UC San Diego
Technical Reports

Title

GRYD: Generalized Reduced-Order Wye-Delta Transformation: Programmer's Manual for Reduction Engine and Applications

Permalink
https://escholarship.org/uc/item/1425f785

Authors
Qin, Zhanhai
Cheng, Chung-Kuan

Publication Date
2003-07-31

Peer reviewed

eScholarship.org Powered by the California Digital Library
University of California

UNIVERSITY OF CALIFORNIA, SAN DIEGO

GRYD: Generalized Reduced-Order Y-Δ Transformation

Programmer's Manual for Reduction Engine and Applications

Zhanhai Qin, Chung-Kuan Cheng
Department of Computer Science and Engineering
UCSD Technical Report No. xxxx-xx

February 2003

Abstract

GRYD is a multi-port linear RCLK-VJ network reduction software package. The package features:

1. an efficient linear network reduction engine based on the generalized Y-Δ transformation algorithm [1];
2. GRYD simulator, which evaluates transient response waveforms to typical input signals, e. .g., impulse, piecewise linear, and expotential functions;
3. GRYD pole analyzer, which evaluates the system transfer function matrix, poles and zeros, and a reduced-model stabilization mechanism[2];
4. GRYD network synthesizer, which realizes the reduced network and outputs a SPICE-compatible netlist file.

The reduction engine takes as input a SPICE[3] netlist file and generates a reduced admittance network in s domain. An important feature of the reduction engine is that each reduced admittance is a *rational function* of s, and the transfer functions of reduced network are *exact* up to a user-specified order β. This programmer's manual covers both the engine and the applications.

Keyword: Y-Δ transformation, interconnect model order reduction, symbolic network analysis, pole analyis, network synthesis.

Contents

1 Getting Started **1**
1.1 Overview 1
1.2 Generalized Y-Δ Transformation 4
1.3 Further Readings 7

2 High-Level Design Description **8**
2.1 GRYD Reduction Engine 10
2.2 GRYD Simulator 17
2.3 GRYD Pole Analyzer 19
2.4 GRYD Network Synthesizer 21
2.5 Utilities 22

3 Data Type and Subroutine Description **27**
3.1 Data Type Definition 27
3.2 Subroutine Description 27

4 Summary and Support **63**

Chapter 1

Getting Started

This chapter is an overview of the generalized Y-Δ transformation. Because the technique, later on we call it GRYD reduction engine, is the basis for all of its applications, such as GRYD simulator, GRYD Pole Analyzer, and GRYD Network Synthesizer. Therefore, acquaintance of the generalized Y-Δ transformation is crucial and necessary. However the scope of this manual is limited to a quick overview on the technique, in order to fully understand the material, we strongly suggest readers to go through the technical report[1].

1.1 Overview

Linear networks in modern VLSI chips such as power/ground meshes, clock distribution networks, and global interconnects are growing fast as feature size shrinks. A linear network with millions of lumped RLC elements imposes a critical challenge to circuit simulation techniques. Simulating such networks in SPICE becomes simply not practical. Instead, two strategies are widely adopted: (1) to increase the efficiency of solving linear simultaneous equations used in SPICE; and (2) to reduce the size of original networks using model order reduction techniques.

In the first category, Chen[5] proposes to employ more efficient preconditioned Krylov-subspace iterative methods with Nodal Analysis (NA) compared with LU factorization with Modified Nodal Analysis (MNA) used in SPICE. Qin[6] shows that SuperLU [7], a variant of LU factorization, provides comparable performance with iterative methods while the robustness of direct methods is kept. Kozhaya[8, 9] explores the regular grid structure

of power/ground networks and use multigrid technique to solve a coarse grid and map the solution back to the original fine grid. These approaches fall into the first category.

In the second category, the moment-matching technique proposed by Pillage[10], has been used widely to approximate waveforms of a linear interconnect network by matching lower order moments with Padé approximation. Works by Lin[11] and Liao[12], for example, are related to this. As each moment can be computed in linear time after an one-time LU factorization, the algorithm runs very efficiently. It is well known that Padé approximation may generate undesired positive poles. To overcome the drawback, Liao[13] proposes a method to realize reduced RC sub-networks as macromodels. Sub-networks are reduced by means of preserving lower orders of the port admittance matrix. The method guarantees the realizability of the macromodels for RC circuits. Feldmann's Matrix Padé Via Lanczos[14], Boley's block Arnoldi[15] and Odabasioglu's PRIMA[16] are admittance-matrix-based model order reduction methods, so that they performs model order reduction on each entry in admittance matrix simultaneously. Kerns[17] first introduces congruence transformations for order reduction of RC circuits. The same author proposes split congruence transformations[18] for passive reductions of RLC circuits.

