

On Nonlinear Determination of Pareto Fronts

Shantanu Gupta

Internal Report



LEHRSTUHL FÜR ENTWURFSAUTOMATISIERUNG
Fakultät für Elektrotechnik und Informationstechnik
Technische Universität München
Professor Dr.-Ing. Ulf Schlichtmann

Abstract

This report focuses on the techniques to examine the trade-off regions of analog circuits where its competing performances become *simultaneously optimal*, i.e Pareto optimal. Normal Boundary Intersection (NBI) and the Recursive Knee Approach (RKA) are used as advanced multicriteria optimization formulations for the determination of Pareto optimal front. In this work, the implementation of the NBI was extended to three circuit performances from a already given implementation for two performances. Some subtle variations have also been made in the optimization environment from the conventional NBI formulation, the most prominent one being the determination of Pareto front boundary. As an alternate method, RKA was implemented for simultaneous optimization of up to 3 circuit performances. The active constraints preventing the further improvement in the performances were determined and examined for both the implementations. Visualization for three dimensional Pareto fronts was systematically handled for both the implementations that in turn confirmed the well natured approximation of Pareto optimal front generated by the implementation.

Contents

1	Introduction	3
1.1	Analog circuit design and optimization	3
1.2	Pareto Optimality in context of analog design	4
1.3	Solving for Pareto Front	5
1.4	Our approach	5
2	Normal Boundary Intersection - NBI	7
2.1	Introduction to NBI	7
2.2	Variations made to NBI	9
2.2.1	The equality constraint - with and without scalar t	9
2.2.2	Using JumpStart Feature	11
2.2.3	Using Inequality over Equality	12
2.3	3D Pareto Fronts	15
2.3.1	Pareto front using the NBI algorithm	16
2.3.2	Verifying the Pareto front obtained	16
2.3.3	Determining Boundary of 3D Pareto Front	17
2.3.4	Visualization of the front	18
2.4	Determination and Verification of Active Constraint Sensitivities	20
3	Recursive Knee Approach - RKA	21
3.1	Introduction	21
3.1.1	Knee of the curve	21
3.1.2	Recursive knee	21

3.2	Implementation Challenges in 3D	23
3.3	Analysis of Pareto fronts obtained	23
3.4	Visualization	23
3.4.1	For two dimensional fronts	23
3.4.2	For three dimensional fronts	24
4	Comparison - NBI vs. RKA	26
4.1	2D Pareto Fronts	26
4.2	3D Pareto Fronts	27
5	Conclusions	28
6	Further Improvements possible	29
7	Acknowledgements	30
8	Appendix	31
8.1	General Structure of Implementation	31
8.2	WiCkeD	31
8.3	Python	33
8.4	Matlab	35
8.4.1	ParetoCurveLoop	35
8.4.2	NBI Code Related Issues	35
8.4.3	RKA Code Related Issues	37

Chapter 1

Introduction

1.1 Analog circuit design and optimization

Analog components are the most fundamental part of any mixed-signal or analog circuitry. Owing to their non-linear behaviour, they always pose a challenge for automated design. Therefore, in a system design, the analog part is often a bottleneck.

Analog design consists mainly of three steps: topology selection, component sizing, and layout generation. First step towards design of analog circuit is to decide upon the topology capable of meeting given performance specification. In the second step actual values have to be assigned to the circuit parameters, primarily the transistor widths and lengths. Only after this step we can simulate circuit performances - like DC Gain for an operational amplifier.

The usual parameters for analog circuit design like transistor widths and lengths not only have upper and lower bounds on their values but are also constrained by certain implicit conditions termed as *sizing rules* [5]. These sizing rules could be classified in three ways:

- Geometric / Electrical
- Function / Robustness
- Inequality / Equality

These sizing rules also referred to as constraints define a feasible parameter space with acceptable performance values. Beyond this space, circuit may not be trusted to work as intended by the designer. Let us represent parameter vector by \mathbf{p} and all the constraints (sizing rules) in form of a vector inequality constraint $\mathbf{c}(\mathbf{p}) \geq \mathbf{0}$. Consequently, the feasible parameter space \mathcal{P} can be written as

$$\mathcal{P} = \{\mathbf{p} \mid \mathbf{c}(\mathbf{p}) \geq \mathbf{0}\} \quad (1.1)$$

The circuit performances \mathbf{f} (like DC gain, transit frequency, phase margin) are nonlinear functions of designable circuit parameters \mathbf{p} : $\mathbf{f} = \mathbf{f}(\mathbf{p})$. This mapping from parameter space \mathbf{p} to performance space \mathbf{f} can only be evaluated by using circuit simulations. As depicted in Fig. 1.1, the feasible parameter space $\mathcal{P} \subset \mathbb{R}^m$ has an image $\mathcal{F} \subset \mathbb{R}^n$ in the performance space with

$$\mathcal{F} = \{\mathbf{f} | \mathbf{f} = \mathbf{f}(\mathbf{p}) \wedge \mathbf{p} \in \mathcal{P}\}, \quad \mathcal{P} = \{\mathbf{p} | \mathbf{c}(\mathbf{p}) \geq 0\} \quad (1.2)$$

In most cases, there are more parameters than performances, i.e. $m > n$. A designer may wish to optimise these performances simultaneously, however a unique design does not exist where all the performances are simultaneously optimized. We always have a trade-off situation, where it is only possible to improve one performance at the cost of another, which leads to the concept of Pareto optimality [4].

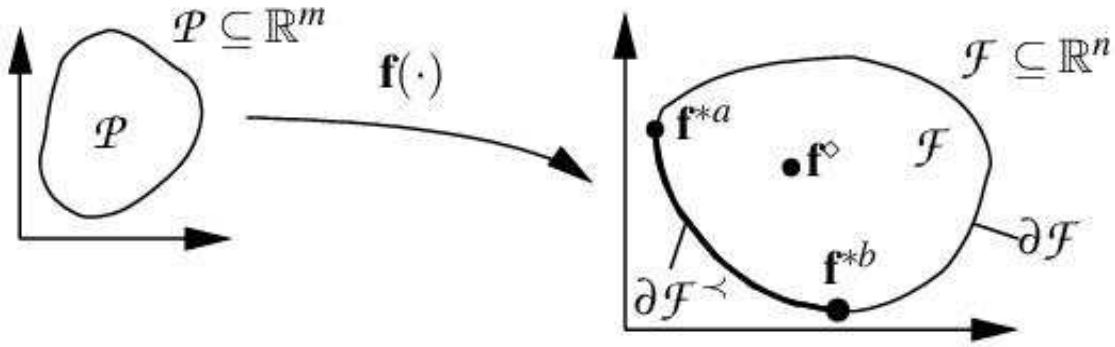


Figure 1.1: Feasible performance space \mathcal{F} with Pareto optimal front $\partial\mathcal{F}^<$

1.2 Pareto Optimality in context of analog design

The concept of Pareto optimality comes into play when we talk about optimization of multiple objectives simultaneously. During multi-objective optimization, we can improve upon one of the objective only on the cost of another. Thus, it is rarely possible to find a set of parameters that optimizes all performances at the same time. We can state the multi-objective optimization problem for circuit performances \mathbf{f} (refer 1.2) as

$$\text{"optimize"} \mathbf{f}(\mathbf{p}) = \begin{bmatrix} f_1(\mathbf{p}) \\ f_2(\mathbf{p}) \\ \vdots \\ f_n(\mathbf{p}) \end{bmatrix} \quad s.t. \quad \mathbf{c}(\mathbf{p}) \geq \mathbf{0}; \quad n \geq 2 \quad (1.3)$$

In remainder of this report, without any loss in generalization, all the optimization would be minimization, since we can always convert a maximization into minimization by multiplying it with (-1).

In multi-objective optimization, a vector $\mathbf{a} = [a_1 a_2 \dots a_n]^T$ is considered more optimal than a vector $\mathbf{b} = [b_1 b_2 \dots b_n]^T$ if it *dominates* \mathbf{b} :

$$\mathbf{a} \prec \mathbf{b} :\Leftrightarrow \forall_{i \in \{1, 2, \dots, n\}} (a_i \leq b_i) \wedge \exists_{i \in \{1, 2, \dots, n\}} (a_i < b_i) \quad (1.4)$$

A vector \mathbf{f}^* is *Pareto optimal* within \mathcal{F} if it is *non-dominated*:

$$\neg \exists_{\mathbf{f} \in \mathcal{F}} \mathbf{f} \prec \mathbf{f}^* \quad (1.5)$$

In Fig. 1.1 point \mathbf{f}^\diamond is dominated, but all the points on the arc between \mathbf{f}^{*a} and \mathbf{f}^{*b} are non-dominated. This portion of the boundary $\partial\mathcal{F}$ denoted as $\partial\mathcal{F}^\prec$ is called *Pareto optimal front*. All the points on the highlighted boundary are non-dominated, and thus Pareto optimal points or *efficient* points. Our goal in this report is to compute trade-off situation between two or three performances efficiently.

1.3 Solving for Pareto Front

The calculation of this trade-off curve is not straightforward in case of analog circuit design. There exists a nonlinear mapping between the feasible parameter space \mathbf{p} (Equation 1.1) and the circuit performances \mathbf{f} (Equation 1.2). The performance values for each given set of parameters can only be obtained by simulation. Hence, it is not practically possible to obtain description of the entire Pareto optimal front by mapping the feasible parameter space to the feasible performance space (owing to the high simulation costs). Instead we have to approximate the Pareto front by finding strategic points over it. These points are usually calculated by transforming multi-objective optimization problem into a single objective optimization formulation, which is then repeatedly solved.

Various methods for calculation of trade-off curves have been reported. Some of the important works focus on efficient determination of the trade-off curves, while others focus on numeric modelling of the Pareto surfaces, rather than on their efficient determination. Simulation based methods have a drawback of not being able to provide information about the limiting sizing rules.

