff < integral * (NumberOfChildren)^2/100000) * limitMultiplier then
        if counter == 100 then
            runFlag = false;
        else
            counter += 1;
        end
    else
        counter = 0;
        integratedDiff = 0;
        limitMultiplier = limitMultiplier * 1.5;
    end
end
```

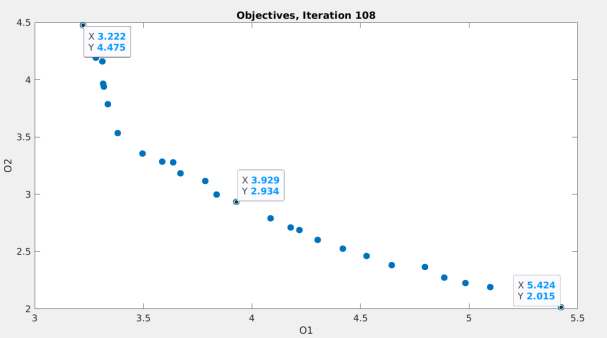

Fig. 7. Simple Amplifier with raw objectives result.

crease until they become smaller than it over time thanks to the limitMultiplier.

Testing and Results

The following functions were tested: ZDT4 (figure 10), ZDT6 (figure 11), a simple amplifier circuit (circuit: figure 12, result shown in log scale on figure 6) and an OTA (circuit: figure 13, result shown on figure 9). Because in the circuits case, we were trying to minimize the power but maximize the Gain Bandwidth, we used the inverse of the GBW as objective function 2. Therefore, to retrieve the achieved specification of any point on 6, use the x-axis to find the DC current drain on the VDD (multiply by 1.8 Volt to find the power) and take the inverse of the y-axis to find the GBW.

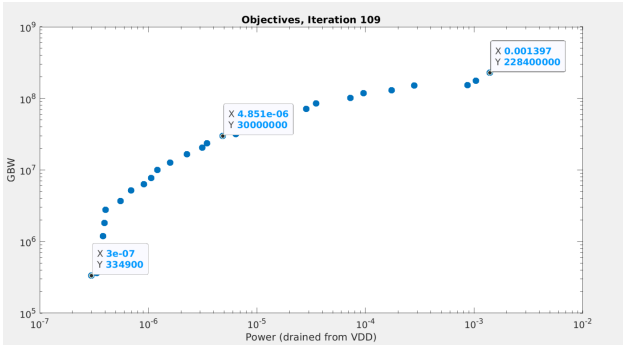

Fig. 8. Simple Amplifier with log objectives result (scaled back to GBW and power in W)

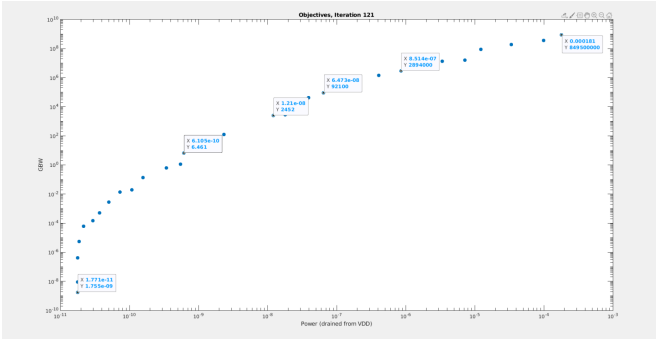

Fig. 9. OTA with log objectives result (scaled back to GBW and power in W) (The reason for some of the extremely low GBW's is that we allowed V_{bias} and $V_{DC,IN}$ to vary, giving the algorithm the option design an amplifier that didn't actually amplify, but was very efficient with power

For the simple amplifier, we only modified the lengths and widths and refrained from changing the bias voltages of the input or the bias voltage to make testing simpler. We did allow the bias voltages and DC input voltage to change for the OTA, but this had as main effect that the algorithm decided to make very power efficient amplifiers that didn't actually amplify. (See figure 9).

The magic stop criterion: know your objective

Looking at figures 6 and 5, we eventually realized that there was also a very, very simple way of making our algorithm converge quickly: taking the logarithm of our GBW and power, and using this as our objective function. (We added 10 to make the results positive, as positive objective functions that decrease if they become better are a requirement for our algorithm). This allowed us to achieve figure 7 in only 108 iterations. (Remembering that it takes a 100 iterations to verify that the function has ceased moving, this means that we effectively found our Pareto front in only 8 iterations!). The final result for the simple amplifier can then be seen in figure 8.