In another aspect, topological analysis [19] is an approach to calculating driving-point admittances using Cramer's rule in s-domain. The determinant of an admittance matrix of a passive network without mutual inductances is equal to the sum of all the tree admittance products of the network. The advantage of topological analysis formula over conventional methods evaluating determinants is that it avoids the usual cancellations inherent in the expansion of determinants in the latter. But enumerating all the trees in a large network is impractical.

Recently Ismail proposes a direct transfer-function truncation (DTT) method[20] to approximate transfer functions in tree-structured RCL networks in s-domain. The transfer functions are kept in rational expressions in s, and an approximation is acquired by directly truncating high-order terms. Such an approximation also matches low-order time moments implicitly, but truncated characteristic denominator may not be stable any more. The method is able to obtain very high-order transfer functions, when AWE fails because of numerical problems.

We have proposed a new model order reduction method[2] based on Y-Δ *transformation* [21] for general RCLK-VJ linear networks. In this approach,

we classify nodes in a network into two categories:

1. external nodes: nodes where responses are of interest;

2. internal nodes: all other nodes in the network.

The principal idea is that, given a linear network, we perform Y-Δ transformation on every internal node, until all such nodes are eliminated. Note that the more the external nodes are specified, the sooner the algorithm terminates. After each transformation, any admittance of order higher than user-specified threshold β will be truncated. For example, suppose

$$\frac{a_0 + a_1 s + \cdots + a_m s^m}{b_0 + b_1 s + \cdots + b_n s^n} \tag{1.1}$$

is an admittance after a Y-Δ transformation, the truncation with respect to β would result in a βth-order admittance

$$\frac{a_0 + a_1 s + \cdots + a_\beta s^\beta}{b_0 + b_1 s + \cdots + b_\beta s^\beta}, \qquad 0 \leq \beta \leq \min(m, n), \tag{1.2}$$

which is an approximation to the exact admittance in (1.1). Different from topological analysis and other traditional symbolic analysis, the approach keeps Y-Δ admittances of order $\leq \beta$. All higher-order terms are discarded. The terms kept, however, are precisely the first $\beta + 1$ terms in exact admittances.

The main contributions of this work are:

1. Y-Δ admittances are kept in the original rational forms of s, but their orders are reduced to no more than β. In Y-Δ admittances, all coefficients of powers of s agree with those in exact admittances up to the order β;

2. First $\beta + 1$ time moments of exact admittances are matched implicitly by truncated Y-Δ admittances, including the 0th moment m_0;

3. Two kinds of common-factor effects are first discovered in Y-Δ transformation. The findings lead to *essential* numerical improvement in traditional Y-Δ transformation, and hence more accurate pole/zero approximation;

4. A Hurwitz polynomial approximation method is employed to treat transfer functions from truncated Y-Δ admittances, so that stable reduced transfer functions are guaranteed.

5. The proposed algorithm is more general than DTT method [20], as it handles linear networks in arbitrary topology with current/voltage sources and inductive K elements proposed by Devgan [22].

1.2 Generalized Y-Δ Transformation

Equations given in this section will be referred intensively when we present the high-level design description in Chapter 2. Particularly, Theorem 1 and 2 are the two most important theoratical results used in GRYD reduction engine. And Corollary 1 is the basis of GRYD network synthesizer.

We would like to summarize the notations and conventions we are going to use throughout the paper before we continue. A current source is said to be *floating* when it flows from one non-datum node to another non-datum node. Decoupling it is to remove the current source, and insert two concatenated ones of the same amount of current between the two end-nodes. They are concatenated at the ground node. Through this equivalent source transformation, current sources become grounded and associated with nodes but not branches, which makes our algorithm simpler. Similarly, voltage sources can be transformed to current sources and decoupled if necessary. For a given linear network:

- n_k is denoted as the k-th labeled node, where k starts from 0. Nodes are eliminated in the order labeled;

- when the k-th node is eliminated, the network will be updated accordingly. We label the network (graph) *before* the elimination as $G_k(V_k, E_k)$, and the network (graph) *after* as $G_{k+1}(V_{k+1}, E_{k+1})$;

- $(n_i, n_j)^{(k)}$ is the branch between n_i and n_j, $Y_{i,j}^{(k)}$ the admittance of $(n_i, n_j)^{(k)}$, $I_i^{(k)}$ the current source impinging on n_i, and $\Gamma_i^{(k)}$ the neighbor set of n_i. Here superscript (k) stands for "in graph G_k";

- The first neighbor of n_i in $G^{(k)}$ is the node in $\Gamma_i^{(k)}$ with the smallest label;

- When it is not ambiguous, superscripts will be ignored, i.e., (n_i, n_j) represents the branch between node n_i and n_j, $Y_{i,j}$ the admittance of branch (n_i, n_j), and I_i the decoupled current source impinging on node n_i in the graph in the context.

