A SWITCHED CAPACITOR FILTER
SYNTHESIS PROGRAM

by
V. Jeannot,
student of the Institut National Polytechnique de Grenoble

Master Thesis Report performed
under responsibility of
Prof.dr.ir. W.M.G. van Bokhoven
and support of
Dr.ir. J.A. Hegt
in the period
1 April through 28 June 1989

The Department of Electrical Engineering of the Eindhoven University of Technology accepts no responsibility for the contents of the M.Sc. Thesis reports.
# CONTENTS

**Chapter I : INTRODUCTION**

**SPECIFICATION**

**Chapter II : SYNTHESIS DESCRIPTION**

- II.1 DATA INPUT OF THE SYNTHESIS
- II.2 CHOICE OF THE SYNTHESIS
- II.3 FILTER COMPUTATION
- II.4 GENERAL SYNTHESIS SYNOPTIC

**Chapter III : FILTER DESCRIPTION**

- III.1 INTRODUCTION
- III.2 BASIC CIRCUITS COMPOSING THE FILTER
  - 3.2.1 Switched capacitor network and signal flow graph
  - 3.2.1.a Clock phases and notation
  - 3.2.1.b Example of SFG representation for SC network
- III.3 SFG LOOP STRUCTURE FOR THE FILTER
  - 3.3.1 Creation of - dC branches
  - 3.3.2 Loop structure
  - 3.3.3 Restriction on the realizable denominator of the transfer function
- III.4 FILTER STRUCTURE
  - 3.4.1 First filter structure
  - 3.4.2 Second filter structure
Chapter IV: A SPECIFIC APPROACH FOR CAPACITANCE COMPUTING

IV.1 INTRODUCTION

IV.2 SDG1 AND SDG2 AFTER ELIMINATING D^2 LOOPS

IV.3 POLYNOMIAL TRANSFORMATION
  4.3.1 Introduction
  4.3.2 Algorithm for polynomial transformation

IV.4 COMPUTING THE DENOMINATOR CAPACITANCES
  4.4.1 Structure
  4.4.2 Algorithm for the computation

IV.5 COMPUTING THE NUMERATOR CAPACITANCES
  4.5.1 Mason's rule application
  4.5.2 Derivation of a linear set of equation for the numerator capacitances
  4.5.3 Resolution of the linear set of equation
  4.5.4 Cofactor computation
    4.5.4.a Structure
    4.5.4.b Continuous fraction expansion
    4.5.4.c Algorithm for computation of the cofactors

IV.6 CONCLUSION

Chapter V: ELIMINATING NEGATIVE CAPACITANCES

V.1 INTRODUCTION

V.2 NEGATIVE CAPACITANCE ELIMINATION: THEORY
  5.2.1 Elimination of minus sign of capacitance in input branches
  5.2.2 Elimination of negative capacitances in the loop branches

V.3 MATRIX DESCRIPTION OF A FILTER

V.4 ALGORITHM FOR THE NEGATIVE CAPACITANCE ELIMINATION
  5.4.1 Initial matrix filter description
  5.4.2 Example of negative capacitance elimination

V.5 CONCLUSION
Les filtres à capacités commutées se prêtent bien à la réalisation de filtres d'ordres élevés, mais les calculs sont alors souvent complexes.

J.A. HEGT, en 1988, propose une synthèse systématique et générale de filtres intrinsèquement insensibles aux capacités parasites présentes dans toute réalisation de filtres intégrés. Il a élaboré deux structures de filtres, appelées SDG1 et SDG2, en utilisant de manière systématique une représentation simplifiée du filtre par graphe de fluence. La structure des deux filtres est décrite dans le chapitre III.

Le logiciel réalisé calcule les valeurs des capacités nécessaires pour former un filtre dont la fonction de transfert en Z est connue, que ce soit pour le premier type de filtre, SDG1, ou le deuxième, SDG2 :

**CALCUL DU FILTRE**

A cette étape de la synthèse, le type de synthèse (SDG1 ou SDG2) a été choisi, (cf § 2.2), la fonction de transfert \( H(z) \) est connue et est supposée compatible avec le type de synthèse choisi. Il s'agit donc du point de départ du travail réalisé en plusieurs étapes :

- **Etape 1** : utilisant la configuration de base décrite dans § 3.4, les capacités du filtre sont calculées (chapitre IV). Ils s'agit de l'étape la plus complexe.

- **Etape 2** : le signe des capacités calculées précédemment est à priori quelconque. Mais les capacités négatives peuvent être éliminées grâce à une modification systématique de la structure du filtre (chapitre V).
- **Étape 3** : le chapitre VI décrit l'optimisation de la dynamique possible du signal d'entrée. Ainsi, l'amplitude maximale du signal de toutes les sorties des ampli op sont identiques après cette opération.

- **Étape 4** : les capacités utilisent une surface relativement importante du circuit intégré. Pour minimiser cette surface, une optimisation de la capacité totale est réalisée (chapitre VIII).

- **Étape 5** : La capacité totale peut encore être réduite grâce à une technique appelée "partage de capacités" décrite dans le chapitre VIII.

La figure suivante résume la structure non seulement de la synthèse, mais aussi celle du logiciel et du rapport.

Voici quelques remarques concernant la figure suivante :

Le logiciel calculant la fonction de transfert la plus proche du filtre désiré est en fait basé sur un programme d'optimisation standard. Cette partie ne sera donc pas décrite ici.

Note (1) : Si l'utilisateur trouve que la fonction de transfert du filtre optimal n'est pas assez fidèle au filtre désiré, alors il devra choisir un filtre d'ordre plus élevé.

Note (2) : Certaines fonctions de transfert ne peuvent être obtenues par SDG1 et/ou SDG2 (voir § 3.3.3 et § 3.4.2). Un filtre d'ordre plus élevé est alors nécessaire.

Ces considérations sont reprises dans le chapitre II.
Figure 1 : description générale de la synthèse

- 5 -
A student exchange between the two following European universities has been organized:
- Technische Universiteit Eindhoven (TUE), in the Netherlands
- Institut National Polytechnique de Grenoble (INPG), in France.

This report of the final work performed at the TUE, Department of Electrical Engineering, before being graduate in the INPG, electronic department (Ecole Nationale Supérieure d'Électronique et de Radioélectricité (ENSERG)) concerns synthesis of switched capacitor filters.

Switched Capacitor (SC) filters, being integrated filters combining switches, capacitances and operational amplifiers, are very popular, because of their low cost compared to other integrated filters (small chip area, standard (C)MOS process, no tuning required ...).

The realization of a high order filter requires much computation effort that can not be done easily by hand. Therefore the filter synthesis must be automated. A synthesis method has been proposed in 1988 by J.A. Hegt [1], based on stray capacitance insensitive design graphs.

The aim of the work is to automate the synthesis described in [1].
A general description of the synthesis is given in chapter II, also introducing this report, because its structure is also the structure of the synthesis and of the software. More details concerning the filter are given in chapter III.

Then the different steps of the software, as announced in chapter II, are outlined in the next chapters.

Capacitance values of a basic filter structure are computed, as described in chapter IV. Chapter V shows how negative capacitances, computed with the previous part are eliminated, yielding a new filter topology. Then this realizable filter is optimized to increase the voltage swing (chapter VI) and to reduce the total capacitance (and the chip area) (chapter VII and VIII).

Appendix A gives the relations between the previous explanations and the derived software, realized in Pascal.

Appendix B describes two examples of synthesis, one for a filter of order 3 and one of order 8.

Appendix C gives a listing of the software.
SPECIFICATIONS

The realized software must automate the computation of the two synthesis methods given in [1], from a transfer function of the filter to a realizable and optimized filter. The realized set of tools must be adaptable to an eventually future synthesis based on stray capacitance insensitive design graphs.
CHAPTER II

SYNTHESIS DESCRIPTION
In this chapter, a general description of the filter synthesis will be given.

II. 1 DATA INPUT OF THE SYNTHESIS

Numerous methods are known for the design of switched capacitor filters, many of them based on a transformation of continuous-time prototype filters. This method can be efficient when the desired filter transfer is tabulated, and when an analog prototype is available. But given a desired filter transfer, for example a graphic representation of the ideal filter transfer function, the user must first try to find a "standard" filter looking like this one. This is not always very efficient.

The synthesis proposed here doesn't require any continuous time model, or tabulated filter. Given the description of the desired filter, and its order, an optimization program will find the nearest available transfer function in the Z domain for that given order. It is this optimal transfer function that will be realized by the switched capacitor filter. The synthesis is able to realize any transfer function except a few ones as described in § 3.3.2 and § 3.4.2. If the user finds that the optimal filter transfer is not close enough to the desired one, he must choose a higher order for his filter in order to get a better approximation.
II . 2 CHOICE OF THE SYNTHESIS

At this step of the synthesis, it is assumed that the desired filter is represented by a known $Z$ transfer function, of order $N$. As described in chapter III, two different filters are available to realize this transfer function, called SDG1 and SDG2 (Stray insensitive Design Graph 1 and 2). There are some transfer functions that are realizable with SDG1 and not with SDG2 (See § 3.3.3 and § 3.4.2 for more details). But SDG1 has the disadvantage, like many other synthesis methods, to require a fixed input signal for a full clock-period, and therefore a sample and hold circuit may be required. As shown in chapter III, when the transfer function is not realizable, the best solution is to use a higher order for the filter.

This choice will be made as depicted now:
- If both synthesis are realizable, then both will be computed, the best realization, with the lower total capacitance for example, will be chosen.
- If neither SDG1 nor SDG2 are realizable, or if SDG2 is not realizable and a fixed input signal for a full clock period is not available, then a filter with a higher order may be used, as described now:

It is always possible to multiply the initial transfer function by a doublet, $(z-a)/(z-a)$, yielding a realizable transfer function. But it is better to take benefit from the extra degree of freedom for a higher order filter to obtain a better approximation in a realizable form.
II . 3 FILTER COMPUTATION

At this step of the synthesis, it is assumed that the type of synthesis, SDC1 or SDC2 has been chosen, as described in § 2.2, and that this type of synthesis can realize the desired transfer function $H(z)$. Now the synthesis proceeds with the following steps:

- **First step:** using the basic structure described in § 3.4, the capacitances of the filter are computed (chapter IV).
- **Second step:** the sign of those computed capacitances can be either positive or negative. But using a systematic modification in the topology, the minus signs can be eliminated, as will be described in chapter V.
- **Third step:** To increase the input voltage range without any saturation, a voltage swing optimization will be done, as described in chapter VI.
- **Fourth step:** the capacitors require a relative large share of the total area on an integrated circuit. Therefore a systematic total capacitance optimization will be done (chapter VII).
- **Fifth step:** The total capacitance can be reduced thanks to a capacitance sharing (chapter VIII).

II . 4 GENERAL SYNTHESIS SYNOPTIC

The following figure gives a summary of the previous parts with a general description of the synthesis and a reference to the chapter where the design steps are described in more details.
Chapter II
SYNTHESIS DESCRIPTION

Input of the desired filter
(Bode diagram, ...)

Choice of the filter order
N=2,...,10,...

Compute the best transfer
function in H(Z)

H(z)

Choice of the synthesis
SDG1 - SDG2

STEP 1
(Chapter IV)

H(z), SDG1/2

Compute the capacitances
with the initial structure

Structure modification
to get only positive
capacitances

STEP 2
(Chapter V)

Voltage swing optimization

STEP 3
(Chapter VI)

Total capacitance
optimization

STEP 4
(Chapter VII)

"Capacitance sharing"

STEP 5
(Chapter VIII)

FILTER COMPUTATION

Figure 2.1: general description of the
synthesis

- 14 -
Note (1) : As described in § 2.1, if the user finds that the optimal filter transfer is not close enough to the desired one, a higher order for the filter must be chosen.

Note (2) : As described in § 2.2, a higher order for the filter is required when SDG1 and/or SDG2 is not realizable.

A number of remarks concerning figure 2.1 can be made:

The software computing the best transfer function for a given amplitude diagram uses standard optimization program tools. That is why this part will not be described here.

The work that has been done concerns mainly the "filter computation", that is to say the computation of its topology and the capacitance values. Step 1, 2, 4 and half of step 3 are available and have been tested successfully; The algorithms for the other part of step 3, and step 5, will be given.

Note that the program concerning the interface with the user can easily be made by any other programmer, because the filter computation can be used like a big black box, input being the transfer function H(z), the output being the description of the finally obtained filter. This will be shown in appendix A.
III. 1 INTRODUCTION

The basis of the software that has been written is the filter synthesis described by J.A. HECT in his thesis [1]. The filter and its most important properties, which are elements that are required to understand the software that has been derived, will be considered in this chapter.

The filter synthesis described here takes the properties of strays insensitive biphase-clocked SC structures as a starting point. The set of elementary signal flow graph (SFG) building blocks that corresponds to these stray-insensitive SC structures will be described in § III.2.

The filter can be represented by an adequate signal flow graph (SFG) composed from branches of this basic set. First the part of this SFG corresponding to the loop structure of the filter will be considered in § III.3. Finally, using this structure, two general graph structures will be proposed in § III.4.
III . 2 BASIS CIRCUITS COMPOSING THE FILTER

3.2.1 SWITCHED CAPACITOR NETWORK AND SIGNAL FLOW GRAPH

3.2.1.a Clock phases and notation
A biphase filter uses the following clocks:

\[
\begin{align*}
\text{on} & : e_0, o_0, \ldots, e_n, o_n, \ldots, \\
\text{off} & : e_{n-1}, o_{n-1}, \ldots, e_1, o_1, \ldots
\end{align*}
\]

Figure 3.1 : Biphase non-overlapping clocks

The clock phase called 'e' (resp 'o') commands the switches 'e' (resp 'o').

There are two topologies of the circuit, one for each clock phase. To represent the interactions between these two topologies, each physical node of the circuits must be represented in the signal flow graph by two nodes, one for each clock phase. As a consequence, the voltage of one node \( V \) is represented by \( V^e \) at the end of the even phase, \( V^o \) at the end of the odd phase.

3.2.1.b Example of SFG representation for SC network
The computation of the SFG will be outlined in the following example:

Figure 3.2 : Example of a SC network
In the circuit of fig 3.2, $V_a$ is connected to the circuit only during clock-phase 'e'. As a consequence for the corresponding SFG, no branch will come from $V_a^0$, this node is redundant. For analogous reasons, $\Delta Q_b^0 = 0$.

During clock-phase 'e', $C$ is charged with $V_a$, the charge of plate a of $C$ is $q_a(nT) = C \cdot V_a^e(nT)$, and the charge of plate b of $C$ is $q_b(nT) = -C \cdot V_a^e(nT)$, $T$ being the sample period.

During 'o', $C$ is discharged to virtual ground. Therefore the following equation is finally obtained:

$$\Delta Q_b^0((n+1/2)T) = q_b((n+1/2)T) - q_b(nT) = -C \cdot V_a^e(nT)$$ (3.1)

Note that half a clock period happened from $V_a^e(nT)$ and $\Delta Q_b^0((n+1/2)T)$, corresponding to a delay time of $z^{-1/2} \cdot d$.

Equation (3.1) can be rewritten like (3.2) when $Z$ transformation is used:

$$\Delta Q_b^0 = -dC \cdot V_a^e$$ (3.2)

Equation (3.2) is represented by the following flow graph:

Figure 3.3 : Signal flow graph of the network of fig 3.2
3.2.2 BASIC CONCEPTS FOR STRAY-INSENSITIVE SYNTHESIS

The following stray capacitances will be considered:

![Stray Capacitances Diagram]

Figure 3.4: Stray capacitances

In [4], Hasler gives the following rules to be insensitive with respect to the stray capacitances of fig. 3.4:

1: In any clock-phase the circuit does not contain other nodes than voltage source nodes, ground or virtual ground nodes.

2: A terminal of a capacitance is never switched from a voltage source node in one clock phase to a virtual ground node in the next phase.

In [1], Hegt demonstrates that these constraints are sufficient but not necessary conditions for stray capacitance insensitivity. In some cases, a floating node can be used, which is a node that is not connected in one phase. The following figure shows a stray capacitance insensitive circuit using such a floating node:

![Stray CapacitanceInsensitive Circuit Diagram]

Figure 3.5: Example of stray capacitance insensitive circuit, containing a floating node in clock-phase 'e'
The dashed lines of figure 3.5 represent the stray capacitances $C_a$, $C_b$ and $C_c$.

It is obviously that $C_c$ has no influence because it is assumed that the output resistances of the op amps are zero.

$C_a$ has no influence in both clock-phases. In phase '0' it is connected to a voltage node that is assumed to have a zero output impedance. Therefore its presence has no influence on the voltage of node A. In phase 'e', the voltage across both capacitances $C_a$ and $C_i$ are the same, because $V_B$ is always 0, B being a virtual ground node. A is a floating node, no charge can come from the input voltage node. Therefore no current is flowing through $C_a$. As a consequence, $C_a$ has no influence in both clock-phases.

For $C_b$, the demonstration is easier, because in both clock-phases it is connected to a real ground and a virtual ground. No current can flow through it, $C_b$ has no influence on the circuit.