1.4 Our approach

In this report we present two formulations to approximate such Pareto optimal fronts (for up-to three performances) which have the following advantages:

- work in a simulation-based environment for high accuracy

- keep simulation costs low by a special optimization formulation for well-directed searches in the performance space, and
- gains valuable information in form of limiting sizing rules

The first formulation is the much acclaimed Normal Boundary Intersection (NBI) algorithm (this was an extension of an earlier done two dimensional implementation). We also report some very beneficial modifications made to this, most prominent one being the determination of three dimensional Pareto front boundary. The second one is a modification to the weighted sum approach, which recursively solves the problem by employing a divide and conquer paradigm. This one is referred to as Recursive Knee Approach (RKA) [2]. RKA was found to give much better approximation for *knee*¹ of the Pareto curve in case of two performances.

Out of the two approaches, the NBI one is found to be useful when dealing with unknown Pareto surfaces (since it can handle both convex and non-convex Pareto fronts) and also it gives well distributed points in three dimensional cases. Whereas RKA gives smoother approximation of the Pareto curve if it convex and is for two performances. For three dimensions, RKA does not gather a good approximation of the entire Pareto front. This can be explained by its tendency to concentrate primarily on the *knee* of the curve.

A systematic way to handle visualization of the obtained Pareto fronts in three dimensions by using *Delaunay triangulation*, a standard method in computational geometry, for linear interpolation of the surface is also presented. We also extract the active constraints i.e. limiting sizing rules from the optimization runs, and examine them to get a rough idea about performance sensitivities with respect to the same.

¹the point of maximum bulge in the curve, this is found by appropriately choosing objective weights

Chapter 2

Normal Boundary Intersection - NBI

2.1 Introduction to NBI

Normal-boundary intersection formulation is a fast and reliable method for finding evenly distributed points on the Pareto front. The algorithm can be outlined by 4 basic steps (for more details, refer Section 3.3 of [1]), a graphical representation is given by Fig. 2.1:

Individual Minima (f^*)¹ In this step, we optimize the performance individually, i.e. no trade-off is considered, and each of the performances (objectives) are optimized one at a time, ignoring rest all of them. This gives us extreme ends of the Pareto front. The individual minimas are marked as f_1^* and f_n^* in the figure 2.1.

Convex hull of individual minima (CHIM: \mathcal{H}) The performance set where performance f_1 gets minimized is termed as individual minima point for performance f_1 and is represented by f_1^* , having found such individual minimas for all the performances, we can apply NBI normalization to them.

$$\hat{f}_i = \frac{f_i - f_{i,min}}{f_{i,max} - f_{i,min}}, \quad i \in \{1, 2, 3...n\}. \quad (2.1)$$

After this normalization we can combine these points to obtain convex hull of individual minimas CHIM, represented as \mathcal{H} . It is of n-1 dimensions if we have n performances in total.

$$\mathbf{F} = [f_1^* \ f_2^* \dots f_n^*] \quad (2.2)$$

$$\mathbf{w} = [w_1 \ w_2 \dots w_n]^T, \quad w_i \geq 0, \quad \sum_i w_i = 1, \quad i \in 1, 2, \dots n$$

$$\mathcal{H} = \mathbf{F} \cdot \mathbf{w} \quad (2.3)$$

The line joining the two individual minimas in the figure represents the convex hull. Since we have two performances here (f_1, f_2), the convex hull is one dimensional.

¹We always convert all the performance optimizations into minimizations in our implementation, therefore those performance which have to maximized, are multiplied by (-1); thus converting the same into minimization.

Normal vector to convex hull (\mathbf{n}): This is basically the search direction for NBI. We start with evenly spread points on the convex hull (every unique weight vector \mathbf{w} gives a unique point on the convex hull, refer equation 2.2) and move along this normal vector, until and unless we hit some constraints, each such optimization gives us a valid point on the pareto front. The normal vector \mathbf{n} is given by equation 2.4.

$$\mathbf{n} = \mathbf{F}\cdot\mathbf{1}, \quad \mathbf{1} = [11\dots 1]^T \quad (2.4)$$

Here we have a slight variation in the definition of normal vector to the one given by [1], there the normal was directed towards the half containing the origin, and here it is directed towards the other half. We will soon observe the benefits of such a definition for normal vector. The figure illustrates multiple line searches parallel to the normal vector \mathbf{n} .

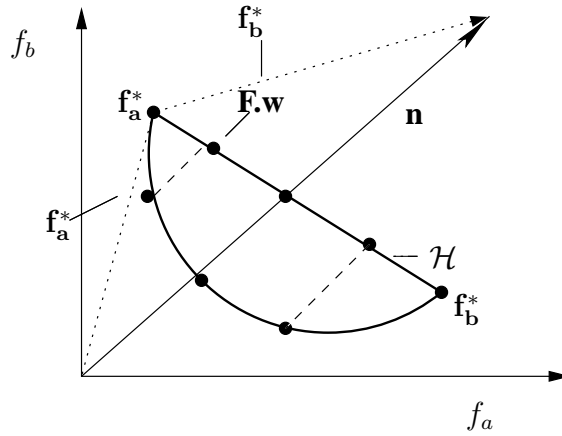


Figure 2.1: NBI with quasi-normal vector \mathbf{n} (not normalized)

NBI optimization problem formulation The NBI formulation combines all the components described above:

$$\begin{aligned} \begin{bmatrix} \mathbf{p}^* \\ t^* \end{bmatrix} &= \underset{\begin{bmatrix} \mathbf{p} \\ t \end{bmatrix}}{\operatorname{argmin}} \quad t \quad \text{s.t.} \quad \mathbf{F}\cdot\mathbf{w} + t\cdot\mathbf{n} = \mathbf{f}(\mathbf{p}) \quad \wedge \quad \mathbf{c}(\mathbf{p}) \geq \mathbf{0}; \\ &w_i \geq 0, \quad \sum w_i = 1, \quad i \in \{1, 2, \dots, n\} \\ &\mathbf{f}^* = \mathbf{f}(\mathbf{p}^*) \end{aligned} \quad (2.5)$$

For any given weight vector \mathbf{w} we have a corresponding point on the convex hull, given by $\mathbf{F}\cdot\mathbf{w}$. On applying rest of the conditions as mentioned in the above optimization formulation, we will get a unique Pareto optimal point \mathbf{f}^* . We have introduced a new scalar parameter t . The geometric idea behind NBI is coded in a vector equality constraint: $\mathbf{F}\cdot\mathbf{w} \in \mathcal{H}$ defines the base point of the quasi-normal vector \mathbf{n} . The left hand side of the equality constraint defines the points along this normal, and t is a measure of distance from \mathcal{H} . Evidently for our case, when dealing with convex Pareto fronts, we will get a

negative value for scalar variable t . $\mathbf{c}(\mathbf{p}) \geq \mathbf{0}$ ensures feasibility of parameter vectors, $\mathbf{f}(\mathbf{p})$ is a feasible performance vector according to equation 1.2. This optimization formulation can be seen as a *line search in the performance space*: the goal is to find most distant point from \mathcal{H} which satisfies the equality and inequality constraints. We choose uniformly distributed points on the convex hull (by choosing appropriate set of weight vector), to get evenly spread points on the actual pareto front. An example output from two dimensional implementation of NBI is shown by figure 2.2.

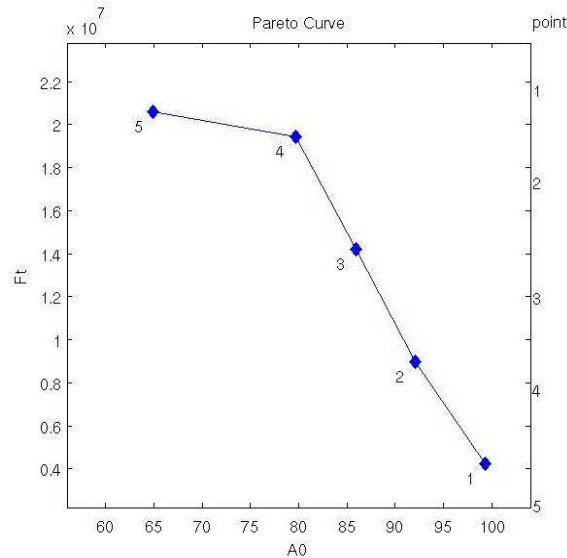


Figure 2.2: Pareto front using NBI for performances A0, Ft from folded cascode

Using NBI optimization formulation we saw how our multiobjective optimization problem gets transformed into a single objective nonlinear optimization problem. We would be using state-of-the-art SQP (Sequential Quadratic Programming) algorithm provided by Matlab's Optimization Toolbox for the computation of this single objective problem (refer Appendix for more implementation details).

2.2 Variations made to NBI

There is a lot of scope in fine tuning the NBI algorithm. We have made few subtle modifications to it, and examined the change in its behaviour. In the following sections we discuss some of the important variations tried with NBI.