The reason for this immediate improvement comes from the fact that the trade-off between the 2 objectives in question is such that the 2 objectives ($\frac{1}{GBW}$ and Power) multiplied together will approximately equal a constant. Namely, $GBW \sim g_m$, and $P \sim I \sim g_m$ (for constant bias). Additionally, the range over which these objectives is studied is very large and thus we get the shape seen in 4, whereas our algorithm prefers the shape of the ZDT4 and ZDT6 objective functions, which are more similar to functions where the sum of the objectives is a constant. As such applying a logarithm to the objective functions changed the shape of our objective function, allowing the spread algorithm to better spread out

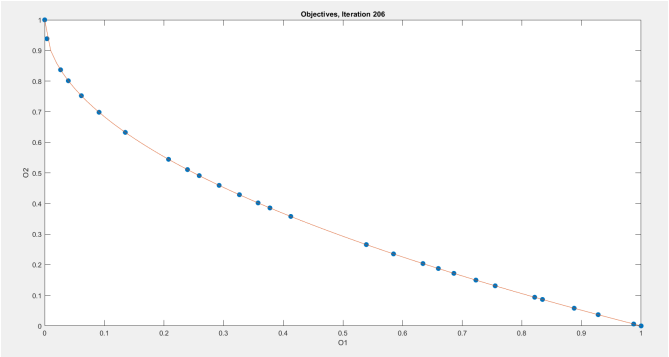

Fig. 10. Result of ZDT4 function

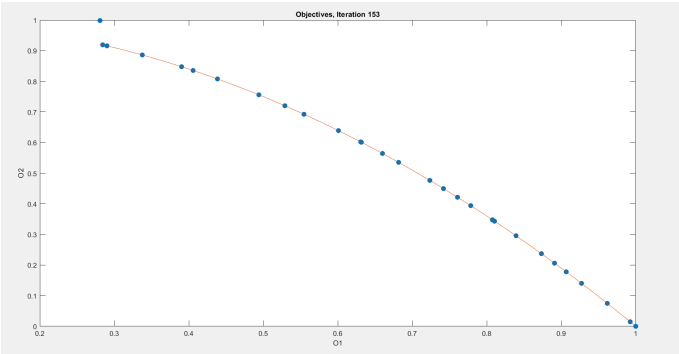

Fig. 11. Result of ZDT6 function

the values in a way that is interesting to the user and allows the stop criterion to better estimate when the optimum has been reached.
Therefore, we recommend that any user of our algorithm takes care to choose his objective function carefully, although this is not strictly necessary. After all, our purpose was to create a ”hands off” algorithm, and therefore the algorithm will stop, even if the objective functions are poorly chosen. (The only specification being that they return positive value that decrease for increasing performance).

GA Speed-up using Response Surface Method(RSM)

First we analyzed which part of our GA consumes the most runtime quantitatively using the profile command.. As shown in the figure 14 (for the OTA in figure 13), the circuit

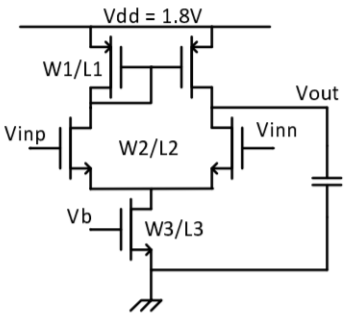

Fig. 12. Simple amplifier circuit

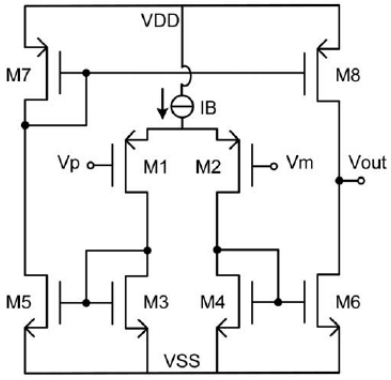

Fig. 13. OTA circuit

Profile Summary
Generated 07-Dec-2020 11:55:52 using performance time.

Function Name	Calls	Total Time	Self Time*	Total Time Plot (dark band = self time)
example_CIRCUIT	1	338.395 s	0.002 s	
myGA	1	338.341 s	2.426 s	
evaluatePopulation	141	329.142 s	0.006 s	
...IT>@(x)interfaceEldo('circuit3',x)	282	329.137 s	0.087 s	
interfaceEldo	282	329.049 s	0.076 s	
geneticOperators	140	326.368 s	0.026 s	

Fig. 14. Result of runtime of each blocks in circuit3

simulation is most time-consuming part, and thus needed to be sped up to make the circuit converge faster.

One of the most popular methods to do so is the Response Surface Method(RSM). Basically, giving the input and output into the proposed model, in our case, it’s a quadratic evaluate function, the result of the model tells you how well this model fits the current data set. If the result gives a good result, which means it is equal or very close to the result that SPICE output, then the result of the model will be used instead of that of SPICE, which saves much time and achieves the speed-up. Otherwise, in case that the result of the model is not accurate enough, the normal run using SPICE will be done.
Based on [4], some models are chosen to be exploited.

We failed to implement a successful RSM, and as such our program would execute slowly. We did note however, that any successful RSM implementation would have to take into consideration that a large quantity of time was spent on a single spice call, and therefore that packing the spice calls into a single large vector is recommended. In fact, if one was to go back to the original parameter sweep, it would be wise to first sweep the spice function with a variety of population sizes to find the execution times, and then to use this information to find the optimal population size. If we had gone ahead with our RSM implementation, it is likely that our small population size would have hurt our effectiveness, with calls to the spice function (but not the analysis itself) taking up most of our time.

Conclusion

In conclusion, we present a working NSGA-II implementation with a well-thought-out stopping criterion. Our algorithm delivers reasonable results, but its execution time could be significantly improved through the use of RSM. However, a

large gain for less work could be achieved by making the population size be the actual size of the population.

REFERENCES

[1] A. Seshadri, "A fast elitist multiobjective genetic algorithm: Nsga-ii," 2000.
[2] M. M. Raghuwanshi and O. Kakde, "Survey on multiobjective evolutionary and real coded genetic algorithms," vol. 11, 01 2004.
[3] M. Safe, J. Carballido, I. Ponzoni, and N. Brignole, "On stopping criteria for genetic algorithms," vol. 3171, 09 2004, pp. 405–413.
[4] Unknown. More curve fitting in matlab. [Online]. Available: https://web.calpoly.edu/m̃dheying/More%20Curve%20Fitting%20in%20MATLAB.pdf