3.2.3 ELEMENTARY SWITCHED CAPACITOR STRUCTURES

Using the constraints given in the previous part, a systematic search is done. The following figure shows the 6 finally obtained circuits with their signal flow graphs:
The mode a, b and c of figure 3.6 will be used in the basic structure, as it will be depicted in § 3.3 for mode a and b, § 3.4 for the mode c. Looking at mode d and e, one capacitance is able to realize two SFG branches. These modes can sometimes be used. Because two branches 'share' the same capacitance, this will be called 'capacitance sharing'. More details will be given in chapter VIII. Mode f of figure 3.6 can be used in case the series switch of a mode c circuit is redundant.
Now, the following circuits with an op amp will be used:

![Op amp network with continuous feedback](image)

Figure 3.7: Op amp network with continuous feedback

Demonstration of the SFG given in fig 3.7:

\[ V(nT) = V^e(n) = -1/C_{FB} \cdot Q(nT) \]  \hspace{1cm} (3.3)

If it is assumed that at the beginning, \( Q(0) = 0 \), then

\[ Q(nT) = \sum_{i=0}^{n} \Delta Q(iT) \]  \hspace{1cm} (3.4)

and (3.3) can be rewritten as:

\[ V^e(n) = -1/C_{FB} \cdot \sum_{i=0}^{n-1} [\Delta Q^e(i) + \Delta Q^o(n + 1/2)] - 1/C_{FB} \cdot \Delta Q^o(n-1/2) \]

\[ -1/C_{FB} \cdot \Delta Q^e(n) \]  \hspace{1cm} (3.5)

The two first elements in the right part is nothing but \( V^o(n-1/2) \).
This can be rewritten as:

\[ V^e(n) = V^o(n-1/2) - 1/C_{FB} \cdot \Delta Q^o(n) \]  \hspace{1cm} (3.6)

or using Z transformation, with \( d = z^{-1/2} \) being the half clock delay function:

\[ v^e = dV^o - 1/C_{FB} \cdot \Delta Q^e \]  \hspace{1cm} (3.7)
Equation (3.7) will be represented by the following flow graph:

Figure 3.8: Signal flow graph of equation (3.7)

The analogous equation of (3.6) is for $V_0$:

$$V_0 = d \cdot V^e - \frac{1}{C_{FB}} \cdot \Delta Q_0$$  \hspace{1cm} (3.8)

Combining (3.6) and (3.8), the SFG representation for the circuits of figure 3.7 is the following:

Figure 3.9: Op amp network with continuous feedback and corresponding signal flow graph representation

3.2.4 SET OF ELEMENTARY SFG STRUCTURES

As an example, combining the circuits of fig 3.7 and 3.6 mode a, the following circuit is available:
Chapter III

FILTER DESCRIPTION

Figure 3.10: circuit and SFG of the combination of fig 3.7 and 3.6, 'mode a'

A simplification can be made, as shown in the following example: the SFG nodes ΔQb and ΔQo will be eliminated, as depicted in the following figure:

Figure 3.11: Finally obtained SFG of circuits of fig 3.10

Note that the inputs of the op amps are disappeared in this representation, but Vb still represents the op amp output.

Figure 3.11 shows that the transfer function depends only on the ratio C/CFB. This is interesting for the following reason: the absolute accuracy of integrated circuit capacitances is very bad, 20% for example. But the accuracy of the ratio of two integrated capacitances on the same chip is much better1, 1% to 0.1%! That is one of the

The low absolute integrated circuit capacitance accuracy is due mainly

1
Chapter III  
FILTER DESCRIPTION  

reasons why SC filters are being used.  
The same transformation as was used for fig 3.11 can be applied  
for every circuit of fig 3.6.  
Finally, the following complete set of branches will be used to  
compose SFG's, for Stray Insensitive biphase-clock SC filters:  

Figure 3.12: Set of SFG branches composing the filter  
The branches of fig 3.12 will be referred to their gain, that is  
to say -C, +dC, -pC, d branches.  

III . 3 SFG LOOP STRUCTURE FOR THE FILTER  

3.3.1 CREATION OF -dC BRANCHES  
The sign of allowed branches is fixed because only positive  
capacitances are realizable. Nevertheless, the sign of a dC branch is  
not fixed, if a -dC branch is needed, it can be constructed by the  
combination of a d branch and a -C branch, as depicted in the  
following figure:  

to the low precision of the dielectric thickness. The ratio between  
two capacitance depends on geometrical factors which are much more  
accurate.
Figure 3.13: Creation of a -dC branch as a combination of a d branch with a -C branch

This will only work if $V_a^O$ in figure 3.13 is really equal to $d-V_b^e$, that is to say if there are no other branches ending on the $V_a^O$ node.

3.3.2 LOOP STRUCTURE

The following figure shows the loop structure of the filter, which will generate the denominator of the transfer function.
Figure 3.14: Loop-structure for the realization of the transfer 

\( a : n \) odd, \( b : n \) even, \( n \) being the filter order

Here the \( \pm dC \) branches are in fact \( +dC \) branches, or combination of \( +d \) and \( -C \) branches as depicted in fig 3.13 (§ 3.3.1). The elimination of negative capacitances will be described in chapter V. Note that in order to be able to do this transformation, no other branches may end on the nodes marked with a dashed circle in fig 3.14.

3.3.3 RESTRICTION ON THE REALIZABLE DENOMINATOR OF THE TRANSFER FUNCTION

The only sign that is really fixed is the sign of \( -C_s \) self-loop. Using Mason's rule it can be shown that the product of the realized poles is:

\[
\prod_{i=1}^{N} Z_{\text{pole}_i} = \frac{1}{1 + C_s} \quad (3.9)
\]
which means that

\[ 0 < \prod_{i=1}^{N} Z_{\text{pole}_i} \leq 1. \]  \hspace{1cm} (3.10)

Of course the upper bound means no real restriction because only stable filters are interesting.

The lower bound means that an odd number of poles on the negative real axes can not be realized by this structure. Several ways to circumvent this problem have been given by Hegt².

III . 4 FILTER STRUCTURE

Using the loop structure depicted in the previous part, two types of filters have been developed.

3.4.1 FIRST FILTER STRUCTURE

The following figure gives the structure of the first filter, called Stray insensitive Design Graph, form 1, SDG1:
The SDG's of fig 3.15 and 3.16 have the disadvantage to require a fixed input signal for a full clock-period, represented by the $d$ branch from $V_{in}$ to $V_{in}$. 

Note that the sign of $-pC$ branch is fixed. However, in case a $+pC$ branch is required, a $-pC$ branch will be used, but the polarity of the node on which this branch ends will be flipped. This method will be used in chapter V describing the negative capacitance elimination. Therefore in the graph of figure 3.15 and 3.16, only the sign of $Cs$ is
3.4.2 SECOND FILTER STRUCTURE

A second filter is given now, called SDG 2. Figures 3.17 and 3.18 give its structure:

This filter doesn't require a fixed input signal for a full
clock-period. However, there is one restriction compared to SDG1, because the signs of both \(-\text{pCo}\) and \(-\text{Cn}\) must be the same. Using Mason's rule, it can be shown that:

\[
P \prod_{i=1}^{N} Z_{\text{zero}} = \frac{C_0}{C_n + C_0} \quad (3.11)
\]

and

\[
0 \leq P \prod_{i=1}^{N} Z_{\text{zero}} \leq 1 \quad (3.12)
\]

The lowest bound means that it is not possible to realize an odd number of zeros on the negative real axis with this graph, except for the case that there is at least one zero at \(z=0\) (which means that \(\text{Co}=0\)).

In cases where the upper bound applies, the reciprocal value of zeros outside the unit-circle can be used, the amplitude transfer function will not change.
CHAPTER IV

A SPECIFIC APPROACH FOR CAPACITANCE COMPUTING
IV. 1 INTRODUCTION

Many methods for capacitance value computation have been studied. A first class is composed of methods appropriate to any filter: for a given set of capacitance values and for a given structure, methods like Mason's or Cramer's rules provide the transfer function of the filter. An iterative approximation program can compare the obtained transfer function with the desired one and find the correct set of capacitance values by iterative steps, as described in fig.4.1.

\[
H(z) = \frac{\sum_{i=0}^{N} N_i \cdot Z^i}{\sum_{i=0}^{N} D_i \cdot Z^i}
\]

Figure 4.1: Algorithm appropriate to any filter for capacitances computing

- 34 -
This class of methods described in fig. 4.1 requires two important program parts:

- A multidimensional function minimization procedure\(^3\) for the iterative steps, which will minimize the error between the desired transfer function and the obtained transfer function for a given set of capacitances

- An efficient tool to compute the filter transfer function for a given set of capacitance values.

Many cycles through the iterative loop that appears in fig. 4.1, mainly composed of the two previous quoted software, will be needed. Therefore each step of it, especially the filter transfer function computation, must be very rapid\(^4\).

For example a Downhill Simplex Method in \(2N+1\) dimensions, which doesn't require the partial derivative \(\frac{\partial \text{error}}{\partial C_k}\) given for example in [2]. Note that this procedure is available, because it is used for the computation of the transfer function.

One of the solutions is to compute the relation between the set of capacitances and the transfer function coefficients only once for a given filter structure. This requires the use of symbolic computing, the symbols being the capacitances. The step that will be repeated many times consists only of putting the last set of capacitance values in the formula, and evaluating it.

Two solutions have been studied to compute the filter transfer function in a symbolic way:

- Using Mason's rules, using a systematic way to find paths and loops, see [3].

- Using a symbolic determinant computation. This method has been studied in more details, yielding a program tested successfully: It is based on developing a row or a column of the determinant with lower
A second class consists of methods specific for the SDG1 or SDG2 synthesis, having the important following advantages:
- no convergence problem of an iterative approximation program
- higher speed because of a direct computation.
- shorter program because specific and nicer algorithms can be used.
A drawback of this alternative solution is that one eventually has to rewrite an important part of the program in case a new synthesis is adopted.

Actually a method specific to SDG1 and SDG2\(^5\) has finally been chosen, and will be depicted in this chapter 4:

First the SFG structure will be simplified, generating a new and more convenient transfer function representation (IV 4 2), then order determinants. This is repeated until there are only order 1 determinants. It is easy to understand that a recursive program can be used. Note that for numerical determinant, an LU decomposition for example may be preferred. But the method considered here is very efficient because the matrices in this application are sparse. It can be shown that a convenient formula representation can be derived. The formula for a 6 order determinant corresponding to a third order filter is computed in less than one second.

However the time available before the end of the training period was too short to go on in this interesting way.

Whenever it was possible, non specific procedures have been isolated, and can easily be used in an eventually new synthesis. Furthermore, the algorithm used here is similar for both SDG1 and SDG2. May be, it can be adopted for an eventually new synthesis.
capacitances that correspond with the denominator polynomial will be considered (IV 4.3). Next the capacitances concerning the numerator polynomial will be computed (IV 4.3). The initial filter structure yields both positive or negative capacitances. The inversion of the sign for negative capacitances will be explained in chapter V.

IV . 2 SDG1 AND SDG2 AFTER ELIMINATING D^2 LOOPS

To make the computation easier, the SFG structure will be simplified by eliminating the op amp d^2 loops. Then every branch that ends on these loops must be divided by 1-d^2 - p. The following figures show the resulting graphs for even order after elimination of the d^2 loops.

fig 4.2 : SDG1, odd order after the d^2 loop elimination
IV.3 POLYNOMIAL TRANSFORMATION

4.3.1 INTRODUCTION

The $d^2$ loop elimination generates new kinds of branch gains, with sometimes a $1/p$ factor: $d/p \cdot C$ or $-C/p$. The direct application of Mason's rules generate the following kind of expression for SDG1 and SDG2, odd order, here 5:

$$H = \frac{N_0 + N_1/p + d^2N_2/p^2 + d^2N_3/p^3 + d^4N_4/p^4 + d^4N_5/p^5}{D_0 + D_1/p + d^2D_2/p^2 + d^2D_3/p^3 + d^4D_4/p^4 + d^4D_5/p^5} \quad (4.1)$$

This kind of numerator or denominator polynomial will be called: "$d/p$ polynomial, first type", for SDG1 and SDG2, for odd order.

With SDG1 synthesis, even order, the transfer function numerator expression is different. The following expression shows for example a
4th order transfer function:

$$H = \frac{N_0 + d^2N_1/p + d^2N_2/p^2 + d^4N_3/p^3 + d^4N_4/p^4}{D_0 + D_1/p + d^2D_2/p^2 + d^2D_3/p^3 + d^4D_4/p^4} \quad (4.2)$$

The numerator polynomial will be called: "d/p polynomial, second type". It should be mentioned that the denominator is always a first type d/p polynomial.

### 4.3.2 ALGORITHM FOR POLYNOMIAL TRANSFORMATION

Several algorithms have been developed for this part. Here, an effective algorithm will be proposed, yielding a short program. The transformation from a standard polynomial into a d/p polynomial first type, defined in the previous part, will be described. The algorithm for the d/p polynomial second type is almost the same.

The transformation will be done in several steps:

The initial polynomial given in the following equation:

$$\mathcal{P}_1(d^2) = \sum_{i=0}^{N} P_1[i] \cdot d^{2i} \quad (4.3)$$

will be transformed into $$\mathcal{P}_2$$:

$$\mathcal{P}_2(p) = \sum_{i=0}^{N} P_2[i] \cdot p^i \quad \text{with } p = (1 \cdot d^2) \quad (4.4)$$

Then $$d^2$$ will be put in factor, yielding $$\mathcal{P}_3$$:

---

6

An approach made by an another student uses the relations obtained directly from the initial polynomial to the final one. They are complex, yielding a very large program.
Chapter IV

A SPECIFIC APPROACH FOR CAPACITANCE COMPUTING

\[ \mathcal{P}_3 = P_3[n]p^n + P_3[n-1]p^{n-1} + d^2 \sum_{i=0}^{N-2} P_3[i]p^i \]  \hspace{2cm} (4.5)

Next if necessary, \( d^2 \) will be put in factor several times, yielding \( \mathcal{P}_4 \):

\[ \mathcal{P}_4(d^2) = P_4[n]p^n + P_4[n-1]p^{n-1} + d^2 \left( P_4[n-2]p^{n-2} + P_4[n-3]p^{n-3} + \ldots + P_4[1]p + P_4[0] \right) \]  \hspace{2cm} (4.6)

Finally, there is no problem to find the numerator polynomial of eq.(4.1).

First step: Transformation of \( \mathcal{P}_1 \) of eq.(4.3) to \( \mathcal{P}_2 \) of eq.(4.4) :

Deriving the both equations and finding their values for \( p=0 \) or \( d^2-1 \) yields:

\[ (i!)P_2[i] = \mathcal{P}_2 \bigg|_{p=0}^{(1)} = (-1)^i \mathcal{P}_1 \bigg|_{d=1}^{(1)} \hspace{2cm} i=0 \ldots N \]  \hspace{2cm} (4.7)

where \( \mathcal{P}_k \) is the \( i \)th derivative of the polynomial \( \mathcal{P}_k \). The \( (-1)^i \) in the right part of eq.(4.7) is due to the fact that \( d^2-1-p \). Therefore

\[ \frac{d}{dp} (d^2) = \frac{d}{dp} (1-p) = -1. \]  \hspace{2cm} (4.8)

Equation.(4.7) provides the value \( \mathcal{P}_2 \).

Second step: Transformation of \( \mathcal{P}_2 \) of eq.(4.4) to \( \mathcal{P}_3 \) of eq.(4.5) :

Eq.(4.5) is rewritten with \( (1-p) \) instead of \( d^2 \) factor :

- 40 -
The identification of the same power of $p$ coefficients between eq. (4.9) and eq. (4.4) yields:

\[
P_3 = P_3[n]p^n + P_3[n-1]p^{n-1} + \sum_{i=0}^{n-2} P_3[i]p^i - P_3[0] + P_3[n+1]p^1 - \sum_{i=0}^{n-2} P_3[i]p^i
\]  \hspace{1cm} (4.9)

Equation (4.10) can be inverted:

\[
P_2[n] = P_3[n]
\]
\[
P_2[n-1] = P_3[n-1] - P_3[n-2]
\]
\[
P_2[n-2] = P_3[n-2] - P_3[n-3]
\]
\cdots
\[
P_2[1] = P_3[1] - P_3[0]
\]
\[
P_2[0] = P_3[0]
\]  \hspace{1cm} (4.10)

Equation (4.10) provides an efficient method to compute $P_3$.

Note that it can be shown that the algorithm to compute the second type polynomial is almost the same.

Third step : Transformation of $P_3$ of eq. (4.5) to $P_4$ of eq. (4.6) : this step is very short : the only difference between eq. (4.5) and
Chapter IV  
A SPECIFIC APPROACH FOR CAPACITANCE COMPUTING

Eq. (4.6), is that \( d^2 \) is put in factor several times in eq. (4.6), in exactly the same way as the transformation from \( P_2 \) to \( P_3 \) (eq. (4.4) to eq. (4.5)). Therefore the second step is repeated with the following new polynomial \( P_2^* \):

\[
P_2^* = \sum_{i=0}^{N-2} P_3[i] \cdot p^i
\]

(4.12)

providing a new polynomial \( P_3^* \). Then this step is done again:

\[
P_2^{**} = \sum_{i=0}^{N-4} P_3^*[i] \cdot p^i
\]

(4.13)

etc.

until the polynomial of eq. (4.6) is reached.

Actually, the second and the third steps are composed of only one recursive procedure. Appendix A provides the notations used in the software.

IV.4 COMPUTING THE DENOMINATOR CAPACITANCES

First, only the capacitances that correspond with the denominator polynomial will be considered, that is to say capacitances of the loop structure.