2.2.1 The equality constraint - with and without scalar t

In the NBI formulation, apart from the sizing rule constraints (equation 1.1), we have an additional equality constraint as given by equation 2.6 (also shown by NBI formulation in

equation 2.5) for exercising the geometric idea behind the approach:

$$\mathbf{F} \cdot \mathbf{w} + t \cdot \mathbf{n} = \mathbf{f}(\mathbf{p}) \quad (2.6)$$

This additional constraint is backbone for NBI formulation, any changes made to it can influence the efficiency of the whole approach. This constraint forces the optimization algorithm to move in a close proximity to the search ray (the one normal to the convex hull), thereby enabling the optimizer to find the pareto point where the search ray intersects the pareto front. The point (\mathbf{p}_o) where optimization stops successfully can also be understood as a point where the search ray hits the active constraints² ($\mathbf{ac}(\mathbf{p})$) given by

$$\mathbf{ac}(\mathbf{p}_o) = \{\mathbf{ac}(\mathbf{p}) \mid \mathbf{ac}(\mathbf{p}) \in \mathbf{c}(\mathbf{p}) \wedge \mathbf{ac}(\mathbf{p}_o) = \mathbf{0}\} \quad (2.7)$$

The vector equality constraint (eq 2.6) can be represented as a set of linear equality constraints and thus enabling us to eliminate the scalar variable t , let us solve this for the case where we have two circuit performances and derive a simplified formulation for NBI from 2.5

$$\mathbf{F} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}; \quad \mathbf{w} = [w \ (1-w)]^T; \quad \mathbf{n} = [1 \ 1]^T; \quad ; \quad \mathbf{f} = [f_1 \ f_2]^T$$

$$0 \cdot w + 1 \cdot (1-w) + t \cdot 1 = f_1 \quad (2.8)$$

$$1 \cdot w + 0 \cdot (1-w) + t \cdot 1 = f_2 \quad (2.9)$$

$$\min_p f_1 \text{ s.t. } f_1 - f_2 + 2w - 1 = 0 \wedge \mathbf{c}(\mathbf{p}) \geq \mathbf{0}; \quad 0 < w < 1 \quad (2.10)$$

Both the versions of NBI formulation were implemented, and tested for example circuits, namely Folded Cascode and Miller Amplifier. This elimination of scalar variable t does not change the mathematical model of the formulation. It was expected that the implementation 2.10 would be more efficient (since it had one less equality constraint). But there was no such benefit observed when it's efficiency was compared with 2.5. As evident from following results (done for folded cascode with performance tolerance of $5e^{-4}$), the values refer to the number of function evaluations taken up by the optimizer

In the final implementation, we preferred the formulation 2.5 because of following benefits

- the implementation is easy to comprehend since it directly follows from the theory of NBI
- It is generalized for extension, we can reuse the same code for extending the implementation to more number of performances.
- the objective function is much more simpler, and does not require simulations for its evaluation.

²In the actual implementation, all the constraints are to be satisfied with some tolerances, the smaller the tolerance value, more accurate is the solution

Performance Pair	't' eliminated	't' present
A0-Ft	19	28
A0-Gainmarg	33	27
A0-F3dB	21	17
A0-PHM	34	29
PHM-Power	29	27
SlewP-PHM	30	25

Table 2.1: Efficiency comparison for 2.10 and 2.5

2.2.2 Using JumpStart Feature

Ideally as suggested by the optimization formulation, every scalar optimization is started from a default parameter set \mathbf{p}_o ³ and an undefined hessian matrix \mathbf{H}_o ⁴. This could make the algorithm sluggish, given the fact that it has to travel a long distance from the initial point to the Pareto point and has to carry out hessian matrix approximation for every new optimization.

From the given current implementation, at the end of every Pareto point optimization we have a set of circuit parameters \mathbf{p} and an approximation of hessian matrix \mathbf{H} for the scalar optimization done by the optimizer. Using these values as the initial condition for the next optimization run, can give us better results. This flexibility was added to our implementation of NBI and was named as *JumpStart*. This kind of feature cannot be used for individual minima calculation, because there our objective function completely changes, thus the last approximation of hessian matrix becomes obsolete.

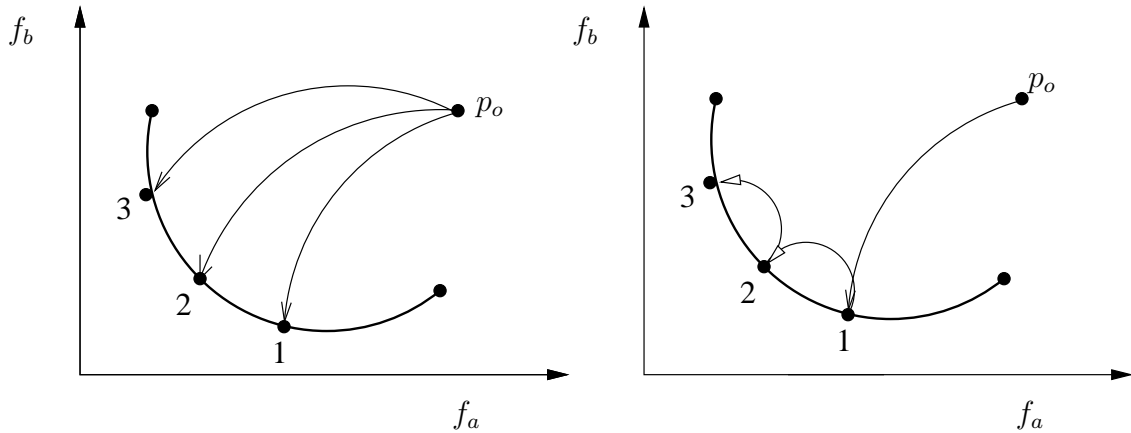


Figure 2.3: The figure shows two optimization runs, (1). Without Jump Start, (2). With Jump Start

But when dealing with intermediate points on the Pareto front, we can start the optimization of a new Pareto points, from the point where the old one stopped, i.e. using the last optimized

³this is taken to be the center of ellipsoid found after applying maximum ellipsoid algorithm on the set of parameters constrained within feasible parameter space

⁴hessian matrix is double derivation matrix of the objective function with respect to all the input parameters

point as a starting point (instead of point \mathbf{p}_o also called as center point). An Ideal *JumpStart* scenario is illustrated by 2.3. The reasons for using *JumpStart* in the final implementation become more clear in the next subsection, where we combine the *JumpStart* with the replacement of equality constraint 2.6 by inequality constraint 2.11.

2.2.3 Using Inequality over Equality

The equality constraint 2.6 forces the optimizer to move along the search ray. We can achieve more efficiency if our algorithm could jump(i.e. using *JumpStart* feature) from one intermediate Pareto point to another, without being constrained in its motion by the equality constraint. In this subsection, we will suggest the remedy to cope up with this situation.

One direct way to achieve this would be to increase the tolerance for the equality constraint, thus making the epsilon region near the search ray as an acceptable area for the optimization. But this kind of remedy would sacrifice our final result, i.e. of computing the accurate Pareto point. Therefore, we make a modification of a kind, which provides higher degree of freedom to our optimization routine without losing out on the Pareto point computed.

We suggest replacement of equality constraint 2.6 by an inequality constraint 2.11

$$\mathbf{F} \cdot \mathbf{w} + t \cdot \mathbf{n} \geq \mathbf{f}(\mathbf{p}) \quad (2.11)$$

On decomposing this vector inequality constraint into scalar inequality constraint, we will obtain as many inequality constraints equal as the number of performances being optimized. Each of these inequality constraint would specify a half plane in the space. Together they will mark out a complete area in the performance space (as shown as shaded region in the figure 2.4), as the value of scalar variable t changes, the inequality constraints also change and thus we get refined shaded area on every iteration of the optimization algorithm.

Inequality constraint gives more degree of freedom to the optimizer, thus helps it in reaching the solution points faster. With the added advantage of finally cornering the same Pareto point as the one obtained by using the equality constraint.

On combining the options of *JumpStart* from the previous section and the geometric constraint from this section, we have following choice

No *JumpStart*, Equality Constraint This combination was the one used by the old implementation, this was giving good results, as far as two dimensional Pareto fronts were concerned, but with three dimensions, and many Pareto points calculations involved, *JumpStart* was an essential feature for the algorithm to work efficiently. Therefore there was a need to try other options.

No *JumpStart*, Inequality Constraint It was expected that inequality owing to it having more degrees of freedom would converge faster to the solution, but such a case is not observed. On an average, inequality constraint without *JumpStart* feature performed

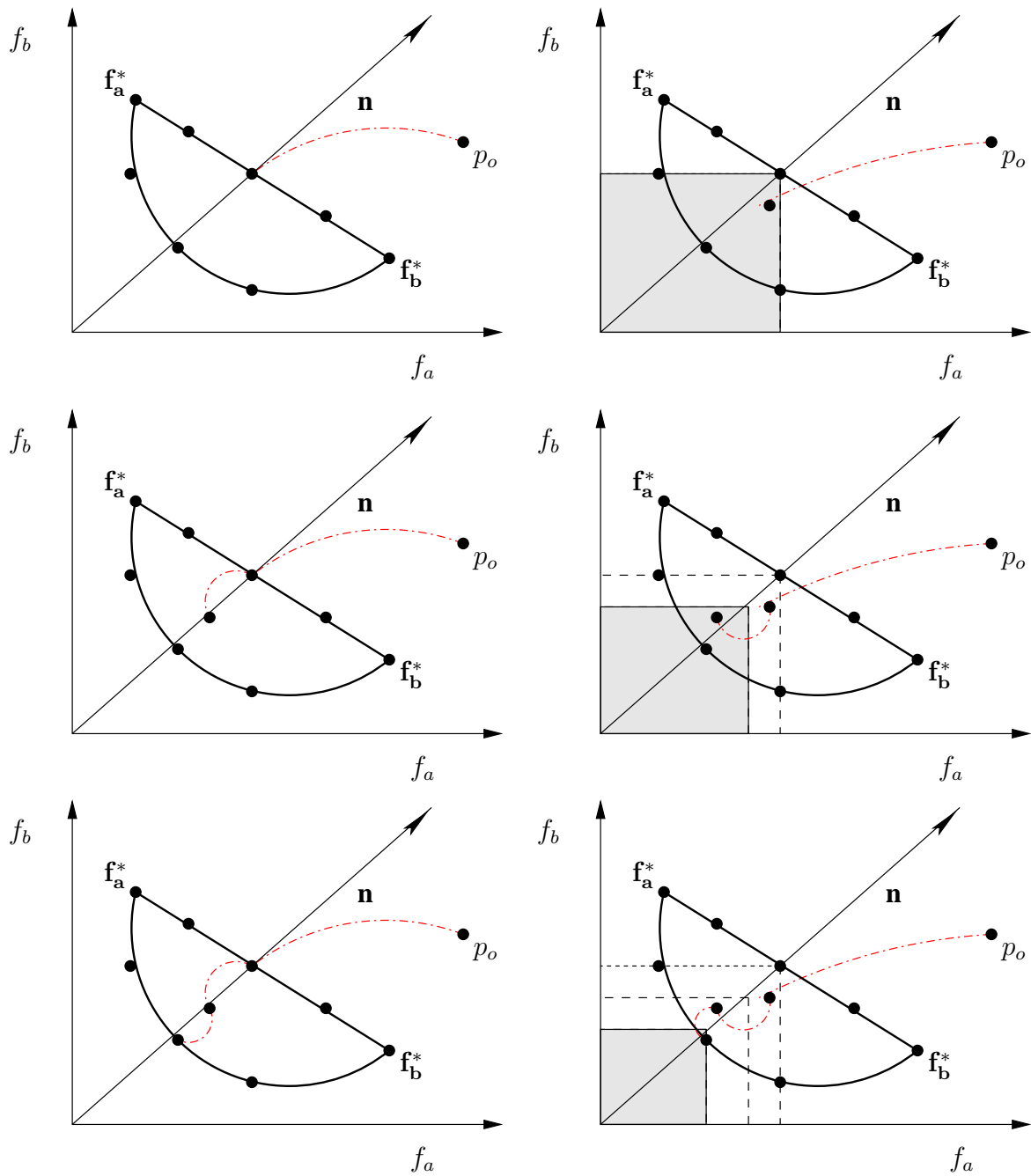


Figure 2.4: Here we see optimization runs for two scenarios: (1). Using equality constraints, (2). Using inequality constraints, the dotted lines show the acceptable area for optimization point, this area gets modified after every iteration of the algorithm.

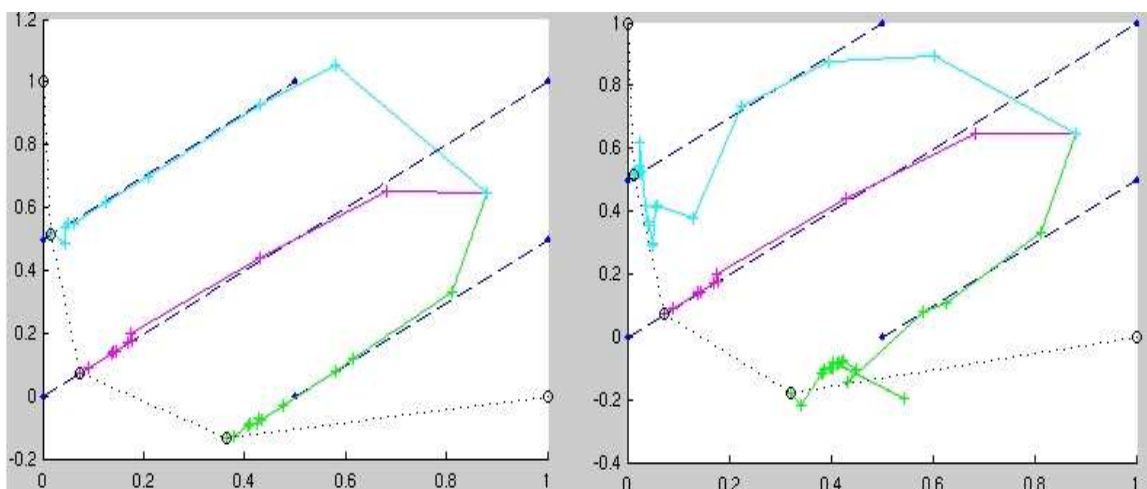


Figure 2.5: Plot of algorithmic iterations (colour coded for every new optimization) for SlewP (x axis) and PHM (y axis) performance pair when using 1) Equality, 2) Inequality; without *JumpStart*

worse than the equality constraint. From the iteration graphs of the algorithm, we observe that a lot many iterations are used up in going towards the allowed region (as defined by the inequalities), while using the inequality constraint. Figure 2.5 shows an example run of the algorithm, where we can notice that inequality constraint makes our algorithm sluggish in attainment of Pareto optimal point.

***JumpStart*, Equality Constraint** While using *JumpStart*, the performance degraded with the equality constraint, this could be attributed to the fact that algorithm in this case tried to reach the search ray (as given by the equality constraint) instead of directly reaching the solution point. Also in a lot of cases, this kind of setup showed premature terminations of the optimizer, thereby giving a poor approximation of the Pareto Front.

***JumpStart*, Inequality Constraint** This came out to be the most efficient option among the four presented for the examples these were tested on. This tackled the problems faced by previous combination efficiently. The good features of this approach are

- More degree of freedom because of inequality constraint
- Easy jumping from last solution to the next solution point by using *JumpStart*. Here the algorithm actually behaves better since there is no need for the optimizer to go after the search ray, thus allowing it to traverse quickly to the solution desired.
- The sluggishness observed in inequality without *JumpStart* goes away since we are starting next optimization from the point where the previous one terminates (thus giving us the benefit of well approximated hessian matrix, and a good initial parameter set)

Figure 2.6 shows results from an example run of the algorithm with *JumpStart* enabled. The comparison is again between equality and inequality form of the geometric constraint i.e. equation 2.6 and equation 2.11.

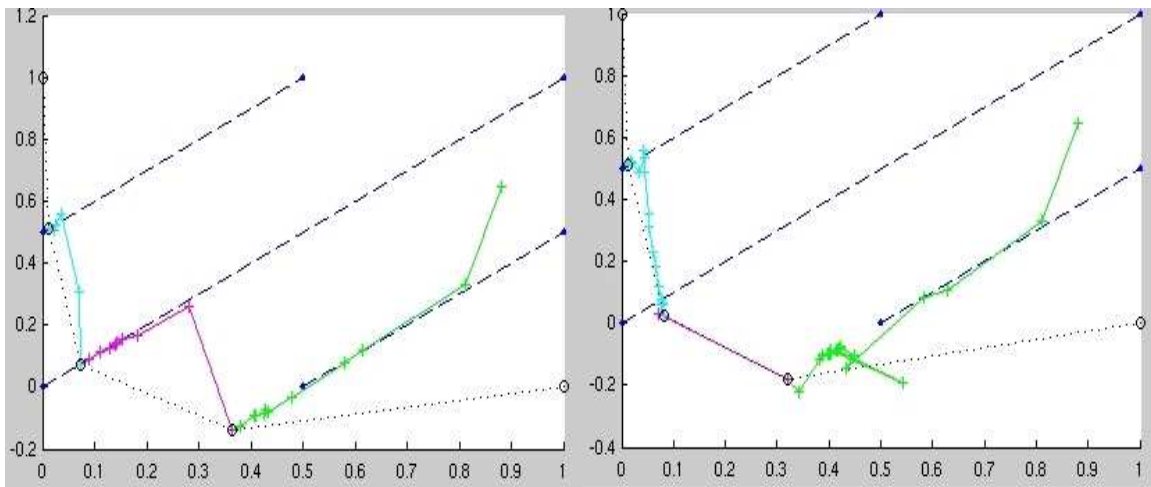


Figure 2.6: Plot of algorithmic iterations (colour coded for every new optimization) for Slewp (x axis) and PHM (y axis) performance pair when using 1) Equality, 2) Inequality; with *JumpStart*

There was a marked improvement in the algorithmic performance when using this approach, the overall number of function evaluations done by the optimizer decreased. At all times, the performance values obtained by the optimization were comparable, or sometimes even better (in the sense they were closer to actual Pareto front) with this approach.

Example: The benefits of this algorithm are even more so evident in case of three dimensional implementation results. There we have many more intermediate Pareto points, and *JumpStart* feature gives great advantages. As an example we tested optimization of *A0*, *Ft* and *PHM* for *folded cascode*, the time taken on a Pentium IV 3 Ghz system with 1GB of RAM was as following:

- **With *JumpStart*** 33 minutes 25.43 seconds
- **Without *JumpStart*** 41 minutes 1.72 seconds

2.3 3D Pareto Fronts

Uptil now, the ideas and implementation we have been talking about was for simultaneous optimization of two performances. In this section we will discuss in detail about three dimensional Pareto fronts.

Lot of ideas which boiled down to simple implementation in two dimensional case, don't apply to three dimensions. As a simple example, we saw how in the case of two dimensions our normal vector \mathbf{n} was simply $[11]^T$, and we also saw that individual minima for one performance becomes the maxima for the other one (this happened due to inherent nature of Pareto points to be nondominated 1.5).

However, such simplifications do not apply for higher dimensions (starting from three). [1] discusses NBI approach in a generalized fashion to handle n dimensional Pareto fronts, the implementation done for three dimensions has also been done along the same lines such that it could later be extended to higher dimensions if the need arises.

2.3.1 Pareto front using the NBI algorithm

The basic steps of the algorithm remain same as they were discussed in the beginning of this chapter. Following are some essential aspects of the approach used for approximation of three dimensional Pareto fronts

- *JumpStart* was used in conjunction with the inequality constraint for better efficiency.
- The relation between number of Pareto points found and the number of points as request by user input could be given by:

$$\text{NumberOfSamples} = \text{input}(\text{input} + 1)/2 \quad (2.12)$$

This kind of computation is done because the number of points found in such a way can symmetrically be distributed over the convex hull. The *input* supplied by user would represent the number of Pareto points on the edge of convex hull.

- The order in which points are chosen for optimization on the convex hull was decided strategically so as to make full use of *JumpStart* feature. This can be understood by figure 2.7 which shows distribution of Pareto points on the convex hull of individual minimas. In this example, user input is 5, and therefore number of samples as computed by equation 2.12 is 15. Suppose we are at optimization of point number 5, then the obvious next point for optimization is 6 (knowing that we are using *JumpStart*; if we are not using *JumpStart* then this special ordering does not have any influence). If instead of 6, we decide to do optimization for point number 9, then the optimizer will take considerably more time. Each such point on the convex hull is given by a weight distribution w_i . A simple algorithm⁵ has been used to find the set of weights $\mathbf{W} = [w_1 w_2 \dots w_{\text{NumberOfSamples}}]$, which give uniformly distributed points on the convex hull, in the order in which they are optimized by the NBI formulation. All this has been designed while keeping in mind the *JumpStart* feature.