IV.4.1: STRUCTURE

The following figure shows the loop structure after the \( d^2 \) loop elimination, valid for both SDG1 and SDG2.
fig 4.4: simplified loop structure, even order

An input branch and a new node are added as shown in the following figure:

fig 4.5: theoretical structure for computation

The following explanation will show why this structure is interesting for capacitance computation.

Because this SFG has the same loop structure as the graphs of figures 4.2 and 4.3, the denominator of its transfer function is exactly the same as for SDG1 and SDG2. According to Mason's rule, the
Chapter IV

A SPECIFIC APPROACH FOR CAPACITANCE COMPUTING

denominator of the transfer function consist of sums of products of loop gains, which contain either $d^2/p^2$ factors or $-Cs/p$. As a consequence odd powers of $1/p$ can only be generated with the $-Cs/p$ factor. The numerator can easily be found, because it is nothing but the denominator, after elimination of the touching loops, here $-Cs/p$ loop. This means that the numerator is equal to the sum of all odd coefficients of the denominator of fig 4.5 and therefore of fig 4.2 and 4.3.

For an order 5, the transfer function of figure 4.5 is:

$$H_1 = \frac{D_0 + d^2D_2/p^2 + d^4D_4/p^4}{D_0 + D_1/p + d^2D_2/p^2 + d^2D_3/p^4 + d^4D_4/p^4 + d^4D_5/p^5}$$

(4.14)

Now, this transfer function will be expanded as a continuous fraction.

Fig 4.6 one loop structure

When the one loop structure of fig 4.6 is considered, its transfer function can be written as:

$$h_1 = \frac{1}{1 - (-Cs/p)}$$

(4.15)

Now a new loop, with a gain of $Cs C_4 d^2/p^2$ is added as depicted in fig 4.7.
This new loop can be eliminated by replacing $-C_s/p$ by

$$\frac{-C_s/p}{1-C_{45} C_{45} d^2/p^2}.$$  \hspace{1cm} (4.16)

resulting in:

$$h_2 = \frac{1}{1 - \frac{-C_s/p}{1 - C_{45} C_{45} d^2/p^2}}.$$  \hspace{1cm} (4.17)

This procedure is continued until the last loop $C_{21} C_{12} d^2/p^2$ is reached, resulting in the following continuous fraction:
IV.4.2 ALGORITHM FOR THE COMPUTATION OF DENOMINATOR CAPACITIES

It is assumed that we know the denominator polynomial, and its $D_0$ coefficient is equal to 1:

$$\text{Denominator} = 1 + \frac{D_1}{p} + \frac{d^2D_2}{p^2} + \frac{d^3D_3}{p^3} + \frac{d^4D_4}{p^4} + \frac{d^5D_5}{p^5}$$  \hspace{1cm} (4.19)

Then it is easy to find the numerator polynomial:

$$\text{Numerator} = 1 + \frac{d^2D_2}{p^2} + \frac{d^4D_4}{p^4}$$  \hspace{1cm} (4.20)

We can rewrite the transfer function $H_1 = \frac{\text{Numerator}}{\text{Denominator}}$ into

$$H_1 = \frac{1}{\text{Denominator}}$$  \hspace{1cm} (4.21)

and then divide denominator by the numerator:

$$\text{Denominator} / \text{numerator} = 1 + \frac{A_0}{p} \frac{\text{Denominator'}}{\text{Numerator}}.$$  The new denominator order has decreased by one. This result in:

$$H_1 = \frac{1}{1 + \frac{A_0}{p} \frac{\text{Denominator'}}{\text{Numerator}}} = \frac{1}{1 + \frac{A_0}{p} \frac{1}{\text{Denominator'}}}$$  \hspace{1cm} (4.21)
Chapter IV  A SPECIFIC APPROACH FOR CAPACITANCE COMPUTING

This procedure is repeated until Numerator and Denominator are reduced to one.

Comparing the finally obtained continued fraction expansion with the expression in equation (4.18), results in values for $C_{\omega}$ and for all products $C_{i,i+1} C_{i+1,i}$; $i=1,\ldots,N$, with $N$ being the filter order. Temporarily, all "forward" capacitances $C_{i,i+1}$ are chosen equal to 1. This can be done without loss of generality: in a later stage, all the capacitances will be rescaled for optimization purposes. Therefore all denominator capacitors are known now.

IV.5 COMPUTING THE NUMERATOR CAPACITANCES

Now, the remaining unknown capacitances, that correspond with the numerator polynomial, will be considered. These are the capacitances $C_0,\ldots,C_N$ that correspond to the branches connected to the input. Because SDG1 and SDG2 have different input branches, the computation will be different, but the nice property of the following algorithm is, that it is valid for both graphs.

Using Mason's rules, a numerator transfer function expression will be developed. In this expression, only $C_0,\ldots,C_N$ are unknown. Moreover, a matrix representation will be developed. Then this matrix will be easily resolved using back-substitution, resulting in values for the numerator capacitances.

IV.5.1 MASON'S RULE APPLICATION

The following expression is the basic relation for the numerator transfer function:
Chapter IV A SPECIFIC APPROACH FOR CAPACITANCE COMPUTING

Numerator = \sum_{i} (\text{Path gain})_{i} \cdot \text{cofactor}_{i} \quad (4.22)

Each path from input to output corresponds to one numerator capacitor \( C_{i} \), \( i = 1, \ldots, N \). Looking at figure 4.2 and 4.3, input to output path number \( i \) goes through \( C_{i} \), and some \( dC/p \) forward branches (capacitances \( C_{j, j+1} \) which have been chosen equal to 1). Then the path gain, being the product of all branches in this path, is \( C_{i} \cdot d^{j_{i}}/p^{k_{i}} \), \( j_{i} \) and \( k_{i} \) being integers.

According to Mason rules, the cofactor of path \( i \) is nothing but the denominator after elimination of all the loops touching this path. Therefore it is a polynomial which coefficients depend only on denominator capacitances, which are already known.

In a first step, the numerical values of the cofactor polynomials are assumed to be known, more explanations about their computation will be given in § IV.4.5.4.

IV.5.2 DERIVATION OF A LINEAR SET OF EQUATION FOR THE NUMERATOR CAPACITANCES

Equation 4.22 can be written in the following way:

\[
\text{Numerator} = \sum_{i=0}^{n} (C_{i} \cdot d^{j_{i}}/p^{k_{i}}) \cdot P_{i} \quad (4.23)
\]

The numerator is a type 1 or type 2 \( d/p \) polynomial that is also known. \( C_{i} \) are unknown, and \( P_{i} \) is the polynomial cofactor of the path going through \( C_{i} \). The numerical value of this polynomial is assumed to be known as well as \( d^{j_{i}}/p^{k_{i}} \) factor. Multiplication of the cofactor with this factor is the same as shifting the cofactor polynomial coefficients \( k_{i} \) positions to the right. Then using (4.23), the identification of the same power of \( 1/p \) coefficients gives the following matrix relation:

\[
\text{Matrix} \times [C] = [\text{Numerator}] \quad (4.24)
\]
Chapter IV A SPECIFIC APPROACH FOR CAPACITANCE COMPUTING

- [C] is the colon vector composed of $C_0, \ldots, C_N$;
- [Numerator] is the colon vector composed of the numerator coefficients $N[0], \ldots, N[N]$;
- Matrix is a numerical matrix which value will be explained in the following part.

As an example, the computation of the numerator capacitances of a third order SOGI will be shown in details. For a third order SOGI, eq (4.24) can be written as:

$$
\text{Num} = \text{d} [ -C_3 -C_2 C_3/p + d^2 (C_3 C_2 C_1 - C_1 C_2 C_3)/p^2 + d^2 C_0 C_1 C_2 C_3/p^3 ]
$$

(4.25)

Note that the d factor represents a fixed delay and is not very important, it will therefore be omitted here. It is due to the fact that the filter has an even clock phase input, and an odd clock phase output.

In equation (4.25), all numerical values, except $C_i$ i=1,..,N, are known. Then by identifying coefficients, results in the following set of equations:

$$
\begin{align*}
-C_3 & = N[0] \quad (4.26.a) \\
-C_2 C_3 & = N[1] \quad (4.26.b) \\
C_1 C_2 C_3 & = -C_1 C_2 C_3 C_1 = N[2] \quad (4.26.c) \\
C_1 C_2 C_3 C_0 & = N[3] \quad (4.26.d)
\end{align*}
$$

Note that the set of equation (4.26) is lower triangular. In the following part, this result will be generalized for both SDGI and SDG2.

IV.5.3 RESOLUTION OF THE LINEAR SET OF EQUATION

A simple back substitution algorithm can be used. Indeed, choosing a right order in the capacitance vector $[C]$ of eq.(4.24), the
matrix of the same equation can be lower triangular, like in eq.(4.26). This can be explained as follows:

The cofactor of any path has its lowest order coefficient equal to 1. As a consequence, any expression in the sum of eq. (4.24) is a polynomial which lowest order value depends only on $C_i$. This lowest order is equal to the number of $dC/p$ branches of path number $i$. In case of SDG1, looking at fig 4.2, this lowest order is $N - i$, in case of SDG1. Thus, numerator coefficient number $j$, $N[j]$ depends on only $C_w$, $C_{w-1}$, ..., $C_i$. Therefore the matrix of equation (4.24) is lower triangular if the column vector $[C]$ of eq.(4.24) is the following:

$$[C_{SDG1}] = [C_w,C_{w-1},...,C_1,C_0]^T$$  \hspace{1cm} (4.27.a)

Note that this is also true for SDG2 with:

$$[C_{SDG2}] = [C_0,C_w,C_{w-1},...,C_2,C_1]^T$$  \hspace{1cm} (4.27.b)

It can also be demonstrated that the diagonal coefficients of the matrix of eq 4.24, actually the "pivots", are equal to ± 1 ($C_j, j+1$ are assumed to be equal to 1). Therefore the resolution of eq (4.24) can be done efficiently and accurately by using back substitution.

This straightforward algorithm for computing numerator capacitances is very efficient, especially in computer time. However, in the last part, the cofactor polynomial was assumed to be known. This is considered in the next part.

**IV.5.4 COFACTOR COMPUTATION**

In the last part, it has been shown that the cofactor of an input to output path depends only on denominator capacitances, but its computation was not described. That's the aim of the following part. Thus
a new graph will be introduced, leading to a new continuous fraction expansion.

IV.5.4. a structure

A path from input to output has the following shape:

Figure 4.8: a path example

According to Mason's rule, the remaining loops that don't touch the path are given in the following figure:

Figure 4.9: remaining loop structure
Chapter IV

A SPECIFIC APPROACH FOR CAPACITANCE COMPUTING

Remarks:
- The $-Cs/p$ loop is always eliminated, because it touches the output, and therefore any path from input to output.
- The number of remaining loops depends on the path (i).
- The eliminated loops are always at the right-hand side of the graph of fig 4.9.
- For some paths, all loops are eliminated. (In this case the cofactor is 1).

Therefore the cofactor is nothing but the denominator transfer function of any graph having this loop structure. Because the eliminated loops are always at the right-hand sides of the graph, a new output will be defined at the outmost left side of the graph of figure 4.9. Therefore the following graph will be used.

Figure 4.10: SFG for the cofactor computing

IV.5.4.b Continuous fraction expansion

An intermediate continuous fraction expansion will be developed for the SFG's for the cofactor computing. In the first step, in a analogous way to § IV 4, the following one loop structure will be considered:
The transfer function of SFG of fig 4.11 is:

\[ h_1 = \frac{1}{1 - \frac{d^2C_{12}C_{21}}{p^2}} \tag{4.28} \]

Now a new loop, with a gain of \( \frac{d^2C_{23}C_{32}}{p^2} \) is added, as depicted in figure 4.12:

\[ h_2 = \frac{1}{1 - \frac{d^2C_{12}C_{21}}{p^2} \left( 1 - \frac{d^2C_{23}C_{32}}{p^2} \right)} \tag{4.29} \]

This procedure is continued until the last non eliminated loop is added, resulting in the following continuous fraction, for example 4 remained loop structure:
Chapter IV  A SPECIFIC APPROACH FOR CAPACITANCE COMPUTING

\[ h_4 = \frac{1}{C_{12} C_{21} \frac{d^2}{p^2}} \]
\[ 1 - \frac{1}{C_{23} C_{32} \frac{d^2}{p^2}} \]
\[ 1 - \frac{1}{C_{34} C_{43} \frac{d^2}{p^2}} \]
\[ 1 - \frac{1}{C_{45} C_{54} \frac{d^2}{p^2}} \]

(4.30)

Note that for a structure with one loop less, the lowest part of eq.(4.30) is omitted.

Any part of this formula is known at this stage, because it is totally composed of denominator capacitances. Therefore, equation (4.30) will be rewritten as a standard transfer function, of which the denominator is the interesting cofactor.

IV.5.4.c algorithm for computation of the cofactors

Using (4.30), the lowest fraction will be reduced to the same denominator, resulting in:

\[ h'_4 = \frac{1}{C_{12} C_{21} \frac{d^2}{p^2}} \]
\[ 1 - \frac{1}{C_{23} C_{32} \frac{d^2}{p^2}} \]
\[ 1 - \frac{1}{C_{34} C_{43} \frac{d^2}{p^2}} \]
\[ 1 - \frac{1}{C_{45} C_{54} \frac{d^2}{p^2}} \]

(4.31)

Then the lowest factor "can go up" in the way (...) shown in eq. (4.31), resulting in:
\[ h^3 = \frac{1}{C_{12} C_{21} d^2/p^2} \]

\[ 1 - \frac{C_{23} C_{32} d^2/p^2 \left[ 1 - C_{43} C_{54} d^2/p^2 \right]}{1 - C_{43} C_{54} d^2/p^2 - C_{34} C_{43} d^2/p^2} \]

This can be repeated until a simple fraction is obtained. Its denominator is the searched cofactor.

IV. 6 CONCLUSION

In this part, an algorithm for computing the capacitances has been provided, specific to SDG1 and SDG2. In case of a new synthesis (SDG3 ??) should be adopted, using the same loop structure as for SDG1 and SDG2, the same denominator capacitance computation can still be used. Moreover, the numerator capacitance computing algorithm can be used even if new kind of input branches are added. In this case, only a short and well delimited part of the program must be rewritten.

Some of the capacitances computed here can be negative. These negative capacitances can be transformed into realizable positive capacitances, by using the method described in the next chapter.
CHAPTER V

ELIMINATING NEGATIVE CAPACITANCES
Chapter V

ELIMINATING NEGATIVE CAPACITANCES

V. 1 INTRODUCTION

The previous part described an algorithm for capacitance computing. The sign of these capacitances can be either positive or negative. But the signs of the allowed branches (-C, +dC, -pC and d, with C≥0) are of course fixed. Therefore negative capacitance values must be transformed into positive ones, without any modification of the transfer function. In the first step, the negative capacitances of the input branches will be eliminated. Next the negative capacitances in the loop structure will be eliminated, using a structure modification of the filter. For this purpose a general SDC filter description will be used. Then elimination of the minus signs of capacitances will yield a realizable filter description.

First, the solution will be described in a theoretical way (V.2). Then, a matrix tool for a general SDC filter description will be given in § V.3. Finally in section § V.4 the consequences of the negative capacitance elimination in the matrix filter description will be described.

V. 2 NEGATIVE CAPACITANCE ELIMINATION : THEORY

V.2.1 ELIMINATION OF MINUS SIGN OF CAPACITANCE IN INPUT BRANCHES

The theoretical basis of this work is the fact that the polarity inversion of a node in a flow graph inverts the sign of any branch in contact with this node. If -Vi is used (resp. -V0), then -V0 must be used (resp. -Vi) because the d branch of the op-amp cannot be changed into -d. The minus signs of input capacitances will be eliminated by
inverting the sign of $V_i$ if necessary.

Note that in this way only the sign of one branch per node $V_i^{(e/o)}$ can be chosen. That is neither a problem for the $dC$ branches of the loop structure, nor for the $dC_0$ branch of SOGI1, because they will be corrected as described in the following part. But in the SDG2 synthesis, two branches coming from the input end on the same node, here $V_{out}$. Their signs can not be corrected independently. This implies a restriction for the realizability of the transfer function for SDG2, as announced in chapter III (§ 3.3.2).

V.2.2 ELIMINATION OF NEGATIVE CAPACITANCES IN THE LOOP BRANCHES

Loop structures are composed of $+dC$ branches (and $d$ branches). As described in § 3.3.1, a $-dC_{i,j}$ branch with $C_{i,j}<0$ will be transformed into a $d$ branch and a $-C$ branch, with $C = |C_{i,j}|>0$. The following figure depicts this transformation:

![Diagram of loop structure transformation](image)

**Fig 5.1**: Creation of a $-dC$ branch as a combination of a $d$ branch with a $-C$ branch

This only works if there are no other branches ending on the $V_0$ node of fig 5.1. This method will also be used for $C_0$ if its value is negative. Note that, as depicted in chapter III, § 3.3.3, the $-C_0$ sign is fixed, resulting in a restriction for the realizability of the transfer function, both for SDGI1 and SDG2.
Chapter V  

ELIMINATING NEGATIVE CAPACITANCES

V . 3 MATRIX DESCRIPTION OF A FILTER

The method considered in the previous section uses structure modifications, therefore a tool is required to describe the filter structure. For this aim, a matrix description is used. It must be efficient both for both computation as well as for transfer to other software like SWITCAP which can provide different kinds of simulation.