2.3.2 Verifying the Pareto front obtained

The Pareto front obtained in three dimensions was verified, so as to ensure correct implementation of the approach. This was done by comparing the results with earlier working implementation of two dimensions.

⁵refer *Points on the convex hull* section in the appendix for details

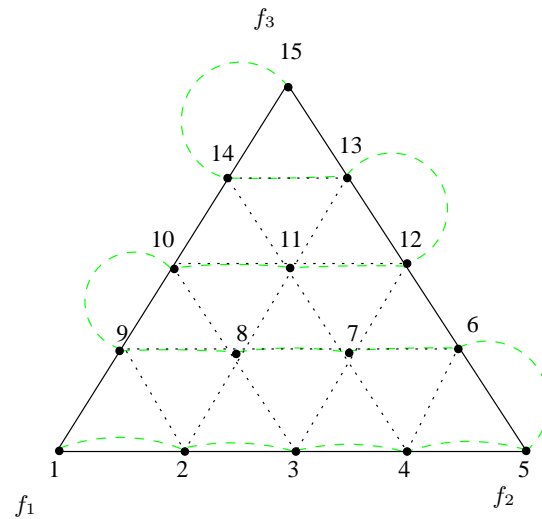


Figure 2.7: Pre-image of Pareto points on the convex hull, and the order in which they are taken up for scalar optimization; point number 1, 5 and 15 represent individual minimas for performances f_1 , f_2 and f_3 respectively

The approach for verification is very simple, we compare the cross section of the three dimensional front (for one of three performances held constant) with the two dimensional one (by specifying the same third performance to be constant). This is possible since the NBI implementation allows performance specifications (like $f_i \geq xyz$) to be handled the same way parameter constraints $c(\mathbf{p}) \geq 0$ were handled.

Example: We have taken folded cascade as the test circuit. The three dimensional implementation was used to draw Pareto front for performances $A0$, Ft and PHM . The cross section of the same was taken at $Ft = 10000000$. This cross section was compared with two dimensional Pareto front for $A0$, PHM obtained from older implementation with an added performance specification $Ft > 10000000$. The result of the same was satisfactory.

2.3.3 Determining Boundary of 3D Pareto Front

The Pareto front obtained using the NBI, is a good approximation of the actual trade-off curve, but it is not complete trade-off curve. The NBI approach finds the Pareto points corresponding to the points taken on the convex hull (these act as base for the search rays). The interpolation of points as obtained by NBI approach gives us front of the kind as shown in figure 2.8. This is Pareto front for a very simple example of sphere, i.e, the three performances are x , y , and z co-ordinates (which are also the parameters), subject to only one constraint - a sphere in three dimensional space. It can be observed from this figure that some of the portions of the Pareto front are missing near the edges.

The solution to this is simple. If we run pairwise optimization for the performances $f_1 f_2 f_3$, i.e. $f_1 f_2$, $f_2 f_3$ and $f_1 f_3$, we will get two dimensional Pareto front for each of them. This gives the

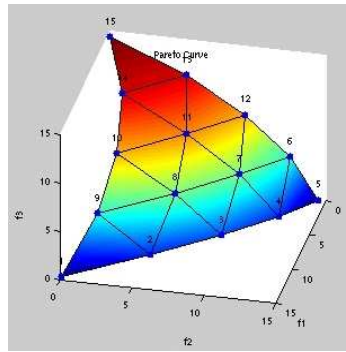


Figure 2.8: NBI Pareto front

trade-off between the two performance values when the third performance is not considered. This is exactly the edge of the Pareto front in three dimensions for the part of Pareto front farthest from the individual minima of the performance ignored in the corresponding two dimensional optimization.

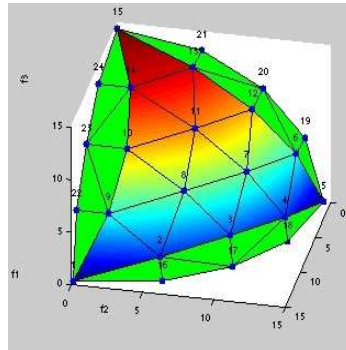


Figure 2.9: Adding boundary to the NBI Pareto front

We can see the results obtained after implementation of approach defined here. The green portions in the figure 2.9 shows the Pareto front boundary computed by running pairwise performance optimizations.

The Pareto front and calculation of its boundary for a real circuit example is shown by figure 2.10. The circuit taken is folded cascode, and the performances chosen for optimization are A_0 , F_t and PHM . All the three performances are being maximized here.

2.3.4 Visualization of the front

Visualization in case of two dimensions is simple, we just have to do linear interpolation of the two extreme points. Since all the Pareto points lie in one plane, they could just be sorted along any one of the coordinates and then later joined to get linear interpolation of the same.

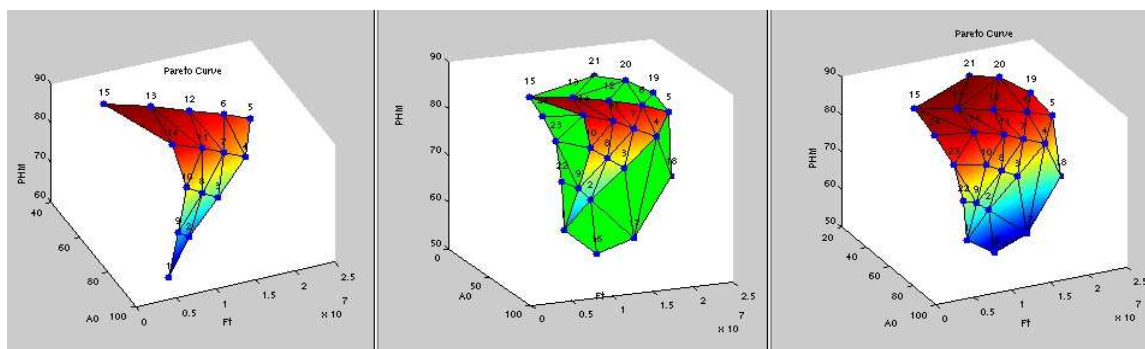


Figure 2.10: Adding boundary to the NBI Pareto front for A0, Ft and PHM performances in folded cascade.

But when dealing with three dimensional fronts, it is difficult problem to decide the order in which they have to be joined and the way the interpolation has to be carried out. Following methods were tried for solving this problem:

- Nonlinear interpolation methods like splines were tried, the results obtained were weird in the sense that they added some unexpected artifacts. Bumps and valleys were created in the Pareto front approximation, which did not follow from the Pareto points used for the interpolation.
- Linear interpolation was taken to be the next alternative, prior to it, there was need to triangularize the set of points. This again was not simple, and thus was handled by a standard technique of Delaunay triangulation⁶. But Delaunay lead to haphazard triangularization again giving us unexpected interpolations.
- Finally, our third attempt used Delaunay in a special way. We took the ideal positions of the points (as on the convex hull), and ran Delaunay on these points. This gave us perfect interpolation neighbours for all the Pareto points; refer figure 2.7 for seeing this triangulation at work. After getting neighbourhood information for all the Pareto points we used it to join the original set of points. This gave us a good visualization of the Pareto front. Also we get a list of triangular planes in three dimensions which can be used as a mathematical description of the Pareto front.

For the visualization of Pareto front boundary, we just did a few more triangulation runs on the new points added, and plotted them alongside the old implementation plot.

⁶refer matlab's help for more information regarding Delaunay

2.4 Determination and Verification of Active Constraint Sensitivities

Determination of active constraint is handled by matlab's optimization engine. The lagrangian multiplier lambda is returned by it, which in theory could be used to evaluate objective sensitivities.

This is exactly what we tried to achieve in our approach. We applied appropriate denormalization steps to the lambda values obtained and then compared the sensitivities obtained with the ones calculated by finite difference method, i.e. recording change in performance value, for a *epsilon* change in constraint.

The result was matching exactly for simple analytical examples, but for real circuits, sensitivity values were not matching. On keeping low tolerance values for constraints and performances, we got close to the actual value of sensitivity. But never got the error below 40%. The order of sensitivity was matching for almost all the cases considered.

There was finally no clear sign which could instill the belief that theoretical calculations apply for the circuits, though they were giving correct results for analytical examples. We at present believe it to be a result of having tolerances and fuzziness in our calculations. For very small values of change in constraint i.e. *epsilon* < 0.007 the sensitivities were not acceptable. May be the tolerances in constraint were nearly of the same order as *epsilon*. But it was very difficult for the optimizer to reach a optimal point for lower tolerance values, so we were not able to confirm our results for lower tolerances. For comparatively large values of *epsilon* > 0.01 the behaviour owing to its non-linearity doesn't show expected performance shift.

Therefore, at present, since we don't have any proofs against the correctness of theoretical calculation, we believe that putting extremely low value of tolerance could give us better approximation for sensitivity. But such a scenario would lead to extremely long running time of algorithm.

Chapter 3

Recursive Knee Approach - RKA

3.1 Introduction

3.1.1 Knee of the curve

The *knee* of the Pareto front is the most important concept here. The *knee* of the Pareto front is the point of maximum bulge on the Pareto front, and can be characterized as the point which is furthest away from the CHIM along a fixed normal direction. The mathematical formulation for finding knee of the curve is given by following equation

$$\begin{aligned} \mathbf{w} &= \text{norm}(\mathbf{n}) \\ \underset{\mathbf{p}}{\text{argmin}} \mathbf{w} \cdot \mathbf{f} \quad \text{s.t.} \quad \mathbf{c}(\mathbf{p}) \geq \mathbf{0} \end{aligned} \quad (3.1)$$

\mathbf{n} is the vector normal to the CHIM, which upon normalization gives us the row weight vector \mathbf{w} . Here \mathbf{f} represents column vector of performances: $[f_1 f_2 \dots f_n]$. This weighted sum minimization goes in the direction perpendicular to the line joining individual minimas. The constraints are given by $\mathbf{c}(\mathbf{p})$. There are no geometrical constraints of the kind we had in NBI, making this algorithm simple and faster for implementation.

If we were to construct a piecewise linear approximation to the Pareto curve using just one Pareto point in addition to the individual minima, the ideal point to use would be the knee point (see figure 3.1). This will be a one step better approximation of the Pareto front, when compared with CHIM. The two pieces of the piecewise linear Pareto curve could also be called as 'refined CHIM'.

3.1.2 Recursive knee

The Recursive Knee Approach (RKA) is based on the simple idea of running the *knee* formulation as discussed by equation 3.1 recursively. That is, after obtaining the knee of the curve,

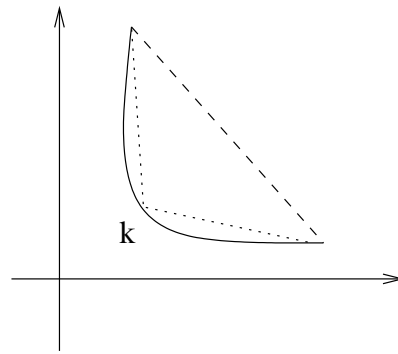


Figure 3.1: Knee of the Pareto curve

and using it to piecewise interpolate the Pareto front, we can run the same optimization again on these refined CHIMs. This will add more points to the piecewise linear approximation, and interpolate the Pareto curve further. This runs recursively until the time when optimal value of the objective function changes by a very small amount. The progression of the algorithm is well illustrated by figure 3.2.

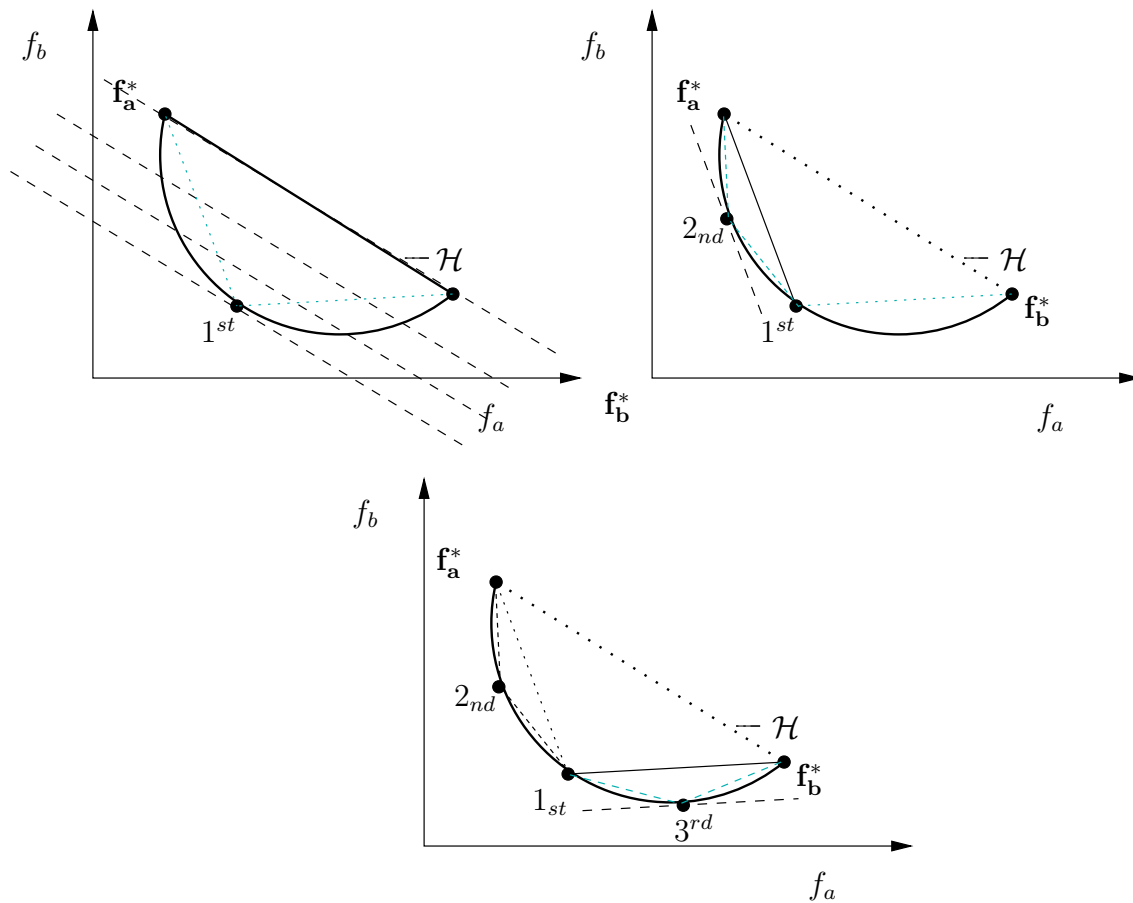


Figure 3.2: Recursive knee in two dimensions

Here by using lesser number of points, we can obtain a finer approximation of the Pareto curve. This is possible because the approach concentrates primarily on the highly nonlinear region of the Pareto front. The only exception where this approach fails is for a nonconvex Pareto curve where the point on the Pareto curve with the worst deviation from the piecewise linear approximation lies above the refined CHIM. In this case our optimization formulation would come up with erroneous results.

3.2 Implementation Challenges in 3D

The implementation of recursive knee for two dimensions was simple. For three dimensions the concept remains the same. However, the implementation gets more involved. Following are some of the important aspects of 3D implementation:

- The initial approximation of Pareto front is a plane in three dimensions. On finding knee of the curve, we divide this plane or CHIM into three new sub-CHIMs for next level of recursion.
- It was not easy to generalize the code for higher dimensions, because it was complicated to generalize the recursive formulation. Hence the current implementation holds only for two and three performances.
- The implementation does not give a good overall approximation of the three dimensional front, this happens because of RKA's tendency to find cluster of Pareto points near knee of the curve.

3.3 Analysis of Pareto fronts obtained

The Pareto fronts obtained using RKA have to be convex in nature for the algorithm to work. The curves obtained using this algorithm have high Pareto point density near the point of maximum bulge, this is advantageous if the designer is interested in knee of the front. Also this way we get an approximation of most nonlinear part of the front with fewest Pareto points.

3.4 Visualization

3.4.1 For two dimensional fronts

The visualization is simple in two dimensions. We use linear interpolation for connecting the points. An example two dimensional front obtained using RKA is shown by figure 3.3.

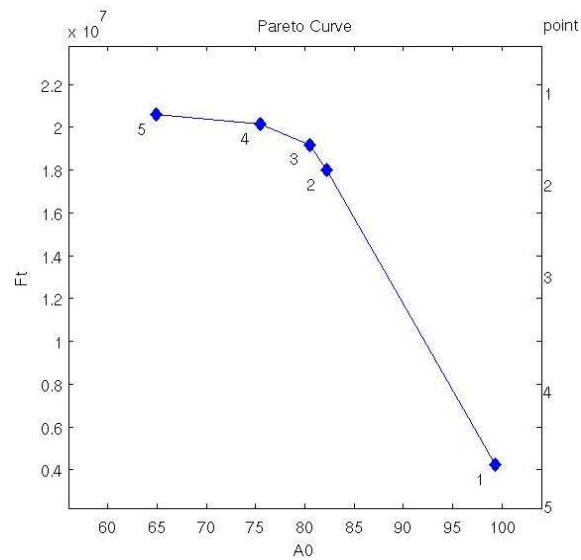


Figure 3.3: Optimizing A0-Ft performance pair using RKA

3.4.2 For three dimensional fronts

Learning from the earlier experiences with cubic interpolations, we directly applied linear interpolation for three dimensional fronts. Here the organization of Pareto points is not so well ordered as we had in the case of NBI. So while computing the Pareto points in the recursion we order them such that our work while interpolating them becomes simple.

At every recursion step, one Pareto point is found. Starting with initial CHIM, we can divide it into three convex hulls using the Pareto point found (see figure 3.4). This way we carry on recursively. We store the points in sets of three, each set of three represents a triangular patch for the Pareto front. Everytime a triangular CHIM gets broken into three sub-CHIMs, we remove the set of three points representing the older CHIM by the sets representing the three new sub-CHIMs.

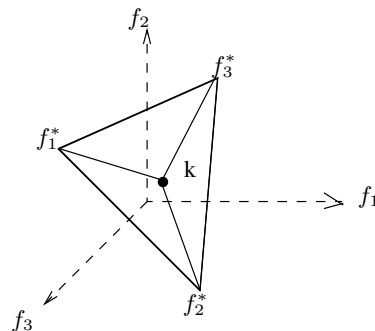


Figure 3.4: Illustration of division of CHIM to give sub-CHIM

An example three dimensional front obtained using RKA is shown by figure 3.5.

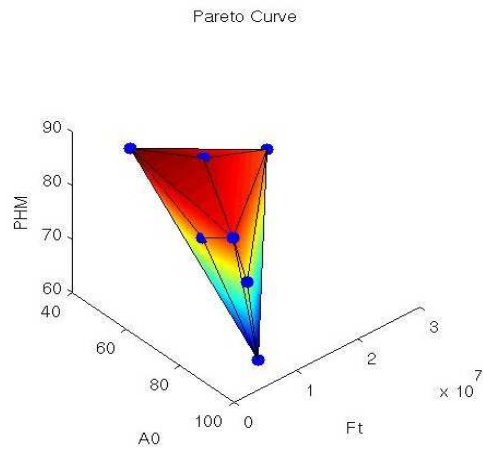


Figure 3.5: Optimizing A0, Ft, and PHM performances using RKA

Chapter 4

Comparison - NBI vs. RKA

Both NBI and RKA are competitive when it comes approximation of Pareto fronts. But both the approaches have their advantages and disadvantages.