The structure of a SDG-filter can be described by a set of different types of branches coming from different op amp or from the filter inputs and going to different op-amp outputs. Because there are N+1 nodes, including the input, \((V_0, V_1, \ldots, V_N)\), \(N+1\) order square matrices will be used, one for each types of branch.

There are four basic types of branches: \(d, -dC, -pC, -C\). But in the SFG, each node \(V_i\) is represented by two nodes: one for each clock phase \((V_i^e, V_i^o)\). Therefore, for example, a \(-pC\) branch concerning the odd clock phase will be different from one concerning the even clock phase.

For that reason, eight \(N+1\) order square matrices will describe any filter based on these branches in a general way. They will be called:
- \(1: M(i,j,pCee)\), representing a \(-pC\) branch from \(V_j^e\) to \(V_i^e\). Its value is zero if there is not a branch, otherwise it is the capacitance value of an existing branch.
- \(2: M(i,j,pCoo)\), representing a \(-pC\) branch from \(V_j^o\) to \(V_i^o\).
- \(3: M(i,j,Cee)\), for a \(-C\) branch and even clock phase.
- \(4: M(i,j,Coo)\), for a \(-C\) branch and odd clock phase.
- \(5: M(i,j,dCeo)\), for a \(+dC\) branch from \(V_j^e\) to \(V_i^o\).
- \(6: M(i,j,dCoe)\), for a \(+dC\) branch from \(V_j^o\) to \(V_i^e\).
Chapter V  
ELIMINATING NEGATIVE CAPACITANCES

In the same way, the following matrices are used:
- 7 : $M(i, i, deo)$, representing a d branch from $V_i$ to $V_i$.
- 8 : $M(i, i, doe)$, representing a d branch from $V_i$ to $V_i$.

Note that these two last matrices, $M(i, j, deo)$ and $M(i, j, doe)$, are diagonal matrices, because any d branch represents one op amp with continuous-time feedback, as depicted in chapter III. Therefore these branches, deo and doe, have their noses and tails on the same op amp. For the same reason, elements of the diagonal of $M(i, i, deo)$ can be equal to zero, (no branch), or 1, but nothing else. These two last matrices are not really important because their values are known and won't be changed during the synthesis, but they complete the filter description.

It is assumed here that a positive value in any matrix represents a realizable branch.

This representation can also be convenient when time domain computing is required. This explains the unusual order of the indices in this matrix description, using $M(nose, tail, xy)$ for a branch which ends on $V_{nose}$ and comes from $V_{tail}$, instead of $M(tail, nose, xy)$.

In appendix B, the set of these eight matrix representations will be shown in case of a third order filter, and in the following part, it will be given as an illustration of a negative capacitance elimination.

Note that different methods for a filter description using a matrix representation have been studied.

In an alternative filter description, square matrices of order
Chapter V
ELIMINATING NEGATIVE CAPACITANCES

V.4 ALGORITHM FOR THE NEGATIVE CAPACITANCE ELIMINATION

In the following part the consequences of the negative capacitance elimination on the matrix filter description will be shown.

V.4.1 INITIAL MATRIX FILTER DESCRIPTION

A third order, SDG2 filter has the following structure:

![Diagram of initial structure for a third order, SDG2 filter](image)

Figure 5.2 Initial structure for a third order, SDG2 filter

$2N+2$, $N$ being the filter order, are used. An element of this matrix represents the type of branch from, and to the $2N+2$ nodes $V_0, ..., V_{N}$, $V_0, ..., V_{N}$. This description needs only four matrices, because each one can represent two kinds of branches corresponding with two clock phases. Although the first filter description needs $8$ different $N+1$ order matrices, this second representation hasn't been chosen because the branches between two op-amps $i$ and $j$ are depicted in the four following elements $(i,j)$; $(i,j+N)$; $(i+N,j)$; $(i+N,j+N)$, making the comprehension somewhat more difficult.

- 61 -
Chapter V  

ELIMINATING NEGATIVE CAPACITANCES

This structure (except its d branches) is represented by the following matrices:

<table>
<thead>
<tr>
<th>branch from node:</th>
<th>node to</th>
<th>( V_{in}^0 )</th>
<th>( V_1^0 )</th>
<th>( V_2^0 )</th>
<th>( V_3^0 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( V_{in}^0 )</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>( V_1^0 )</td>
<td>( C_1 )</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>( V_2^0 )</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>( V_3^0 )</td>
<td>( C_3 )</td>
<td>0</td>
<td>0</td>
<td>( C_s )</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>branch from node:</th>
<th>node to</th>
<th>( V_{in}^0 )</th>
<th>( V_1^0 )</th>
<th>( V_2^0 )</th>
<th>( V_3^0 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( V_{in}^e )</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>( V_1^e )</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>( V_2^e )</td>
<td>( C_2 )</td>
<td>( C_{12} )</td>
<td>0</td>
<td>( C_{32} )</td>
<td>0</td>
</tr>
<tr>
<td>( V_3^e )</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

**Figure 5.3.a**: Matrix description for a third order, SDGl filter, Coo and pCo0 matrices

**Figure 5.3.b**: Matrix description for a third order, SDGl filter, dCo0 and dCeo matrices

Note that there is no branch of type \( C^{ee} \) and \( pC^{ee} \), therefore these zero matrices have not been represented.
V.4.2 EXAMPLE OF NEGATIVE CAPACITANCE ELIMINATION

Using the same transfer function as the one used in appendix B yields to the following capacitance signs:

\[
\begin{align*}
  C_0 &< 0 & C_{12} &> 0 & C_5 &> 0 \\
  C_1 &< 0 & C_{21} &< 0 \\
  C_2 &> 0 & C_{23} &> 0 \\
  C_3 &< 0 & C_{32} &< 0
\end{align*}
\]

Figure 5.4: example of capacitance signs

Note that \( C_0 \) and \( C_{3} \) have the same sign, and that \( C_5 \) is positive. Therefore this filter will be realizable.

Because of the minus signs of \( C_0 \) and \( C_{3} \), \( V_3 \) is transformed into \(-V_3\), resulting also in the sign inversion of both \( C_{23} \) and \( C_{32} \) because they are touching this node. This is also done for \( C_1 \), yielding \(-V_1\) instead of \( V_1 \), and inverting the sign of both \( C_{12} \) and \( C_{21} \). Therefore, the minus sign elimination in input branches yields the following capacitance signs:

\[
\begin{align*}
  C'_0 &> 0 & C'_{12} &< 0 & C_5 &> 0 \\
  C'_1 &> 0 & C'_{21} &> 0 \\
  C'_2 &> 0 & C'_{23} &< 0 \\
  C'_3 &> 0 & C'_{32} &> 0
\end{align*}
\]

Figure 5.5: capacitance signs after the elimination of minus signs in input branches

Figure 5.5 shows two remaining negative capacitances, \( C_{12} \) and \( C_{23} \). Therefore, the method described in V.2.2 will be applied. After
verifying that $V^e$ is really equal to d time $V^o$, the transformation described in figure 5.1 goes on the move of $C'^{12}$ value to the Cee matrix, with a sign inversion, transforming in this way the $dC'^{12}$ branch into a $d$ branch and a $-C'^{12}$ branch, with $C'^{12} = (-C'^{12}) > 0$. This is also done for $C'^{23}$, resulting in the structure described in the following matrices (except d branches):

<table>
<thead>
<tr>
<th>branch from node</th>
<th>$V^e_{in}$</th>
<th>$V^e_i$</th>
<th>$V^e_2$</th>
<th>$V^e_3$</th>
<th>branch from node</th>
<th>$V^o_{in}$</th>
<th>$V^o_i$</th>
<th>$V^o_2$</th>
<th>$V^o_3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$V^e_{in}$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>$V^o_{in}$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$V^e_i$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>$V^o_i$</td>
<td>-C$_1$</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$V^e_2$</td>
<td>0</td>
<td>-C$_{12}$</td>
<td>0</td>
<td>0</td>
<td>$V^o_2$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$V^e_3$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>$V^o_3$</td>
<td>-C$_3$</td>
<td>0</td>
<td>-C$_{23}$</td>
<td>C$_5$</td>
</tr>
</tbody>
</table>

$C^{ee}$ branches $C^{oo}$ branches

Figure 5.6.a: filter description after negative capacitance elimination, Cee and Coo matrices

This is not necessary for both SDG1 and SDG2, because this is inherently guaranteed by the synthesis method. However, this verification may be important for a new (SDG3) synthesis.
**Chapter V**  
**ELIMINATING NEGATIVE CAPACITANCES**

<table>
<thead>
<tr>
<th>to node</th>
<th>$V_{i,n}^e$</th>
<th>$V_i^e$</th>
<th>$V_2^e$</th>
<th>$V_3^e$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$V_{i,n}^e$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$V_i^e$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$V_2^e$</td>
<td>$C_2$</td>
<td>0</td>
<td>0</td>
<td>$C_{32}$</td>
</tr>
<tr>
<td>$V_3^e$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

**dCoe branches**

<table>
<thead>
<tr>
<th>branch from node :</th>
<th>$V_{in}^o$</th>
<th>$V_i^o$</th>
<th>$V_2^o$</th>
<th>$V_3^o$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$V_{in}^o$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$V_i^o$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$V_2^o$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$V_3^o$</td>
<td>$-C_0$</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

**dCeo branches**

**Figure 5.6.b**: filter description after negative capacitance elimination, dCoe and dCeo matrices

<table>
<thead>
<tr>
<th>to node</th>
<th>$V_{in}^o$</th>
<th>$V_i^o$</th>
<th>$V_2^o$</th>
<th>$V_3^o$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$V_{in}^o$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$V_i^o$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>$C_{21}$</td>
</tr>
<tr>
<td>$V_2^o$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$V_3^o$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

**pCoo branches**

**Figure 5.6.c**: filter description after negative capacitance elimination, pCoo matrix

Note that there are no branches of type pCee, therefore this zero matrix has not been represented.
The following figure shows the signal flow graph of the filter described in fig 5.6:

![Signal Flow Graph](image)

Figure 5.7: SFG of the SDG2, order 3 filter of fig 5.6, which is the result of negative capacitance elimination

**V.5 CONCLUSION**

This part has shown how negative capacitances are eliminated, resulting in a realizable filter. For this purpose a matrix filter description has been developed.

Although the resulting filter structure is realizable, it will be rescaled. Effectively, the maximum amplitudes of each op amp can be really different. Therefore, in the following chapter, using the last described matrix representation, all op amp output amplitudes will be rescaled, resulting in optimal maximum voltage swings.
CHAPTER VI

VOLTAGE SWING OPTIMIZATION
VI. 1 INTRODUCTION

The maximum voltage amplitude of any op-amp must be the same. Otherwise a very low signal in one node degrades the filter noise over signal ratio. Furthermore, a high input signal level will saturate only the part of the filter having the maximum voltage amplitude. Then the voltage swings of the other parts won't be used, and therefore will be spoilt.

Therefore a voltage swing optimization is required.

The theoretical basis of this optimization given in VI.2 will show that the maxima of the op amp output voltages as a function of frequency are required. After their computations, considered in part VI.3, the algorithm for voltage swing optimization will be given in part VI.4.

VI. 2 THEORETICAL BASIS

The aim of this optimization is that any op amp will "saturate" for the same input filter level. (Note that this can happen for different frequencies). Therefore it consists of having the same maximum voltage for every op amp output, including the filter output. For this purpose, calling Max(V_i) the maximum gain from input to the output of op amp number i before optimization, this op amp output voltage will be multiplied by Max(V_{out})/Max(V_i). Of course, this must be done without any modification in the input to output transfer function. For this purpose, any capacitances connected with the output of op amp number i
will be multiplied by $\alpha = \text{Max}(V_1)/\text{Max}(V_{\text{out}})$, including the continuous feed-back capacitance, which was previously assumed to be equal to 1. Because the charge going through this capacitor doesn't change, its voltage will decrease by a factor of $\alpha$. Therefore, the op amp output voltage is divided by $\alpha$. But at the same time, the other capacitances connected to the output are also multiplied by $\alpha$. Therefore the charge going through these capacitances doesn't change, and there is no modification for the overall filter transfer itself. The following figures show the consequences of this scaling.

![Figure 6.1.a](image)

**Figure 6.1.a**: example of an op amp and its connected capacitances before any modification

![Figure 6.1.b](image)

**Figure 6.1.b**: structure of fig 6.1.a after scaling

Note that all capacitances $C_{i,i+1}, i=1...N$ will be rescaled, and are not equal to 1 anymore.
VI.3 MAXIMUM OP AMP OUTPUT VOLTAGE COMPUTING

The set of capacitance values before scaling is known. Therefore the transfer function from the filter input to each op amp output can be computed. The maximum op amp output voltage computing will be done in two steps: the first step consists of computing the transfer function from input to each op amp output, the second one consist of finding the maximum gain of a given transfer function. Note that these steps have not been included in the available software.

VI.3.1 COMPUTATION OF THE TRANSFER FUNCTION FROM INPUT TO EACH OP AMP OUTPUT

The modifications occurring during the negative capacitances elimination just affect the sign of $V_i$, but never its module. Therefore the initial filter structure, given in figures 3.15 to 3.18, will be used, using the capacitance values computed as described in chapter V. Because the loop structure is the same for the computation of any transfer function, their denominator will be the same, and will be noted as $D_{EN}$. The following notation for the transfer function from input to the output of op amp $i$ will be used:

$$H_i = \frac{V_i}{V_{in}} = \frac{NUM_i}{DEN} \tag{6.1}$$

Note that both $D_{EN}$ and $NUM_{N}$, $N$ being the filter order, are known because they are the input data of the synthesis.

Looking at figure 3.15, $V_n$ depends on $V_{in}$ and on $V_{n-1}$, resulting in the following equation, case of SDG1, odd order:

$$V_n = -C_1V_n^2 + d^2V_n^2 + dC_{N-1,n}V_{n-1} + -pC_{N}V_{in} \tag{6.2}$$
Chapter VI

VOLTAGE SWING OPTIMIZATION

Note that $V_{in}^{o}$ is equal to $d \cdot V_{in}^{e}$, resulting in:

$$V_{N}^{O} = -C_{S}V_{N}^{O} + d^{2}V_{N}^{O} + dC_{N-1,N}V_{N-1}^{O} + -d \cdot p_{C_{N}}V_{in}^{e}$$  \hspace{1cm} (6.3)

Relation (6.3) is divided by $V_{in}^{e}$, resulting in:

$$\frac{V_{N}^{O}}{V_{in}^{e}} = H_{N} = -C_{S}H_{N} + d^{2}H_{N} + dC_{N-1,N}H_{N-1} + -d \cdot p_{C_{N}}$$ \hspace{1cm} (6.4)

Equation (6.3) gives an expression for $H_{N-1}$:

$$H_{N-1} = \frac{1}{dC_{N-1,N}} \cdot ( (1+C_{N} - d^{2})H_{N} + d \cdot p_{C_{N}} )$$ \hspace{1cm} (6.5)

Using (6.1) and (6.5) and multiplying by $DEN$, the following recursive equation is obtained:

$$NUM_{N-1} = \frac{1}{dC_{N-1,N}} \cdot ( (1+C_{N} - d^{2})NUM_{N} + d \cdot p_{C_{N}} \cdot DEN )$$ \hspace{1cm} (6.6)

Any element of the right part of eq. (6.6) is known. Then $NUM_{N-1}$ can be computed. An analogous relation as (6.6) can be written for $NUM_{N-2}$, resulting in:

$$NUM_{N-2} = \frac{1}{dC_{N-2,N-1}} \cdot ( (1-d^{2})NUM_{N-1} + p_{C_{N-1}}DEN - dC_{N,N-1}NUM_{N} )$$ \hspace{1cm} (6.7)

Because $NUM_{N-2}$ has been just computed by using (6.6), $NUM_{N-2}$ can be computed. Note that in this case, no $d$ factor appears with $p_{C_{N-1}}$, because this branch comes directly from $V_{in}^{e}$. 

- 71 -
The general expression is:

\[
\text{NUM}_i = \frac{1}{\text{dC}_{i-1,1}} \cdot ((1-d^2) \cdot \text{NUM}_i + p\text{C}_i \cdot \text{DEN} - d\text{C}_i + 1 \cdot \text{NUM}_{i+1}) \quad (6.8)
\]

\[i \text{ even, } i=2 \ldots N-3.\]

\[
\text{NUM}_i = \frac{1}{\text{dC}_{i-1,1}} \cdot ((1-d^2) \cdot \text{NUM}_i + d \cdot \text{pC}_i \cdot \text{DEN} - d\text{C}_i + 1 \cdot \text{NUM}_{i+1}) \quad (6.9)
\]

\[i \text{ odd, } i=1 \ldots N-2.\]

The same kind of equation can be written for SDG1, even order, and SDG2, giving an algorithm for the transfer function computing.

**VI.3.2 MAXIMUM AMPLITUDE OF OP AMP OUTPUT VOLTAGE**

The previous part enables one to say that the transfer functions from input to each op amp output are known. Therefore their maximum amplitude computation means finding the maximum of these transfer functions. This problem is difficult because of local minima. More studies must be done for the computation of the maximum voltages op amp output, because the very simple algorithm consisting of scanning the transfer function for an important number of frequencies is not correct, because the variation of this function with the frequency can be very important when a high Q filter is used.

**VI.4 SCALING CAPACITANCES FOR VOLTAGE SWING OPTIMIZATION**

The maximum voltage at the op amp outputs, Max(Vi), are supposed to be computed. Then the scaling of capacitances for voltage swing optimization is not difficult.