The foremost advantage of NBI approach is its ability to work for convex and non convex Pareto fronts. This is not possible for RKA, it works only with convex Pareto fronts. Further the points found by NBI are evenly distributed on the Pareto front, this could be taken both as an advantage and disadvantage depending on designers need. If he wishes to find good approximation for complete Pareto front, then this kind of even distribution is helpful. But sometimes in its attempt to distribute the points evenly, NBI does not pay much attention to the areas of maximum bulge or nonlinearity in the Pareto front. This problem is solved by RKA since it finds only the Pareto points in the more nonlinear parts of the Pareto front. Thus helping us in getting approximation of most nonlinear part with less number of points. We can comment some more upon these two approaches by talking in the context of two and three dimensional Pareto fronts.

Another use of RKA could be in cases where we are finding only one Pareto point, and we want it to be the point of best trade-off (i.e. the *knee*). In such cases RKA can give us solution in one shot, while NBI would take a lot of Pareto point computations to reach such a point.

4.1 2D Pareto Fronts

For convex two dimensional Pareto fronts RKA is always observed to be a good choice. This is so, because we want faster approximation of nonlinear parts of the Pareto fronts where there is actual trade-off to observe. The NBI approach would have to compute a lot of Pareto points to come close to the nonlinear approximation as given by RKA. We can observe this better in figure 4.1.

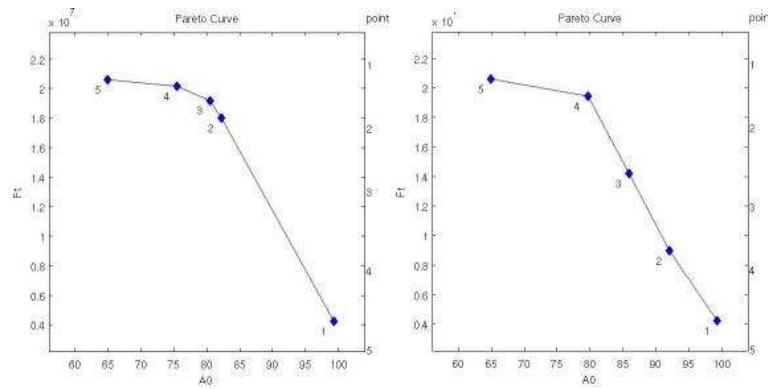


Figure 4.1: Comparing two dimensional output from RKA and NBI

4.2 3D Pareto Fronts

In three dimensions, the obtainment of Pareto points and their interpolation both are not easy. A lot of efforts were needed for a good interpolation of three dimensional Pareto fronts. The results obtained for three dimensional fronts using NBI were highly satisfactory, especially after supplementing them with approximation of Pareto front boundary. While in case of RKA we got a good approximation of the region near the point of maximum bulge.

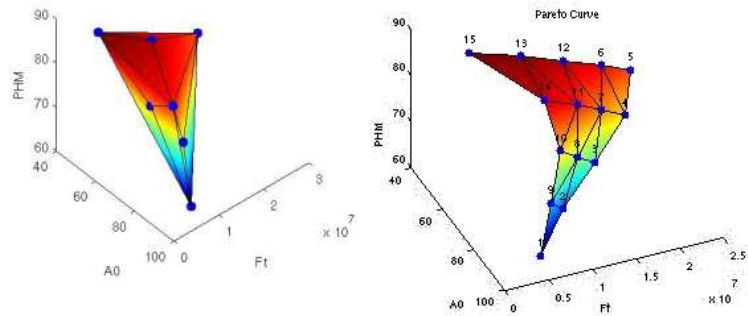


Figure 4.2: Comparing three dimensional output from RKA and NBI

Chapter 5

Conclusions

In this report we presented approaches for exploring Pareto optimal regions of a circuit performance space. We have used normal boundary intersection and recursive knee approach as our optimization formulation for solving the multi-criteria objective problem.

Both the techniques were implemented for handling two or three circuit performances. We identified and analyzed the active constraints observed in the optimization. The boundary of the Pareto front was found in case of three circuit performances by using NBI method for pairwise optimization. Few modifications were also applied to NBI to make the implementation more efficient.

The implementation was tested with two circuit examples provided: folded cascode, and miller amplifier. The resultant Pareto fronts gave the ultimate performance capabilities of a given circuit topology at full transistor-level accuracy. Visualization of the results helps us to examine the trade-offs between competing performances. The results from these implementation could be considered for hierarchical optimization.

Chapter 6

Further Improvements possible

The current approach has scope for improvement in following areas:

Theoretical Some of the possible theoretical improvements

Combining approaches One can try combining the Pareto points obtained using both the approaches. This will give a good information about the *knee* of the curve (due to RKA) as well as rest of the front (due to NBI).

Performance sensitivities with respect to constraints This is another aspect which has to be looked at in detail. Theoretically, we should have obtained exact values of performance sensitivities with respect to constraints, but in practice we got heavy deviations.

Implementation based Some of the possible implementational improvements

Inter Process Communication (IPC) The current implementation uses files as a medium of communication between different processes, i.e. matlab, wicked, and python modules. This wastes a lot of time in I/O activities. An implementation based on the lines of IPC could be expected to be more efficient.

Interpolation The interpolation used for the visualization of three dimensional Pareto fronts is linear in our case. One could try some kind of cubic interpolation for obtaining smoother Pareto fronts.

Chapter 7

Acknowledgements

I am deeply indebted to my guide Dipl-Ing Guido Stehr for his constant help and support. His guidance helped me in exploring the subject field far and wide, and also helped me to direct my energy in the right direction. I would also like to thank Dipl-Ing Daniel Mueller for helping me in preparing my presentation and documentation for the work done. And finally, I would like to thank Dr. Helmut Graeb for giving me this golden opportunity of working in this wonderful environment at TU-Munich.

Chapter 8

Appendix

8.1 General Structure of Implementation

The implementation of the approaches discussed in the previous section is handled by three important components

- WiCkeD Worst Case Distance Analyzer is a multifaceted tool for electronic design automation. This has been entirely developed at TU Munich.
- Python programming languages: It was used primarily as an interfacing language to the WiCkeD.
- MATLAB: This was used for its highly robust and efficient optimization functions, and its ability to handle matrix operations.

It is difficult to start off with explanation of each of their roles, therefore we present a diagrammatic overview of the whole implementation in figure 8.1. Three communication pipes are used for handshake between python and matlab, namely StartMatlabPipe, CmdPipe, and OKPipe.

8.2 WiCkeD

In this implementation, WiCkeD has been used to do circuit simulation to get performance values for a given parameter set, and it also takes care of finding sensitivity matrices for performances and constraints (sizing rules) defined for the circuit. WiCkeD is handled by class *WickedInterface.py* written in python.

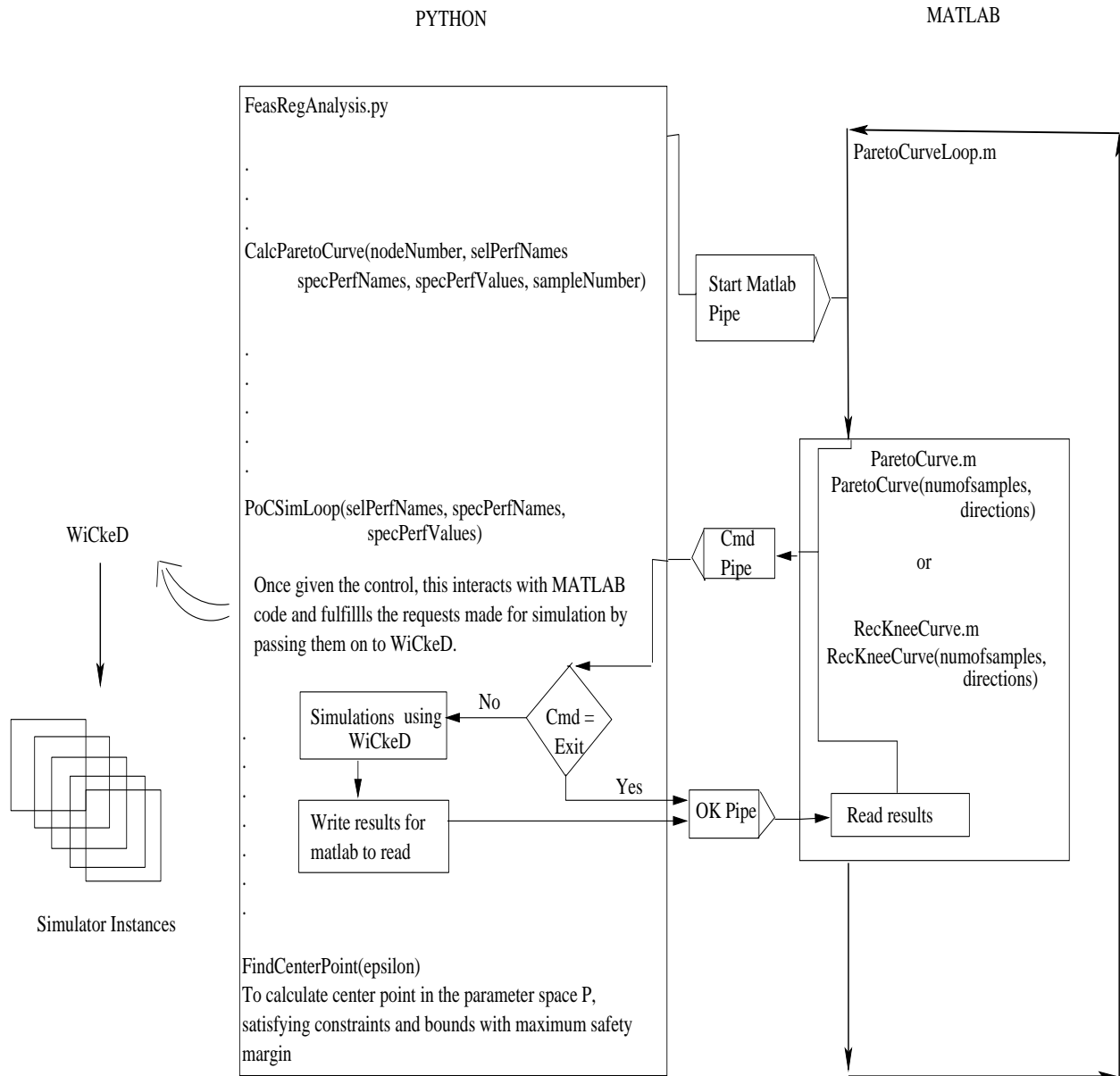


Figure 8.1: Implementation Overview

8.3 Python

Python as explained earlier is used for interfacing. The important scripts in python are

WickedInterface.py This can help us to basic operation in WiCkeD such as appending and deleting nodes, running simulations, extracting sensitivity values, name and number of parameter, performances etc.