The theoretical part of this chapter shows that the continuous
feed-back capacitances are involved. To represent their values, a simple vector $[FBC_i]_{i=1..N}$ ("Feed-Back Capacitance") is used. Therefore any capacitance connected with the output of the op amp number $i$, including the continuous feed-back capacitance $FBC_i$, will be multiplied by $\alpha = \text{Max}(V_i)/\text{Max}(V_{out})$. This must be done only for $i=1$ to $N-1$, because the output doesn't change, and the input must not be rescaled.

VI . 5 CONCLUSION

Except the computation of the maximum of the transfer function, this scaling for voltage swing optimization isn't difficult, because it needs only simple modifications in the matrices, describing the filter.
CHAPTER VII

TOTAL CAPACITANCE OPTIMIZATION
In order to reduce the chip surface and its cost, the following part will describe a total capacitance optimization.

**VII. 1 PRINCIPLE**

As shown in chapter III, § 3.2.4, fig. 3.11, the voltage transfer of the filter is determined by the ratio of the capacitances of the input networks and the capacitances of the op amp networks. This gives N degrees of freedom that will be used now to reduce the value of any capacitance without any change of the overall transfer function.

But there is a limit for capacitance value reduction, as depicted in the following part.

**VII. 2 SMALLEST AVAILABLE CAPACITANCE**

Capacitances can not be too small for several reasons, here are a three of them:

- The SDG described here is insensitive to stray capacitances only if the op amps are perfect. Of course, this is unrealizable (because of finite gain and finite bandwidth mainly). Therefore parasitic capacitances must be relatively low. Therefore the value of

9

One part of stray capacitances is proportional to the desired capacitance value. However, the parasitic capacitances due to the
useful capacitances cannot be too small.

- Low values for capacitances means very low current. This can be compared to the use of high value resistances, which means a significant noise contribution. In the same way, it can be shown that the noise of this kind of filter increases when the capacitance values decrease.

- the reproduction of small capacitances is uncertainly, yielding to a bad accuracy.

The use of large capacitances must be avoided because they take a large area, and because they represent a large load for switches and op amps.

The choice of the smallest capacitance value is a trade off. Different choices can be made concerning different parts of the filter, but that is difficult. Therefore the following simple strategy will be used: it will be supposed that no capacitance can be smaller than a unity of capacitance. The absolute value of the unit capacitance can be chosen later. As a consequence, the choice between total capacitance and sensitivity or noise can be made according to the technology used, by the choice of this unit value $^{10}$.

---

switches, interconnections, op amps, etc are not directly related to these desired capacitance values.

10

For the NMOS process used in the Technische Universiteit of Eindhoven, the unity capacitance is about 0.5 pF.
VII . 3 ALGORITHM FOR TOTAL CAPACITANCE OPTIMIZATION

Now, it is assumed that the smallest available capacitance value is equal to one. In order to reduce the total capacitance, or to correct values smaller than one, the following algorithm is used:

First any capacitance connected (eventually through switches) to a given op amp input is scanned, including the continuous feedback capacitance, in order to find the lowest one. Then they are divided by this lowest value. This is done for every op amp input. None of these rescaled capacitance are lower than one unit. The transfer function is not modified because the ratio of continuous feedback capacitance and input network capacitances doesn't change.

Note that after scaling for voltage swing optimization and for total capacitance optimization, there is no available degree of freedom anymore except for the choice of the absolute value of the unit capacitance.

VII . 4 CONCLUSION

Assuming that the lowest available capacitance is 1, it has been shown in this chapter how the total capacitance can be optimized. Now the filter could be realized with these capacitance values. However, in the following chapter another method will be described for further reduction of the total capacitance.

---

In some filter, a certain number of branches doesn't exist. The corresponding capacitances are null. But their computed values are never null, because of round off errors. These very small capacitance values must be replaced in this case by zero.
CHAPTER VIII

CAPACITANCE SHARING
At this step of the synthesis, no degree of freedom for scaling is available. However, here a method will be described for a further reduction of the total capacitance, called capacitance sharing.

VIII. 1 TOTAL CAPACITANCE SHARING

The following figure shows the mode d and e of figure 3.6:

Figure 8.1: Basic stray capacitance insensitive circuits

mode d

Figure 8.2: mode a and mode b is replaced by a mode e

But there is no reason why two branches having the same structure as depicted in fig.8.2 also have the same capacitance value.
Chapter VIII
CAPACITANCE SHARING

The following solution can be used:

The values of these two different capacitances can be corrected, using the same degree of freedom for voltage optimization (mode e), or for total capacitance minimization (mode f), without any modification of the whole transfer function of the filter. After correction, the capacitance values will be equal, and the capacitance sharing will be possible.

Total capacitance sharing yields sometimes a trade off between total capacitance and voltage swing. This choice can be made only by the user, requiring an interactive program.

Note that it can be shown that the total capacitance sharing increases the accuracy of the filter.

VIII . 2 PARTIAL CAPACITANCE SHARING

There is an alternative solution when the values of two branches, -C and dC', having the right topology, are large and when their difference is big, say for example C=4 units and dC'=7 units, and a total of 11 units.

The total capacitance sharing described in § VIII.1 can yield an unacceptable reduction of the voltage swing.

In this case, two networks will be used. First a mode e or f with a capacitance of 4 units will be used. Then the complement for the dC' branch will be made with this kind of branch, with C''= 7 - 4 = 3 units. Therefore the total capacitance is 4 + 3 = 7, that is to say 4 units less than before partial capacitance sharing.

Note that this methods doesn't reduce the voltage swing. The number of switches will increase, but the area used by a switches is

- 80 -
much lower than the obtained area reduction. But, after rescaling as described in chapter VII, this method yields sometimes a larger total capacitance!

In conclusion, these methods can be very interesting, but cannot be done automatically, an interactive program will be necessary. Note that this step is not included in the available software.
Almost every part of the initial specifications is available:

- Software corresponding to the first step, the capacitance computing, is available for both synthesis methods. This set of tools can be adapted to an eventually future synthesis based on stray capacitance insensitive design graphs having the same loop structure.
- The structure modification, step 2, has been implemented successfully.
- One part of the voltage swing optimization step is included in the software, concerning the capacitance scaling. Algorithms for the computation of the input to each amp amp output transfer function is given in this report. Finding their maximum will be a difficult procedure. But writing this part doesn't require to understand any part of the whole software.
- Total capacitance optimization is available.
- The last step, capacitance sharing, requires an important interaction with the user, but the basic operations are not difficult, because two procedures of step 2 and step 3 can be used. This part is not implemented.

In conclusion, this training period gave technical knowledge about switched capacitor filters, and a very interesting experience abroad.
COST ESTIMATION

The available software, and the present report required a (student) engineer during almost three months, (the help of the coach is not included here).

No costly tools have been used, except an IBM model 50 during three quarters of the training period.
APPENDIX A

SOFTWARE DESCRIPTION
A.1 GENERAL DESCRIPTION

The program has been realized with the PASCAL language, with "TURBO PASCAL VERSION 5". Many improvements are provided by turbo pascal 5 compared to standard pascal. One of the improvements is the possibility to use "Units", which enable a structured presentation of the software: in the main program, only the important steps are described, each step is in fact a procedure of which it is not necessary at the beginning to understand how it works, but only what it does.

In this chapter, the name of the procedure or function will be given together with its description and with the name of the unit where it can be found, like:

procedure_name/Unit_name

The following figure describes the organization of the different units:

Figure A.1: Organization of the different units
appendix A

SOFTWARE DESCRIPTION

The main program uses "only" procedures or "black boxes":
- given a transfer function, is SDG1 or SDG2 possible?
  synthesis_is_possible/filt_II
- given a transfer function and the type of synthesis,
  compute filter capacitance/filt_II
yields the filter with its structure and the capacitance values, after several optimization step. Note that the capacitance sharing optimization is not implemented.

-UNIT FILT_II uses units filt_III and filt_II2 as depicted in figure A.1.
-Unit filt_III provides several tools for the algebraic capacitance computation, described in chapter IV.
-Unit filt_II2 gives different tools for the other and following steps. Note that in the first unit filt_III, the matrix filter description is not required.

A.2 STEP 1: CAPACITANCE COMPUTING

The theoretical aspects are depicted in chapter 4.
Here is the description of the main part of Unit II.
This main part uses tools from Unit_III.

STEP 1 PART 1: POLYNOMIAL TRANSFORMATION

Theoretical aspect: § 4.3
The polynomials corresponding to eq.4.1 or 4.2 are called dpDEN and dpNUM, standing for polynomials for the denominator and for the nume-
rator, expressed in $d\ (-z^{-1/2})$ and $p\ (1-d^2)$.

Two procedures are used:
- $d\_to\_dp\_type1/unit\_II1$ gives the translation of the initial transfer function polynomial (numerator or denominator) into the polynomial of eq (4.1).
- $d\_to\_dp\_type2/unit\_II1$ does the same kind of work and provides polynomials of the second type, like the numerator of eq. (4.2).

These procedures use the following procedure:
- $d\_to\_pPoly\_convert/unit\_II1$
which does the first step described in § 4.3.2, $P_1$ being $dPoly$, $P_2$ being $pPoly$.

The recursive procedure is called CHANGE.

**STEP 1 PART 2 : COMPUTATION DENOMINATOR CAPACITANCES**

Theoretical aspect : § 4.4

**initialization**:
- $P_1$ is the initial $d/p$ denominator polynomial of eq.(4.19)
- $P_2$ is the polynomial of eq (4.20)

The step indicated in eq (4.21) is done in the "for to" statement.
The capacitances of the denominator are $BDen\_cap$ for Backward DENominator CAPacitance : $BDen\_cap[i]= C_{i+1,1}$. $BDen\_cap[N]= C_s$. (The forward capacitances $C_{i,1+1}$ are assumed to be 1 at this step of the synthesis)

**step 1 PART 3 : COMPUTE NUMERATOR CAPACITANCES**

Theoretical aspect : § 4.5
- 3.1 Initialization : the Matrix of eq (4.24), called $MAT$ in the program is initialized to zero.
- 3.2 the matrix $MAT$ is filled in, using :
  
  cofactor/unit\_II1 which yields the cofactor polynomial of a path.

The cofactor computation is explained in more details in chapter
IV.5.4.
- 3.3 the equation 4.24 is resolved, giving the numerator capacitances, using a standard back substitution resolution (see § 4.5.3)
  
  *resolve triangular/Unit_III*

The numerator capacitances are \( \text{NUM\_CAP} : \text{NUM\_CAP}[1] = C_1 \).

Note that this part (step 1 part 3.3) depends on the synthesis \((\text{SDG}/2)\) (see 4.5.3), mainly because of the difference between the capacitance vector in eq.\((4.27.a)\) and \((4.27.b)\).

\( \text{CSDG}/2 \) of eq \((4.27.a/b)\) are called Xpoly, standing for unknown polynomial.

A . 3 STEP 2 : NEGATIVE CAPACITANCES ELIMINATION

The theoretical aspects are depicted in chapter 5. Unit_II.

Here is the description of the main part of Unit_II2.

This main part uses tools from chapter 5.

step 2 part.1 initialization: generate the filter description

The matrix representation of the filter (fig 5.3) is called \( \text{M\_filter} \), which stand for \( \text{Matrix\_filter} \). It is used in the following way: \( \text{M\_filter}[\text{nose, tail, branch\_kind}] \); nose is the nose of a branch, tail is the tail of a branch, branch\_kind is the type of branch, that is to say a type\_branch\_kind \( \rightarrow \)\( (\text{Cee, pCee, Coo, pCoo, doe, dCoe, deo, dCeo}) \);
The branches concerning the denominator capacitances are included in the matrix. This part is correct for both SDGI and SDG2. The following procedure is used:

*Fill_in_denominator/Unit_I12.*

Branches concerning the numerator are included in the matrix. This part is of course specific to each synthesis. That is why a CASE filter_synthesis of SDGI: ; SDG2: ; end; statement is used.

When debugging, it may be interesting to have an output of the matrix filter description. The following procedure:

*Write_the_matrix/Unit_I12*

provides an output like fig.5.3.

**Step 2 part 2 negative capacitance elimination**

This is done by the procedure:

*remove_neg_cap/Unit_I12*

Looking at the procedure remove_neg_cap in Unit_I12, the following steps are used:

1: minus signs in input branches are eliminated, as depicted in § V.4.2, by inverting the signs of corresponding nodes.

2: minus sign in dC branches are eliminated with a topology modification, as described in § V.4.2.

There is a small difference between SDGI and SDG2. The two branches -dC0 and -pC1 both have their noses on V1. The first step will correct the sign of -pC1 if necessary. Although it is an input branch, the eventually minus sign of -dC0 will be corrected in the second step.
A . 4 STEP 3 : VOLTAGE SWING OPTIMIZATION

The theoretical aspects are depicted in chapter 6. Here is the description of the main part of this main part uses tools from Unit II and Unit II2.

STEP 3 part 1 Computation of the transfer function from input to each op amp output.

The algorithm for this part is described in chapter 6 (§ 6.3.1). This part has not yet been implemented.

STEP 3 part 2 : find the maximum of a given transfer function

This part is not yet available. A simulation was done for some examples, where the maximum amplitudes were found in another way, using a simulator and finding on the graphic the maximum output amplitude by inspection.

STEP 3 part 3 : rescaling

This part used the following procedure for each op amp output, except the output:

\[ \text{rescale_output/unit_{II2}} \]

as described in chapter 6 (§ 6.2). Note that the scaling factor is

\[ \alpha = \frac{\text{Vmax[op_amp]}}{\text{Vmax[filter_order]}} \]
A . 5 STEP 4: TOTAL CAPACITANCE OPTIMIZATION

The theoretical aspects are depicted in chapter 7.
Here is the description of the main part of Unit II.
This main part uses tools from Unit II2.

First the lowest capacitances connected to an input amplifier is found,

\[ \text{lowest}_\text{cap}/\text{unit}_{II2} \]

Note that in some filters, a certain number of input branches doesn't exist. The corresponding capacitances are null. But their computed values are never null, because of round off errors. These very small capacitance values corresponding to non existing branches must be replaced in this case by zero (see \( \text{lowest}_\text{cap}/\text{Unit}_{II2} \)).

Second, the scaling is made using :

\[ \text{total}_\text{cap}_\text{scale}/\text{unit}_{II2} \]

A . 6 STEP 5: CAPACITANCE SHARING

The theoretical aspect are depicted in chapter 8.
Here is the description of the main part of Unit II.

This part will be interactive with the user.
The software has been used for several synthesis examples, in order to test it. Here are the results of two synthesis examples, one of order 3, and one of order 8.

B . I SDG1, ORDER 3 FILTER SYNTHESIS

The following figure shows the amplitude transfer function of a special filter.

![Amplitude transfer function of the desired filter](image)

Figure B 1 : Amplitude transfer function of the desired filter

The obtained transfer function is:

\[
H(d) = \frac{V_{out}}{V_{in}} = -1.529 + 0.984d^2 - 1.641d^4 + 0.749d^6 \\
-2.618 + 0.853d^2 - 0.678d^4 + d^6 \quad \text{(B.1)}
\]

and has the corresponding amplitude diagram drawn in fig.B.2.
Eq (B.1) is used as the starting point for the realized software. Equation (B.1) is rewritten into eq.(B.2):

\[
H = \frac{0.749 + \frac{1}{p}0.780 + \frac{d^2}{p}1.386 + \frac{d^3}{p}1.437}{1 + \frac{1}{p}1.618 + \frac{d^2}{p}3.940 + \frac{d^3}{p}1.443}
\]  

(B.2)

The denominator capacitances are then computed, resulting in:

\begin{align*}
C_{d1} &= C_{d3} = 1 \\
C_{d1} &= -0.892 \\
C_{d2} &= -0.892 \\
C_{d3} &= -0.749
\end{align*}

Figure B.3: Denominator capacitance values

Then the numerator capacitances are computed:

\begin{align*}
C_0 &= 1.437 ; C_1 = -0.718 ; C_2 = -0.780 ; C_3 = -0.749
\end{align*}

The fact that \( C_1, C_2 \) and \( C_3 \) are negative means that \(-V_1, -V_2, -V_3\) must be realized, instead of \( V_1, V_2, \) and \( V_3 \). Because the sign of \( V_1 \) is changed, the sign of \( C_0 \) must also be changed.
Appendix B  

SYNTHESIS EXAMPLES

Now there are three negative capacitances: \( C_0 \), \( C_{21} \) and \( C_{32} \). Their signs is changed by realizing \(-dC\) as \( d*(-C)\) as depicted in chapter III.

The following topology is obtained:

![Diagram of realizable SFG for the transfer function of eq.(B.1)](image)

Figure B.4 : Realizable SFG for the transfer function of eq.(B.1)

Note that this topology is described by 8 matrices, as depicted in chapter V.

The two steps concerning voltage swing optimization and total capacitance optimization will not be described here.

This filter has been successfully tested with SWITCAP.

B.  SDG1, 8-TH BAND STOP FILTER SYNTHESIS

The desired specification are:
A possible transfer function is given by the following 8th order elliptical filter:

\[
\text{Numerator} = 0.269 - 0.756d^2 + 1.717d^4 - 2.412d^6 + 2.891d^8 - 2.412d^{10} \\
+ 1.717d^{12} - 0.756d^{14} + 0.269d^{16}
\]

\[
\text{Denominator} = 0.123 - 0.310d^2 + 0.730d^4 - 1.487d^6 + 2.455d^8 - 2.609d^{10} \\
+ 2.557d^{12} - 1.930d^{14} + d^{16}
\] (B.3)

The coefficients of eq (B.3) were used as a starting point for the synthesis program. The previous polynomial is transformed as described in chapter IV, resulting in the following polynomial:
Then the capacitances are computed:

\[
\begin{align*}
C_{65} &= -0.629 & C_{21} &= -0.488 & C_0 &= -2.187 \\
C_{56} &= 1.000 & C_{12} &= 1.000 & C_1 &= 0.638 \\
C_{76} &= -0.771 & C_{32} &= -0.221 & C_2 &= 0.000 \\
C_{67} &= 1.000 & C_{23} &= 1.000 & C_3 &= 0.302 \\
C_{87} &= -0.450 & C_{43} &= -1.655 & C_4 &= 0.000 \\
C_{78} &= 1.000 & C_{34} &= 1.000 & C_5 &= 2.901 \\
C_s &= 7.117 & C_{54} &= -0.389 & C_6 &= 0.000 \\
& & C_{45} &= 1.000 & C_7 &= 2.286 \\
& & & C_8 &= -0.000
\end{align*}
\]

Fig B.7: Capacitance values
The negative capacitances are eliminated, voltage swing and total capacitance optimization steps are done.

As a verification, a simulation (SWITCAP) was done. The following figure shows the amplitude transfer function of eq (B.3). The difference between this figure and the simulation result can not be seen in this figure.
LINEAR MAGNITUDE PLOT

Gain vs. Hertz graph

- Gain ranges from 0.000 to 1.000
- Hertz ranges from 10.00 to 500

Graph shows a sharp increase at around 300 Hz.
Appendix C

SOFTWARE : UNIT II.PAS

C . 1 : UNIT FILT_II

(******************************************************************************)
UNIT filt_II;
(******************************************************************************)
(* input is - transfer function, a d polynomial fraction.
- kind of topology : SDG1, SDG2, ...
output is the description of the topology with only positive capacitances. That is to say that the output is a realizable filter, capacitance are also rescaled here.

use filter unit where there are many tools for this part, you must see at least the interface of this unit. *)

{$define debug} (define for debugging help )
(* {$define adderror} *)( for debugging part of the pgm when a negative cap can't be removed )
{$N+} (co processor is used)

(******************************************************************************)
INTERFACE
(******************************************************************************)
uses filt_III,filt_II2;

FUNCTION SYNTHESIS IS POSSIBLE(filter_order:indice;
filter_synthesis:type_filter_synthesis;
dNum,dDen:type_poly
):boolean;
(* is true when the synthesis is possible *)

PROCEDURE COMPUTE_FILTER_CAPACITANCES(filter_order:indice;
filter_synthesis:type_filter_synthesis;
dNum,dDen:type_poly;
var M_filter:filter_description_matrix;
var amp_cap:type_poly;
var ff:text (output file for debugging)
);
IMPLEMENTATION

FUNCTION SYNTHESIS_IS_POSSIBLE(filter_order: indice;
    filter_synthesis: type_filter_synthesis;
    dNum, dDen: type_poly
): boolean;

(* is true when the synthesis is possible *)
var zero_product, pole_product: real;
begin
    pole_product := dDen[filter_order]/dDen[0];
    zero_product := dNum[filter_order]/dNum[0];
    if odd(filter_order) then begin
        pole_product := -pole_product;
        zero_product := -zero_product;
    end;

    case filter_synthesis of
    SDG1: synthesis_is_possible := (pole_product > 0) and (pole_product <= 1);
    SDG2: synthesis_is_possible := (pole_product > 0) and (pole_product <= 1) and
        (zero_product >= 0) and (zero_product <= 1);
    end; (of case of statement )
end; (of function synthesis_is_possible)
PROCEDURE COMPUTE_FILTER_CAPACITANCES(filter_order:indice;
    filter_synthesis:type_filter_synthesis;
    dNum,dDen:type_poly;
    var M_filter:filter_description_matrix;
    var amp_cap:type_poly;
    var ff:text (output file for debugging)
    );

VAR
  {var used in different part : }
  count,countl,i,j,Ci:integer;

  {var for computing d and 1/p transfer function polynomials : }
  dpDen:type_poly; (d and 1/p DENominator polynomial )
  dpNum:type_poly; ( " NUMerator " )

  {var for computing denominator capacitances : }
  BDen_cap:type_poly;(DENominator cap :C2l=BDen_cap[1]...
  { Cfilter_order,filter_order-1=NUM_cap[filter_order];
    Cs=NUM_cap[order])
  P1,P2,Ptemp:type_poly;

  {var for computing numerator capacitances : }
  Num_cap:type_poly;(NUMerator capacitances:C_0=num_cap[0] to
    C_filter_order}
  dpcofactor,dpfactor,Xpoly:type_poly;
  Mat : max_filter_order_matrix;
  sig:real;

  {var for computing on matrix filter description)
  branch_kind:type_branch_kind;
  nose,tail:indice;
  error_string:string;

  {var for scaling capacitance)
  { amp_cap:type_poly; }
  Vmax:type_poly;
  op_amp:indice;
  low_cap:real;

(********************************************************************)
begin (*MAIN PROGRAM *)
(********************************************************************)
(********************************************************************)
(* STEP 1 *)
(* PART 1 *)
(* change fraction in power of d into fraction in power of 1/p and d *)
(********************************************************************)

(* 1.1.1 : convert the numerator *)
(*-------------------------------*)
case filter_synthesis of
  SDG1:
    if odd(filter_order) then d_to_dp_type1(filter_order,dNum,dpNum)
    else d_to_dp_type2(filter_order,dNum,dpNum);
  SDG2:
    d_to_dp_type1(filter_order,dNum,dpNum);
end;

(* 1.1.2 : convert the denominator *)
(*-------------------------------*)
d_to_dp_type1(filter_order,dDen,dpDen);

(* 1.1.3 : rewrite dpNum and dpDen in order
  that dpDen=1 +a1/p + ...) *)
(*-------------------------------*)
for i:=filter_order downto 0 do
begin
  dpNum[i]:=dpNum[i]/dpDen[0];
  dpDen[i]:=dpDen[i]/dpDen[0];
end;

writeln(ff);
writeln(ff,blk,'Intermediate information : ');
writeln(ff,blk,'d^2 and 1/p tranformed "polynomial" : ');
for i:=0 to filter_order do
  writeln(ff,blk,'dpNum',i:1,' : ',dpNum[i]:5:3);
writeln(ff);
for i:=0 to Filter_order do
  writeln(ff,blk,'dpDen',i:1,' : ',dpDen[i]:5:3);
Appendix C

SOFTWARE : UNIT II.PAS

(*****************************************************************)
(* STEP 1 *)
(* PART 2 *)
(*
compute denominator capacitances *)
(*****************************************************************)

(* 1.2.1 init P1 and P2 *)
(*--------------------*)
P1:=dpDen; \{ denominator polynomial in d and 1/p \}
P2:=nul_poly; \(=0\)
For count:=0 to filter order do
if not odd\{count\) then P2[count]:=P1[count];
BDen_cap:=nul_poly; \(=0\)

(* 1.2.2 compute Cs, C\{filter\_order\}, C\{filter\_order\-1\} *)
(*--------------------------_:_----------_:_---------*)
for count:=0 to filter_order-1 do
begin
(* P1:=(P1-P2)*)
for count1:=0 to filter_order do
P1[count1]:=P1[count1]-P2[count1];
(* P1:=left shift of P1 until P1[0]<>0 *)
repeat
for count1:=1 to filter_order do
for count1:=0 to filter_order do
P1[count1]:=P1[count1];
P1[filter_order]:=0.0;
until P1[0]<>0;
(* P1[0] is the capacitance value)
BDen_cap[filter_order-count]:=P1[0];
(* P1:=P1/P1[0]*)
for count1:=filter_order downto 0 do
P1[count1]:=P1[count1]/P1[0];
(* P2 <-> P1*)
Ptemp:=P1; P1:=P2; P2:=Ptemp;
end;
BDen_cap[filter_order]:=BDen_cap[filter_order];
\{-Cs instead of +Ci,i-1\}
write\(ln\)(ff);write\(ln\)(ff,blk,\{‘denominator capacitances’\});
for count1:=0 to filter_order-1 do
begin
write\(ln\)(ff,blk,\{‘C’,count1+1:1,count1:1,’ = ‘,BDen_cap[count1]:6:3\});
write\(ln\)(ff,blk,\{‘C’,count1:1,count1+1:1,’ = ‘,1.0:6:3\});
end;
write\(ln\)(ff,blk,\{‘Cs = ‘,BDen_cap[filter_order]:6:3\});
SOFTWARE:
UNIT II.PAS

(* ********************** ****************************************** *)
(* STEP 1 *)
(* PART 3 *)
(* compute numerator capacitances *)
(* ********************** ****************************************** *)

(* 1.3.1 init *)
(*------------------------*)
for i:-0 to max_filter_order do
  for j:-0 to max_filter_order do
    mat[i,j]:=0;
Num_cap:=nul_poly; (=0)
dpcofactor:=nul_poly; (=0)
dpfactor:=nul_poly; (=0)

(* 1.3.2 fill in the matrix and resolve the numerator capacitances *)
(*------------------------*)
(* The numerator of the transfer function depends on
  'numerator capacitances' (C_0 to C_filter_order)
in a linear way, that is to say that we can write :
  MAT*C=Num with :
  * C is a vector built with C0 to C_filter_order
  * Num is the d and p numerator polynomial, more precise,
    it's a vector made with the polynomial coefficients
    dpNum[0] : constant coefficient
    dpNum[1] : coefficient of 1/p
    dpNum[2] : coefficient of d^2/p^2 ...
  * Mat is filter_order by filter_order matrix, that coefficients
    are combination of denominator capacitances according to masson rules *)

case filter_synthesis of
  SDG1:
  (****)
  begin
  (* fill the matrix *)
  (* C vector = [C_filter_order,.. C_1,C_0] then if filter_order=4
  [* Mat[i,0]*C_4 + Mat[i,1]*C_3 + Mat[i,2]*C_2 + mat[i,3]*C_1 +
    Mat[i,4]*C_0= 
  [* C_0 ]
  Mat[filter_order,filter_order]:=-1;
  [* C_1,... C_order ]
  for j:-1 to filter_order do
  begin
  (* the path going though C_j touch loop
  (* number j-1,j... filter_order-1 */
  cofactor(j-2,BDen_cap,dpcofactor);
  (* multiple cofactor with the path product: * )
  [* (-1)* d'(don't care)/p^filter_order-j ]
  dpfactor:=nul_poly; (=0)

  - 106 -
for count := filter_order-j to filter_order do
  dpfactor[count] := -1 * dpcofactor[count-(filter_order-j)];
(* put dpfactor in the column of Mat number (filter_order-j) *)
for count := 0 to filter_order do
  mat[count,filter_order-j] := dpfactor[count];
end;
(* resolve the system: find numerator capacitance *)
(* for SDG1 we have here a lower triangular matrix *)
resolve_triangular(filter_order,Mat,dpNum,Xpoly);
(* now, C_0-Xpoly[filter_order,... C_filter_order-Xpoly[0] *)
for i := 0 to filter_order do
  num_cap[i] := Xpoly[filter_order-i];
end; (* of computing numerator capacitance in case SDG1 *)

SDG2 :
(*****)
begin
(* fill the matrix *)
(* C vector = [C_filter_order, .. C_2,C_0] then if filter_order=4
(* Mat[i,0]*C_0 + Mat[i,1]*C_4 + Mat[i,2]*C_3 +
  mat[i,3]*C_2 + Mat[i,4]*C_1 =
  dpNum[i] = coefficient in 1/p^1 *)

(* C_0 *)
(* the cofactor of the path going through C_0 touch loop
(* number filter_order-1 *)
cofactor(filter_order-2,BDen_cap,dpcofactor);
(* multiple cofactor with the path product: -1 *)
dpfactor := nul_poly; (=0)
for count := 0 to filter_order do
  dpfactor[count] := -1 * dpcofactor[count];
(* put dpfactor in the column of Mat number 0 *)
for count := 0 to filter_order do
  mat[count,0] := dpfactor[count];
(* C_1,...C_filter_order *)
for j := 1 to filter_order do
begin
(* the cofactor of the path going through C_j touch loop
(* number j-1,j,... filter_order-1 *)
cofactor(j-2,BDen_cap,dpcofactor);
(* multiple cofactor with the path product: *)
(* ((-1)^(j+filter_order))*d^(don't care)/p^(filter_order-j+1))
dpfactor := nul_poly; (=0)
if odd(j+filter_order) then sig := -1.0 else sig := 1.0;
for count := filter_order-j to filter_order do
  dpfactor[count] := sig * dpcofactor[count-(filter_order-j+1)];
(* put dpfactor in the column of Mat number (filter_order-j+1) *)
for count := 0 to filter_order do
  mat[count,filter_order-j+1] := dpfactor[count];
end;

{* resolve the system : find numerator capacitance *)
{* for SDG2 we have here a lower triangular matrix *)
{* resolve Mat*Xpoly=dpNum *)
resolve_triangular(filter_order,Mat,dpNum,Xpoly);
(* now, C_0-Xpoly[0] but C_i-Xpoly[filter_order-i+1] *)
Num_cap[0]:=Xpoly[0];
  for i:=l to filter_order do num_cap[i]:=Xpoly[filter_order-i+1];
end; {* of computing numerator capacitance in case SDG2 *)

end; {* of computing numerator in any case }

(* write numerator capacitances *)
writeln(ff);writeln(ff,blk,'numerator capacitances');
for count:=0 to filter_order do
  writeln(ff,blk,'C',count:l,': ',Num_cap[count]:6:3);

- 108 -
SOFTWARE: UNIT II.PAS

(******************************************************************)
(* STEP 2 *)
(* PART 1 *)
(******************************************************************)
(* generate the filter description *)
(******************************************************************)

(* 2.1.1 initialization *)
(*----------------------*)
for branch_kind:=Cee to dCeo do {all branch kink}
  for tail:=0 to filter_order do
    for nose:=0 to filter_order do
      M_filter[nose,tail,branch_kind]:=0.0;

(* 2.1.2 generate branches concerning the denominator (loops) *)
(*------------------------------------------------------------*)
fill_in_denominator(filter_order,BDen_cap,M_filter);

(* 2.1.3 generate branches concerning the numerator *)
(*--------------------------------------------------*)
case filter_synthesis of
  SDG1:begin
    {* d branch from Vin e to Vin o } 
    M_filter[0,0,deo]:=1.0;
    {* dC0 branch } 
    M_filter[1,0,dCeo]:=Num_cap[0];
    {* -pC branch from Vin e and from Vin o } 
    count:=1;
    repeat
      M_filter[count,0,pCoo]:=Num_cap[count];
      M_filter[count+1,0,pCee]:=Num_cap[count+1];
      count:=count+2;
    until count>-filter_order+l;
  end; {of case SDG1}
  SDG2:begin
    if odd(filter_order)
      then begin
        {* -pC0 branch } 
        M_filter[filter_order,0,pCoo]:=Num_cap[0];
        {* -C and dC branches from Vin o } 
        count:=1;
        repeat
          M_filter[count,0,Coo]:=Num_cap[count];
          M_filter[count+1,0,dCoe]:=Num_cap[count+1];
          count:=count+2;
        until count>-filter_order+l;
      end (of case SDG2, filter-order is odd)
    else begin
      {* -pC0 branch } 
      M_filter[filter_order,0,pCee]:=Num_cap[0];
      {* -C and dC branches from Vin o } 
      count:=1;
    end (of case SDG2, filter-order is even)
repeat
    M_filter[count,0,dCeo]:=-Num_cap[count];
    M_filter[count+1,0,Cee]:=Num_cap[count+1];
    count:=count+2;
    until count>=filter_order+1;
end; {of case SDG2,filter_order isn't odd}
end; {of case SDG2}
end; {of case of statement}

{add an error}
{ifdef adderror}
(Cs <- -Cs)
M_filter[filter_order,filter_order,Coo]:= -M_filter[filter_order,filter_order,Coo];
{endif}
{ifdef debug}
    writeln(ff,blk,'matrix before the topology modifications :');
    writeln(ff);
    write_the_matrix(filter_order,M_filter,ff);
    writeln(ff);writeln(ff);
{endif}
(********************************************************************)
(* STEP 3 *)
(* eliminate negative capacitances *)
(********************************************************************)
remove_neg_cap(filter_order,filter_synthesis,M_filter,error_string);

writeln(ff);
writeln(ff,blk,error_string);
writeln(ff,blk,' in removing negative capacitances ');

- 111 -
Appendix C

SOFTWARE : UNIT II.PAS

(********************************************************************)
(* step 4 *)
(* swing optimization *)
(********************************************************************)

(* 4. 1 compute transfer function from input to every op amp output *)
(*not done *)

(* 4. 2 compute maximum output value for every op_amp output *)
(* here we have only a simulation *)
Vmax:=nul_poly; (=0)
Vmax[1]:=3.035 ; Vmax[2]:=2.399 ; Vmax[3]:=1.075 ;Vmax[4]:=2.0;
Vmax[5]:=2.23;