FeasRegAnalysis.py This is the most important python script. It contains all of the important functions for our implementation. The file defines *Feasibility Region* as its base class. It has instance of *WickedInterface.py* defined within it for communicating with WiCkeD. This also has *matlabHandle* named variable for handling the instance of matlab opened within python script using *pymat* (a package for integrating python and matlab). Following are some of the essential functions within this module:

FindCenterPoint(epsilon) This function is useful for finding point in the parameter space which has maximum safety margins for parameter bounds and constraints. This uses maximum volume ellipsoid method iteratively to converge at the required center point. The epsilon values is for tolerance of the value found. The usual value for epsilon is 0.05.

CalcParetoCurve(..params..) The job of this function is to call matlab script for optimization after extracting the relevant information from the parameters provided. The parameters for this function are *nodeNumber*, *selPerfNames*, *specPerfNames*, *specValues*, *sampleNumber*, *boundary*. *nodeNumber* is the node we start our optimization from, therefore parameter set from this node shall be our initial parameter set. It is recommended to have *nodeNumber* represent center node. *selPerfNames* is a list for performances selected for optimization, this can have a length of two or three. *specPerfNames* represents names of function which are specified to fulfill certain conditions, apart from the usual constraints $\mathbf{c}(\mathbf{p})$. The values specified for them is listed by corresponding entries in *specValues*. The *sampleNumber* and *boundary* have special interpretation dependent upon the optimization formulation.

- **NBI** Here the *sampleNumber* represents number of Pareto points to be calculated between any pair of performances given in *selPerfNames* (including the individual minimas). For two dimensional case this will be total number of Pareto points computed, but in three dimensions, if we have **div** number of points between any two performance individual minimas, then the total number of points on the Pareto front is given by

$$\text{numberOfPoints} = \text{div} * (\text{div} + 1) / 2 \quad (8.1)$$

The *boundary* variable is for internal use. This has a default value of 0, and is used only by *CalcParetoCurveWB()* function.

- **RKA** Here we have altogether different meaning for *sampleNumber*. It is used as a tolerance value for stopping our recursion while finding Pareto points. If the difference between the initial and final objective function value for any recursion thread gets smaller than the tolerance (given by *sampleNumber*), then that recursion thread ends. If nothing is given, *sampleNumber* defaults to zero. We have another way of stopping the recursion, i.e. by setting the maximum recursion depth variable in the matlab code. The *boundary* variable does not have any significance for RKA.

The first task of this function is to extract parameter values, upper and lower bounds etc from the node number provided and store them in a matlab file named *init.mat* after doing 1st normalization i.e. *center point normalization*. It is done as following

$$p_{normalized} = (p - p_o)/p_o; f_{normalized} = (f - f_o)/f_o; c_{normalized} = c/c_o \quad (8.2)$$

Then this function triggers the *StartMatlabPipe*, by writing to it *directions* of optimization and *sample number* requested. Thereafterwards this function calls *PcC-SimLoop*, and waits for it to finish. The result are collected thereafterwards and treated with denormalization for the center point normalization. And finally the results are written to a matlab file, (which is used later for plotting the graphs).

PoCSimLoop *PoCSimLoop* is another function defined within *FeasRegAnalysis.py*, and its task is to receive commands from Matlab through *CmdPipe* and interpret the command supplied. Thereafterwards it uses *WickedInterface* object to do the task requested by matlab, writes the results to file named *results.mat* and then triggers the Matlab to read the results by writing into the *OKpipe*. After receiving *Exit* command from matlab, this function returns back to *CalcParetoCurve()*.

CalcParetoCurveWB(...) The parameters for this are same as the ones we had in *CalcParetoCurve()*. The idea behind this function is to prepare a python wrap up to the task of finding Pareto front boundary. This function calls the 3D optimization for three performances, and then pairwise calls f_1f_2, f_2f_3 and f_3f_1 .

When *CalcParetoCurve()* is called internally by *CalcParetoCurveWB()*, then the value of variable *boundary* would be 1 (for pairwise optimizations). In such a case one another routine at the end of *CalcParetoCurve()* does the job of appending the new boundary data acquired to an already existing Pareto graph file. (obviously representing a 3D Pareto front).

PlotParetoCurves(path) This function does the job of calling matlab's *ParetoPlot* function or *ParetoPlotKnee* function depending upon the matlab file passed. The matlab file have a identifying flag *algoType*, whose value decides the type of algorithm used for making that file, 0 indicates NBI, 1 indicates RKA. So the corresponding function is called after checking *algoType* variable in this function.

pythonstartup.py This file is a small script to make the instance of the base class, i.e. *Feasibility Region*. Before that it has to mention the center node number, for starting it could

be mentioned as any existing node in WiCkeD interface, but in precise terms center node implies - a node with parameter set as found by *FindCenterPoint()* method from *Feasibility Region*.

8.4 Matlab

Matlab is of crucial importance in our implementation. We have two sessions of Matlab working simultaneously during the working of our algorithm. The first one is opened within *python* script *FeasRegAnalysis.py*, by using *pymat*, an interfacing package for python and matlab. This is essential for enabling python to store variables in a format such that second Matlab session (one running the actual optimization formulation) can read the results gathered by the former. Basically we can divide the matlab code in two sections, one for handling NBI implementation, and the other for RKA. But before that we must talk a little about a script which handles the triggers from Python environment.

8.4.1 ParetoCurveLoop

This loop is for catching if anything has been written down in *StartMatlabPipe*. Whenever anything is written down, it reads the values, decides upon which algorithm is suitable to call (NBI:ParetoCurve.m, or RKA:RecKneeCurve.m) and then waits for them to finish. Then it After which it gets ready again to be triggered by Python for another optimization. Thus it goes in loops.

8.4.2 NBI Code Related Issues

NBI is implemented by *ParetoCurve.m* matlab script. The code has following important functional modules

ParetoCurve(sampleNumber, directions) This is the backbone which handles complete NBI algorithm. This file does following jobs in the order stated

- loads *init.mat* written by CalcParetoCurve()
- Finds individual minimas
- Makes extreme values matrix *fExtr*
- Does NBI normalization to get *Fnbi*
- Calls CalcPointsNBI for handling intermediate Pareto points.
- Denormalizes the complete set of performance points obtained
- Saves them in a matlab file.

- Returns back to ParetoCurveLoop()

CalcPointsNBI() This implements the NBI algorithm formulation. Main steps are

- Calculates weights for an even spreading on the convex hull
- Append variable t for optimization formulation.
- For each weight value \mathbf{w} , it runs the optimization, and then stores the performance points and corresponding parameter set.

objfun (1st objective function) This is for individual minima computation

objfunt (2nd objective function) This is for actual NBI objective function, i.e. equation 2.5.

nonlinconstr - gives the set of constraints The original constraints of the circuit are added on to by the inequality constraint 2.11. The matlab convention requires us to multiply all the constraints by (-1). This function also has the responsibility of finding sensitivity matrix for all the constraints with respect to all the parameters. We have added inequality constraints, and a parameter t to the initial set of constraints and parameters respectively. Therefore we have to generate complete sensitivity matrix for them carefully. Figure 8.2 explains this, C_i is for scalar components of vector inequality constraint 2.11.

	c1	c2	c3	...	cn	C1	C2	CONSTRAINTS
p1	g (wicked finds this)					$-\frac{df1}{dp1}$	$-\frac{df2}{dp1}$	Obtained by differentiating inequality constraints C_i by parameter p1, p2...etc
.						.	.	
.						.	.	
.						.	.	
.						.	.	
pn						.	.	
t	0	0	0	...	0	n1	n2	PARAMS

Obtained by differentiating inequality constraint C_i with respect to t. n1 and n2 are components of normal vector n.

Figure 8.2: Explaining the sensitivity matrix in nonlinconstr

8.4.3 RKA Code Related Issues

All the structure remains same, we replace `CalcPointsNBI()` with `CalcPointsRecKnee()`. The variable *rec-depth* defines maximum depth allowed for recursion.

There are no extra inequality constraints, or any extra parameters introduced here, thus making this approach much more simpler to understand and implement.

Bibliography

- [1] G. Stehr, H. Graeb and K. Antreich, *Performance Trade-off Analysis of Analog Circuits By Normal-Boundary Intersection*, DAC 2003, June 2-6, 2003, Anaheim California, U.S.A.
- [2] I. Das, *On characterizing the 'knee' of the Pareto Curve based on Normal-Boundary Intersection*, Mobil Strategic Research Center, Dallas, TX, U.S.A.
- [3] I. Das, *An Improved Technique for Choosing Parameters for Pareto Surface Generation Using Normal-Boundary Intersection*, Mobil Strategic Research Center, Dallas, Texas, U.S.A.
- [4] I. Das and J. E. Dennis. *Normal-boundary intersection: A new method for generating the Pareto surface in nonlinear multicriteria optimization problems*. SIAM Journal on Optimization, 8(3):631-657, Aug. 1998.
- [5] H. Graeb, S. Zizala, J. Echmueller, K. Antreich. *The Sizing Rules Method for Analog Intergrated Circuit Design*. IEEE/ACM ICCAD 2001 San Jose California Nov 4-8, 2001.