(* 4. 3 rescale op amp output voltage with Vmax[filter_order]*)
(*-----------------------------------------------------------*)
amp_cap:=nul_poly;
for op_amp:=1 to filter_order do
  amp_cap[op_amp]:=1.0;
for op_amp:=1 to filter_order do
  rescale_output(filter_order,op_amp,Vmax[op_amp]/Vmax[filter_order],
                 M_filter,amp_cap);
for op_amp:=1 to filter_order do
    begin
        low_cap:=lowest_cap(filter_order, op_amp, M_filter, amp_cap);
        total_cap_scale(filter_order, op_amp, 1/low_cap, M_filter, amp_cap)
    end;

end; (of the procedure compute_filter_capacitances)
C.2 UNIT FILT_ILI

(*************************************************************************)
unit filt_III;
(*************************************************************************)
( this unit is used by filt_II : filter part II )

($N+) (co processor is used)
(define debug)

 ifdef debug
  ($D+,R+,S+) {debug information ; rang checking ; stack checking}
 else
  ($D-,R-,S-)
 endif

(*************************************************************************)
interface
(*************************************************************************)

const max_filter_order-10;
  blk=' '; ('BLacK': for ouput printing)
type indice=0..max_filter_order; { for out of range verification }
type_poly=array[indice] of real; { out of range verification }
max_filter_order_matrix=array[indice,indice] of real;
string5=string[5];
Var iiiii:indice;(not important, used in the unit initialization part)
nul_poly:type_poly; (nil polynomial )

PROCEDURE D_TO_DP_TYPE1(order:indice;
  dPoly:type_poly;var dpPoly:type_poly);
(* convert a standard polynomial dPoly like equation 5.6 into the
  special polynomial dpPoly like equation 5.7 :
  dPoly(d^2)= dPoly[0]+dPoly[1]*d^2+..+ dPoly[order]*d^2*order to
  dpPoly = dpPoly[0] + dpPoly[1]/p + d^2* (dpPoly[2]/p^2
  +dpPoly[3]/p^3+ d^2*( ...dpPoly[order-1]/p^order-1
  + dpPoly[order]/p^order))
  with p=1-d^2 and order the order of dPoly(d^2) *)

PROCEDURE D_TO_DP_TYPE2(order:indice;
  dPoly:type_poly;var dpPoly:type_poly);
(* convert a standard polynomial dPoly like equation 5.6 into the
  special polynomial dpPoly like equation 5.7 in case of SDGl,
Appendix C

SOFTWARE : FILT III.PAS

even order:
\[ \text{dPoly}(d^2) = \text{dPoly}[0] + \text{dPoly}[1] \cdot d^2 + \ldots + \text{dPoly}[\text{order}] \cdot d^2 \cdot \text{order} \]
\[ \text{dpPoly} = \text{dpPoly}[0] + (d^2 \cdot \text{dpPoly}[1]/p + \text{dpPoly}[2]/p^2 + d^2 \cdot (\ldots \text{dpPoly}[\text{order}-1]/p^{\text{order}-1} + \text{dpPoly}[\text{order}]/p^{\text{order}})) \]
with \( p=1-d^2 \) and order the order of \( \text{dPoly}(d^2) \)

**PROCEDURE** COFACTOR(order: indice; BDen_cap:type_poly;
var dpcofactor:type_poly);
(* give the d&p cofactor polynomial when a path is not touching
  loop number 1 to order. The polynomial is in d&p, p in the denominator,
  and have only even power of p *)
(* order can be negative or nul, in this case the cofactor is 1 *)

**PROCEDURE** RESOLVE_TRIANGULAR(order: indice; Mat:max_filter_order_matrix;
poly:type_poly;var Xpoly:type_poly);
(* resolve Mat * Xpoly = poly, when Mat is lower triangular *)
(* order is the the order of the actually used matrix *)

- 115 -
uses crt;

(* warning : there is initialization part :
(* 1: Nul_poly is a max_filter_order polynomial equal to zero   *)*

PROCEDURE D_TO_PPOLY_CONVERT(order:indice;dPoly:type_poly;
var pPoly:type_poly);
(* dPoly(d^2) is transform into pPoly(p=1-d^2) *)
(* used by d_to_dptype1 and d_to_ptype2 *)
var
    i,j:indice;
    den:real;
begin
    pPoly:-nul_poly;
    for i:-O to order do
    begin
        if i-O then den:=1 else den:=den*(i);(den-i)
        for j:-O to order do
            pPoly[i]:=pPoly[i]+dPoly[j]/den;
    (* compute d/dp (dPoly) *)
    if order-i-1>-O then
        for j:-O to order-i-1 do
            dPoly[j]:=-dPoly[j+1]*(j+1));
    dPoly[order-i]:=0.0;
    end;
end; (of procedure d_to_pPoly_convert)

(*---------------------------------------------------------------*)
PROCEDURE D_TO_DP_TYPE1(order: indice;
     dPoly:type_poly;var dpPoly:type_poly);
(*---------------------------------------------------------------*)
(* convert a standard polynomial dPoly like equation 5.6 into the
special polynomial dpPoly like equation 5.7 :
dPoly(d^2)= dPoly[0]+dPoly[1]*d^2.+ dPoly[order]*d^2*order to
dpPoly = dpPoly[0] + dpPoly[1]/p + d^2*( dpPoly[2]/p^2 +dpPoly[3]/p^3+
d^2*( ...dpPoly[order-1]/p^order-1 + dpPoly[order]/p^order))
with p=1-d^2 and order the order of dPoly(d^2) *)

var
    pPoly,dpPl:type_poly;
    i,j:indice;

procedure change(order:integer);
var j:indice;

- 116 -
Appendix C

SOFTWARE : FILT_III,PAS

begin
if order >= 2 then
begin
(* put d^2 in factor : dpPl=dpPl[order]*p^order+
     dpPl[order-1]*p^order-1
   * + d^2 * ( dpPl[order-3]*p^order-3+ .. + dpPl[1]*p +dpPl[0] ) *)
dpPl[0]:=pPoly[0];
for j:=1 to order-1 do
  dpPl[j]:=dpPl[j-1]+pPoly[j];
dpPl[order]:=pPoly[order];
(* put pPoly equal to the part multiplied by d^2 of dpPl *)
pPoly:=nul_poly;
for j:=0 to order-2 do
  pPoly[j]:=dpPl[j];
pPoly[order-1]:=0.0; pPoly[order]:=0.0;
(* and go on with the remaining pPoly *)
change (order-2);(*recursive*)
end;
end;(*of procedure change*)

(* we are going to compute first dpPl :
   dpPl=dpPl[n]*p^n+dpPl[n-1]*p^(n-1) + d^2*( dpPl[n-2]*p^(n-2)
   +x*( .. dpPl[1]*p +dpPl[0] ) *)
begin (*of procedure d_to_dp_type1 *)
(* first step : dPoly(d^2) is transform into pPoly(p=1-d^2) *)
d_to_pPoly_convert(order,dPoly,pPoly);

(* second step : pPoly(p=1-d^2) is transform into dpPl(d^2,p) *)
dpPl:=nul_poly; (=0)
change(order);

(* third step : rewrite dpPl we have computed into dpPoly we want :
   we exchange the order of the coefficients *)
dpPoly:=nul_poly;
for i:=0 to order do
  dpPoly[i]:=dpPl[order-i];
end; (*of procedure d_to_dp_type1 *)

********************************************************************
PROCEDURE D_TO_DP_TYPE2(order:indice;
                        dPoly:type_poly;var dpPoly:type_poly);
********************************************************************
(* convert a standard polynomial dPoly like equation 5.6 into the special
polynomial dpPoly like equation 5.7 in case of SDGl, even order:
   dPoly(d^2)= dPoly[0]+dPoly[1]*d^2+.. + dPoly[order]*d^2*order to
   dpPoly = dpPoly[0] + d^2*(dpPoly[1]/p + dpPoly[2]/p^2 + d^2*(..
Appendix C

SOFTWARE : FILT_III, PAS

\[ \text{.dpPoly}[\text{order}-1]/p^{\text{order}-1} + \text{dpPoly}[\text{order}]/p^{\text{order}} \]
with \( p=1-d^2 \) and order the order of \( \text{dPoly}(d^2) \)

(* we are going to compute first \( \text{dP1} \):
\[ \text{dP1}=\text{dP1}[n]*p^n+d^2*(\text{dP1}[n-1]*p^{n-1} + \text{dP1}[n-2]*p^{n-2} + d^2*(\text{..dP1}[1]*p+dP1[0])) \) *)

\[ \text{var} \]
\[ \text{pPoly,dpP1}:\text{type}\_\text{poly}; \]
\[ i,j:\text{indice}; \]

\[ \text{procedure change2(\text{order}:\text{integer});} \]
\[ \text{var j}:\text{indice}; \]
\[ \text{begin} \]
\[ \text{if order } >=2 \text{ then} \]
\[ \text{begin} \]
\[ \text{(* put } d^2 \text{ in factor : } \text{dP1}=\text{dP1}[\text{order}]*p^{\text{order}}+ \]
\[ \text{(*} \]
\[ \text{dP1[order-3]*p^{order-3}+ dP1[1]*1+dP1[0]} \)
\[ \text{dpP1[0]}:=\text{pPoly}[0]; \]
\[ \text{for j=1 to order do} \]
\[ \text{dpP1[j]}:=\text{dpP1[j-1]}+\text{pPoly}[j]; \]
\[ \text{(* put Poly equal to the part dpl 0..order-2)} \]
\[ \text{pPoly}:=\text{nul}\_\text{poly}; \]
\[ \text{for j=0 to order-2 do} \]
\[ \text{pPoly}[j]:=\text{dpP1}[j]; \]
\[ \text{pPoly[order-1]}:=0.0; \text{pPoly[order]}:=0.0; \]
\[ (* \text{and go on with the remaining Poly}) \]
\[ \text{change2 (order-2);} \text{\{recursive\} \]
\[ \text{end}; \text{\{of procedure change\} \]
\[ \text{begin} \text{\{of procedure d_to_p_type2\} \]
\[ (* \text{first step : dPoly}(d^2) \text{ is transform into pPoly}(p=1-d^2) \) \]
\[ \text{d_to_pPoly_convert(\text{order,dPoly,pPoly});} \]
\[ (* \text{second step : pPoly}(p=1-d^2) \text{ is transform into dpP1}(d^2,p) \) \]
\[ \text{dpP1}:=\text{nul}\_\text{poly}; \text{\{0\} \]
\[ \text{change2(\text{order};} \]
\[ (* \text{third step : rewrite dpP1 we have computed into dpPoly we want :} \]
\[ \text{we exchange the order of the coefficients}) \]
\[ \text{dpPoly}:=\text{nul}\_\text{poly}; \]
\[ \text{for i=0 to order do} \]
\[ \text{dpPoly[i]}:=\text{dpP1}[\text{order-i}]; \]
\[ \text{end;} \text{\{of procedure d_to_dp_type2\} \]

- 118 -
PROCEDURE COFACTOR(order:indice; BDen_cap:type_poly;
    var dpcofactor:type_poly);

(*-------------------------------*)
(* give the d&p cofactor polynomial when a path is not touching
   loop number 1 to order. The polynomial is in d&p, p in the denominator,
   and have only even power of p *)
var NN,DD,DD_save:type_poly;
    (NN current numerator of the continuous fraction )
i,count:indice;

begin

{init poly : NN_order,DD_order}
NN:=nul_poly; (NN:=0)
DD:=nul_poly;
DD[0]:=1;

if order>-1 then
begin

NN[0]:=BDen_cap[order]; ( = C(order+1,order) )
(compute NN_i,DD_i i=order-1...1,0 )
for i:=order-1 downto 0 do
begin

(* save Di+1)
    DD_save:=DD;
(* Di:=Di+1-(d*d/p*p)*Ni+1)
    for count:=2 to max_filter_order do
        DD[count]:=DD[count]-NN[count-2];
(* Ni:=Ci+1,i*Di+1 but NO is useless)
if i>-1 then
begin
    for count:=0 to max_filter_order do
        NN[count]:=BDen_cap[i]*DD_save[count];
end;
end; ( of 'if' statement )
dpcofactor:=DD; ( the cofactor is D0 )
end; (of procedure cofactor )
PROCEDURE RESOLVE_TRIANGULAR(order:indice;Mat:max_filter_order_matrix;
poly:type_poly;var Xpoly:type_poly);
(*----------------------------------------------------------------*)
(* resolve Mat * Xpoly = poly, when Mat is lower triangular *)
(* order is the the highest indice of the actually used matrix :  
   Mat(0..order,0..order) *)

var ii,jj:indice;
   sum:real;
begin
  Xpoly[0]:=poly[0]/Mat[0,0];
  for ii:=1 to order do ( Xpoly(ii) is going to be computed )
  begin
     (* Xpoly[ii]=(poly[ii]-(Mat[ii,0]*Xpoly[0]+..  
         Mat[ii,ii-1]))/Mat[ii,ii] )
     sum:=0;
     for jj:=0 to ii-1 do sum:=sum+Mat[ii,jj]*Xpoly[jj];
     Xpoly[ii]:=(poly[ii]-sum)/Mat[ii,ii];
  end;
end; (of procedure resolve_triangulaire )

(*****************************************************************)
(** initialization instructions **)  
(*****************************************************************)

Begin
   ( Nul_poly is a max_filter_order polynomial equal to zero )
   for iii:=0 to max_filter_order do Nul_poly[iii]:=0.0;
   writeln;writeln;
   writeln('you are going to use FILTER unit :');
   writeln;'it's a set of tools for SDG filter synthesis);
   writeln;'first written by V.Jeannot, ENSERG 89, France ');
end.
Appendix C

SOFTWARE : FILT II.PAS

C . 3 : UNIT FILT_II.2

(******************************************************************)
unit filt_II2;
(******************************************************************)
{    this unit is used by filt_II : filter part II  }

{$N+} {co processor is used}
{$define debug}

{$ifdef debug}
    {$D+,R+,S+} {debug information ; rang checking ; stack checking}
{$else}
    {$D-,R-,S-}
{$endif}
(*******************************************************************)
interface
(*******************************************************************)
uses crt,filt_iil;
type
    type_branch_kind = (Cee,pCee,Coo,pCoo,doe,dCoe,deo,dCeo);
    { 8 types of branch }
    filter_description_matrix-array[indice,indice,
    type_branch_kind] of real;
    type_filter_synthesis=(SDG1,SDG2);
FUNCTION MINREAL(a,b:real):real;
{give the lowest value from two reals}

FUNCTION MAXREAL(a,b:real):real;
{give the highest value from two reals}

FUNCTION PRINTABLE_BRANCH_KIND(branch_kind:type_branch_kind):string5;
(* give a kind of branch printable string *)

PROCEDURE WRITE_THE_MATRIX(filter_order:indice;
    M_filter:filter_description_matrix;var ff:text);
(* give an printable output of a filter description matrix *)

PROCEDURE FILL_IN_DENOMINATOR(filter_order:indice;BDen_cap:type_poly;
    var M_filter:filter_description_matrix);
(* put the value of the denominator capacitance in their right place of
M_filter, the filter description *)
PROCEDURE REMOVE_NEG_CAP(filter_order:indice;
   filter_synthesis:type_filter_synthesis;
   var M_filter:filter_description_matrix;
   var error_string:string);
   (change the topology of a SC filter in order to delete all negative
    capacitances see H.Hegt:contribution to SC filter synthesis, 1988)
    ( input is the description of the filter_description,
     output is this one but corrected )

PROCEDURE RESCALE_OUTPUT(filter_order:indice;
   op_amp:indice;factor:real;
   var M_filter:filter_description_matrix;var amp_cap:type_poly);
   (multiple capacitances connected to the output number op_amp
    by 'factor' in order to divide op_amp voltage by factor )

FUNCTION LOWEST_CAP(filter_order:indice;op_amp:indice;
   M_filter:Filter_description_matrix;
   amp_cap:type_poly):real;
   (give the lowest capacitance value connected to the input of
    the Operational amplifier number op_amp )
   (use MINreal(a,b:real):real)

PROCEDURE TOTAL_CAP_SCALE(filter_order:indice;op_amp:indice;
   factor:real;var M_filter:filter_description_matrix;
   var amp_cap:type_poly);
   (scaling of delta Q of op_amp input : all capacitances connected to
    the input of this op_amp are multiplied with factor )

FUNCTION TOTAL_CAPACITY(filter_order:indice;
   M_filter:filter_description_matrix;amp_cap:type_poly):real;
   ( give the total capacitance before capacitance sharing )
**SOFTWARE: FILT II.PAS**

(*******************************************************************)

IMPLEMENTATION

(*******************************************************************)

(*-------------------------------------------------------------*)
FUNCTION MINREAL(a, b: real): real;
(*-------------------------------------------------------------*)
{give the lowest value from two reals}
begin
  if a<b then minreal := a else minreal := b;
end; {of function MINreal}

(*-------------------------------------------------------------*)
FUNCTION MAXREAL(a, b: real): real;
(*-------------------------------------------------------------*)
{give the highest value from two reals}
begin
  if a>b then MAXreal := a else MAXreal := b;
end; {of function MAXreal}

(*-------------------------------------------------------------*)
FUNCTION PRINTABLE_BRANCH_KIND(branch_kind: type branch_kind): string;
(*-------------------------------------------------------------*)
(* give a kind of branch printable string *)
begin
  case branch_kind of
    Cee : printable_branch_kind := 'Cee';
    pCee: printable_branch_kind := 'pCee';
    Coo: printable_branch_kind := 'Coo';
    pCoo: printable_branch_kind := 'pCoo';
    doe : printable_branch_kind := 'doe';
    dCoe: printable_branch_kind := 'dCoe';
    deo : printable_branch_kind := 'deo';
    dCeo: printable_branch_kind := 'dCeo';
  end;
end; {of function printable_branch_kind }
Appendix C

SOFTWARE : FILT_II.PAS

PROCEDURE FILL_IN_DENOMINATOR(filter_order:indice;BDen_cap:type_poly;
var M_filter:filter_description_matrix);

(* put the value of the denominator capacitance in their right
place of M_filter, the filter description *)

var count:Indice;
helpvar:real; (don't care)
begin
(* fill in deo and doe matrix d branches *)
for count:-l to filter_order do
begin
M_filter[count,count,deo]:=-1.0;
M_filter[count,count,doe]:=-1.0;
end;

(* fill in dcoe matrix with forward dc branches that
are assumed to be 1*)
count:=1;
repeat
M_filter[count+1,count,dcoe]:=-1.0;
count:=count+2;
until count>=filter_order;

(* fill in dceo matrix with backward dc branches *)
count:=2;
repeat
M_filter[count,count+1,dceo]:=-1.0;
count:=count+2;
until count>=filter_order;

(* fill in dcoe matrix with backward dc branches *)
(* BDen_cap[i]=Ci+1,i; BDen_cap[filter_order]=Cs *)
count:=1;
repeat
M_filter[count,count+1,dcoe]:=BDen_cap[count];
count:=count+2;
until count>=filter_order;

(* fill in dceo matrix with backward dc branches *)
(* BDen_cap[i]=Ci+1,i; BDen_cap[filter_order]=Cs *)
count:=l;
repeat
M_filter[count,count+1,dceo]:=BDen_cap[count];
count:=count+2;
until count>=filter_order;

(* add Cs branch *)
if odd(filter_order)
then M_filter[filter_order,filter_order,Co]:=-
BDen_cap[filter_order]
else M_filter[filter_order,filter_order,Cee]:= -
BDen_cap[filter_order];
end; (of procedure fill_in_denominator)
PROCEDURE WRITE_THE_MATRIX(filter_order:indice;
    M_filter:filter_description_matrix; var ff:text);

(* give an printable output of a filter description matrix *)
var row,column:indice;
    branch_kind:type_branch_kind;
begin
    for branch_kind:=Cee to dGeo do ( all kind of branch )
    begin
        writeln(ff,blk,' BRANCH TYPE : ',
            printable_branch_kind(branch_kind):5);
        writeln(ff);
        writeln(ff,blk,' branch from node :');
        write(ff,blk,' to /',',', Vin',' ');    
        for colum:=1 to filter_order do
            write(ff,' V',colum:1,' ');
        writeln(ff);
        writeln(ff,blk,'node');
        for row:=0 to filter_order do
        begin
            if row=0 then write(ff,blk,'Vin ')
                else write(ff,blk,'V',row:1,' ');
            for colum:=0 to filter_order do
            write(ff,M_filter[row,colum,branch_kind]:6:3,' ');
            writeln(ff); writeln(ff);
        end;
    end;
end; ( of procedure write_the_matrix )
PROCEDURE REMOVE_NEG_CAP(filter_order:indice;
  filter_synthesis:type_filter_synthesis;
  var M_filter:filter_description_matrix;
  var error_string:string);

(change the topology of a SC filter in order to delete all negative
  capacitances see H.Hegt:contribution to SC filter synthesis, 1988)
  (input is the description of the topology, output is this one but
  corrected)

var nose,tail:indice;  
  (tail : francais: queue)
  (nose : francais: nez,tete)
{**************************}
  branch_kind:type_branch_kind;
  chaine:string;
  {$ifdef debug}
    ff:text;
  {$endif}

PROCEDURE change_sign_VCopamp:indice);  
  (change the output sign of the opamp number ‘opamp’)

var branch_kind:type_branch_kind;
  counter:indice;
begin
if opamp=0 then
  writeln(‘error : it’s impossible to change the sign of the input’)
else
begin
  for counter:=0 to filter_order do
  begin
    for branch_kind:=Cee to dCeo do
    (all kinds of branch are concerned)
    begin
      M_filter[counter,opamp,branch_kind]:=-M_filter[counter,opamp,branch_kind];
      M_filter[opamp,counter,branch_kind]:=-M_filter[opamp,counter,branch_kind];
      end;
    end;
  end;
end; (or procedure change_sign)

FUNCTION Ve_is_dVo(nose:indice):boolean;
  (* is true when Ve=d*Vo e:even;o:odd*)

var tail:indice;
  tempbool:boolean; (temporary result)
Appendix C
SOFTWARE : FILT_II.PAS

begin
  tempbool:=true;
  for tail:=0 to filter_order do
    begin
      (verify doe matrix)
        if tail>nose then
          begin
            tempbool:=tempbool and (M_filter[nose,tail,doe]=1.0);
            (d branch from Vo to Ve)
          end
        else
          begin
            tempbool:=tempbool and (M_filter[nose,tail,doe]=0.0);
            (no other one)
          end
      (verify the other matrix : no other branch)
        tempbool:=tempbool and (M_filter[nose,tail,dCoe]=0.0) and
        (M_filter[nose,tail,Cee]=0.0) and
        (M_filter[nose,tail,pCee]=0.0);
      if not(tempbool) then
        write(') ;
    end;
  Ve_is_dVo:=tempbool;
  if not(tempbool) then
    writeln( 'error Ve is not dVo', nose:3);
end; {of function Ve-dVo)

(*-------------------------------------------------------------*)
FUNCTION Vo_is_dVe(nose:indice):boolean;
(* is true when Vo=d*Ve *)
var tail:indice;
  tempbool:boolean; (temporary result)
begin
  tempbool:=true;
  for tail:=0 to filter_order do
    begin
      (verify doe matrix)
        if tail>nose then
          begin
            tempbool:=tempbool and (M_filter[nose,tail,deq]=1.0);
            (d branch from Ve to Vo)
          end
        else
          begin
            tempbool:=tempbool and (M_filter[nose,tail,deq]=0.0);
            (no other one)
          end
      (verify the other matrix : no other branch)
        tempbool:=tempbool and (M_filter[nose,tail,dCoe]=0.0) and
        (M_filter[nose,tail,Coo]=0.0) and
        (M_filter[nose,tail,pCoo]=0.0);
      if not(tempbool) then
        write(') ;
    end;
  Vo_is_dVe:=tempbool;

- 127 -
Appendix C

SOFTWARE : FILT_II.PAS

if not(tempbool) then
    writeln( 'error Vo is not dVe', nose:3);
end; {of function Vo_is_dVe}

BEGIN {OF PROCEDURE REMOVE_NEG_CAP : 'remove negative capacitance'}

(******************************************************************************)
{%ifdef debug}
    assign(ff,'inter.pas');
    rewrite(ff);
{%endif}

(* eliminate negative capacitance in -pC or -C od dC from input branches *)

(*------------------------------------------------------*)
tail:=0; {look only to branch coming from input}
for nose:=1 to filter_order do
    ('for 1' : no ending branch in the input!)
begin
    for branch_kind:=Cee to dCeo do
        { that is to say all kind of branch }
        (* don't care of dCO branch of SDGI *)
        if (filter_synthesis<>SDGl) or (branch_kind<>dCeo) then
            (* if nose=tail then it's impossible to correct the topology *)
            if (M_filter[nose,tail,branch_kind]<0)
                and (nose<>tail) then
                    change_sign_V(nose);
    end;
{%ifdef debug}
    writeln(ff,'matrix after removing negative branch from input' );
    write_the_matrix(filter_order,M_filter,ff);
{%endif}

(* eliminate negative capacitance in -dC branch *);

(*------------------------------------------------------*)
for nose:=1 to filter_order do
    ('for 1' : no ending branch in the input!)
begin
    for tail:=0 to filter_order do {SDGI C0:dCeo from input to node l}
    begin
        if M_filter[nose,tail,dCeo]<0 then
            (try to correct )
            if Ve_is_dVo(nose) then
                {eliminate dC branch}
                begin
                    (* dC tranformed into -(C) branch *)
                    M_filter[nose,tail,Coo]:=- M_filter[nose,tail,dCeo];
                    M_filter[nose,tail,dCeo]:=0 {remove dC branch}
                end
            else {error}
                begin
                    sound(1200);delay(200);nosound;
                    - 128 -
Appendix C

SOFTWARE : FILT_II.PAS

write(' I can't correct the sign of branch dCeo ');
writeln(' from node ',tail:3,' to node ',nose:3);
end;
if M_filter[nose,tail,dCoe]<0 then
  if Vo_is_dVe(nose) then (eliminate_dC branch)
  begin
    (dC transformed into -(-C) branch)
    M_filter[nose,tail,Cee]:=- M_filter[nose,tail,dCoe];
    M_filter[nose,tail,dCoe]:=0 (remove dC branch)
  end
else (error)
  begin
    sound(1100);delay(200);nosound;
    writeln(' I can't correct the sign of branch dCoe ',
            ' from node ',tail:3,' to node ',nose:3);
  end;
end;
end;

(* verify that all - branches have positive capacitances *)
(*-------------------------------------------------------------*)
error_string:='no error';
for nose:=0 to filter order do
  for tail:=0 to filter order do
    for branch_kind:-Cee to dCeo do (that is to say all kind)
      if (M_filter[nose,tail,branch_kind]<0) and
      (error_string='no error') then
        begin
          str(tail:3,chaine);
          error_string:=
          concat('error : negative capacitance',
                 printable_branch_kind(branch_kind),
                 ' from node ',chaine,' to node ');
          str(nose:3,chaine);
          error_string:=concat(error_string,chaine);
        end;
{$ifdef debug}
close(ff);
{$endif}
end; {of procedure remove_neg_cap}
PROCEDURE RESCALE_OUTPUT(filter_order, op_amp: indice; factor: real;
var M_filter: filter_description_matrix; var amp_cap: type_poly);
(* ----------------------------------------------------------------*)
{multiple capacitances connected to the output number
op_amp by 'factor'in order to divide op_amp voltage by factor}
var
  branch_kind: type_branch_kind;
  count: indice;
  test: real; {for debugging}
begin
  {scale all capacitances connected to the output}
  amp_cap[op_amp]: = amp_cap[op_amp] * factor;
  for branch_kind:= Cee to dCeo do {all kind of branches }
  { * don't care of d branch: }
  if (branch_kind<> doe) and (branch_kind<> deo) then
  for count:= 1 to filter_order do
  begin
    test:= M_filter[count, op_amp, branch_kind];
    M_filter[count, op_amp, branch_kind]:= M_filter[count, op_amp, branch_kind] * factor;
  end;
end; {of procedure rescale_output}

FUNCTION LOWEST_CAP(filter_order: indice; op_amp: indice;
M_filter: Filter_description_matrix;
amp_cap: type_poly): real;
(* ---------------------------------------------------------------*)
{give the lowest capacitance value connected to the input of
the Operational amplifier number op_amp }
{use Minreal(a, b: real): real}
var mintemp: real;
  branch_kind: type_branch_kind;
  count: indice;
  test: real; {don't care}
begin
  mintemp:= Amp_cap[op_amp];
  {scan all capacitances connected to the input }
  for branch_kind:= Cee to dCeo do {all kind of branches }
  { * don't care of d branch }
  if (branch_kind<> doe) and (branch_kind<> deo) then
  for count:= 0 to filter_order do
  if M_filter[op_amp, count, branch_kind]> 1.0e-08 then
  {see VII.3, note 11}
  begin
    test:= M_filter[op_amp, count, branch_kind];
    mintemp:= Minreal(mintemp, M_filter[op_amp, count, branch_kind]);
  end;
  end;
end;
lowest_cap:=mintemp;
end; {of function lowest_cap}

(*---------------------------------------------------------------*)
PROCEDURE TOTAL_CAP_SCALE(filter_order, op_amp: indice; factor: real;
  var M_filter: filter_description_matrix;
  var amp_cap: type_poly);
(*---------------------------------------------------------------*)
{scaling of delta Q of op_amp input : all capacitances connected to
the input of this op_amp are multiplied with factor}

var
  branch_kind: type_branch_kind;
  count: indice;
  test: real; {don't care}
begin
{scale all capacitances connected to the input}
  amp_cap[op_amp]:= amp_cap[op_amp]*factor;
for branch_kind:= Cee to dCeo do {all kind of branches}
  {don't care of d branch}
  if (branch_kind<>doe) and (branch_kind<>deo) then
    for count:= 0 to filter_order do
      begin
        test:= M_filter[op_amp, count, branch_kind];
        M_filter[op_amp, count, branch_kind]:=
          M_filter[op_amp, count, branch_kind]*factor;
      end;
end; {of procedure total_cap_scale}
FUNCTION TOTAL_CAPACITANCE(filter_order: indice;
    M_filter:filter_description_matrix;
    amp_cap:type_poly):real;
BEGIN
    sum_cap:=0.0;
    for branch_kind:=Cee to dCeo do (all kind of branches)
        if (branch_kind<>doe) and (branch_kind<>deo) then
            for count1:=0 to filter_order do
                for count2:=0 to filter_order do
                    sum_cap:=sum_cap+abs(M_filter[count1,count2,
                        branch_kind]);
            for count1:=1 to filter_order do
                sum_cap:=sum_cap+abs(amp_cap[count1]);
            total_capacitance:=sum_cap;
    total_capacitance:=sum_cap;
END; (of function total_capacitance)
C.4 : PROGRAM ES_FILT.PAS

program es_filt;

{ THIS PROGRAM TEST THE BIG PROCEDURE 'COMPUTE_FILTER_CAPACITANCES' }

(*******************************************************************************
uses filt_III, { define max_filter_order,...} 
   filt_II2, 
   filt_II; { define the main procedure }

var
(var input of the program : )
   filter_synthesis:type_filter_synthesis;
   filter_order:indice;
   dDen:type_poly; {denominator of the transfer function in power 
   of d^2}
   dNum:type_poly; {numerator of the transfer function in power 
   of d^2 }

(var output of the program )
   M_filter:filter_description_matrix;
   amp_cap:type_poly;
   ff:text; {output file, if used }

(var used here)
   count:indice;

*******************************************************************************
begin { main program }
(*******************************************************************************
{$define debug}

(* output file reading *)
assign(ff,''); {standard output }
{$ifdef debug}
   assign(ff,'nul');
{$endif}
if paramcount>1 then
   assign(ff,paramstr(1));
rewrite(ff);
(* see this matrix *)
writeln(ff);
writeln(ff,copy(blk,0,10),
'-------------------------------_ .. _------------');
writeln(ff,copy(blk,0,10),
' - D A T A O U T P U T - ');
writeln(ff,copy(blk,0,10),
'-------------------------------');
writeln(ff);
writeln(ff,blk,'Final matrix after the topology modifications :');
writeln(ff);
WRITE_THE_MATRIX(filter_order,M_filter, ff);
writeln(ff);writeln(ff);
for count:-l to filter_order do
   writeln(ff,blk,'CA' ,count:l,' - ' ,amp_cap[count]:6:3);
writeln(ff);
writeln(ff,blk,'Total capacitance : ',
   TOTAL_CAPACITY(filter_order,M_filter,amp_cap):6:3);
close(ff);
end.
INITIALIZATION OF INPUT PARAMETERS

filter_synthesis:=SDG1;
filter_order:=3;

dDen:=nul_poly; (=0)
dNum:=nul_poly;

dDen[0]:=2.618; dDen[1]:=0.853; dDen[2]:=1.0;
dNum[0]:=-1.529; dNum[1]:=-0.984; dNum[2]:=-0.749;

dDen[0]:=-2.618; dDen[1]:=0.853; dDen[2]:=-0.678; dDen[3]:=1.0;
dNum[0]:=-1.529; dNum[1]:=-0.984; dNum[2]:=-1.641; dNum[3]:=0.749;

writeln(ff);
writeln(ff,copy(blk,0,10),',-----------------------------------------------');
writeln(ff,copy(blk,0,10),', FILTER SYNTHESIS -');
writeln(ff,copy(blk,0,10),',-----------------------------------------------');

writeln(ff); writeln(ff,blk,' input data ');
case filter_synthesis of
  SDG1: writeln(ff,blk,'1 : Filter type : SDG1');
  SDG2: writeln(ff,blk,'1 : Filter type : SDG2');
end;

writeln(ff); writeln(ff,blk,'2 : filter of order : ',filter_order:2);
writeln(ff); writeln(ff,blk,'3 : initial d^2 polynomial :');
for count:=0 to filter_order do
  writeln(ff,blk,'dNum[count]:',dNum[count]:6:3);
writeln(ff);
for count:=0 to filter_order do
  writeln(ff,blk,'dDen[count]:',dDen[count]:6:3);
writeln(ff);

IF SYNTHESIS_IS_POSSIBLE(filter_order,filter_synthesis,dNum,dDen) then
  writeln(ff,blk,' the synthesis is possible, no error will occur')
else
  begin
    writeln(ff,blk,' the synthesis is not possible.');
    writeln(ff,blk,' one error at least will occur when the program');
    writeln(ff,blk,' will try to remove negative capacitances');
  end;

COMPUTE_FILTER_CAPACITANCES(filter_order,filter_synthesis,dNum,dDen,
  M_filter,amp_cap,ff);

(*-----------------------------------------------------------------*)

SOFTWARE : ES_FILT.PAS

appendix C