The traditional Y-Δ transformation is generalized in this paper in the following four aspects:

1. Y-Δ transformation is applied to more general circuits in contrast to DTT [20], which is tree-based. Due to the generalization, two kinds of common-factor effects emerge in Y-Δ admittances and they are treated to assure numerical stability.

2. Current/Voltage sources are handled together with admittances in Y-Δ transformation. It is no longer the straightforward source transformation when similar common-factor effects to 1) are identified as well.

3. Since many nodes will be eliminated in a general circuit using the transformation, importance of the order of picking the nodes is studied and treated.

4. Mutual K elements are integrated in the transformation through a simple conversion, so that circuits with mutual inductances can be handled as well.

The generalized Y-Δ transformation formulae cover linear resistors, capacitors, self inductors and K elements, and current/voltage sources.

Theorem 1 *Suppose n_k is the node being eliminated. For all $n_i, n_j \in \Gamma_k^{(k)}$, $(n_i, n_j)^{(k+1)} \in E_{k+1}$ in G_{k+1} after n_k is eliminated. And the admittance $Y_{ij}^{(k+1)}$ is calculated as*

$$Y_{ij}^{(k+1)}(s) = Y_{ij}^{(k)}(s) + Y_{ij}^{\prime(k+1)}(s), \quad \forall n_l \in \Gamma_k^{(k)}, \tag{1.3}$$

where

$$Y_{ij}^{\prime(k+1)}(s) = \frac{Y_{ik}^{(k)}(s) \times Y_{jk}^{(k)}(s)}{\sum_l Y_{lk}^{(k)}(s)}. \tag{1.4}$$

If $I_k^{(k)} \neq 0$,

$$I_i^{(k+1)}(s) = I_i^{(k)}(s) + \frac{Y_{ik}^{(k)}(s)}{\sum_l Y_{lk}^{(k)}(s)} I_k^{(k)}(s), \quad \forall n_l \in \Gamma_k^{(k)}, \quad \forall n_i \in \Gamma_k^{(k)}. \quad (1.5)$$

For admittances and current sources not mentioned above, they will be inherited by G_{k+1} *from* G_k.

The theorem states that when we perform Y-Δ transformation on n_k, neighbors of n_k in G_k will become pairwise adjacent in G_{k+1}. In practice, we calculate $Y_{ij}^{(k+1)}$ in (1.3) up to the term of order β only. Since computation of higher-order terms is skipped, we get an approximation of $Y_{ij}^{(k+1)}$ whose numerator and denominator are equal to the first β terms in $Y_{ij}^{(k+1)}$'s numerator and denominator, respectively. $I_i^{(k+1)}$ in (1.5) is calculated in the same way. Th. 1 does not cover voltage sources, because they can be changed to current sources via source transformation before any elimination begins.

It is worth noting from Th. 1, that in Y-Δ transformation, coefficients of admittance are derived directly from admittance in original circuits and are kept in its original rational form. Stable reduced-order models can be derived from low-order truncated admittances using Hurwitz polynomial approximation[2].

This corollary is the basis of our GRYD network synthesizer (2.4).

Corollary 1 *If all RLC elements in a linear network are positive,* $Y_{ij}^{(k+1)}$ *in (1.3) is a rational function of* s

$$Y_{ij}^{(k+1)}(s) = \frac{A_{ij}^{(k+1)}}{B_{ij}^{(k+1)}} = \frac{\sum_{p=0}^{m} a_p s^p}{\sum_{q=0}^{n} b_q s^q}, \quad (1.6)$$

and a_p, b_q *in (1.6) are non-negative.*

Corollary. 1 holds immediately due to (1.3) in Th. 1.

The following theorem, quoted from [1], defines the type-I and type-II common factors found in Y-Δ transformation.

Theorem 2 $\forall k$, *suppose* n_k *is to be eliminated,* n_k *has* m *neighbors in* G_k, *and the admittances of branches incident to* n_k *are denoted as* $\frac{A_1}{B_1}, \frac{A_2}{B_2}, \cdots, \frac{A_m}{B_m}$.

Type-I and type-II common factors resulted from n_k *are equal to* ω_k*, which is defined as:*

$$\omega_k = \frac{\sum_{i=1}^{m} \left(A_i \prod_{j=1,j\neq i}^{m} B_j \right)}{W_k}, \tag{1.7}$$

where

$$W_k = \prod_{i=0}^{k-1} \omega_i^{p_i} \tag{1.8}$$

and p_i *is the number of denominators in* $\{B_1, B_2, \cdots, B_m\}$ *that carry factor* ω_i*.*

1.3 Further Readings

The purpose of this report is to help maintain and improve the current version of the software package. The rigorous derivation of the algorithms implemented in the program is beyond the scope of the report. Interested readers are encouraged to refer to the technical paper[2] and reports[1][4] for explanations and theoratical proofs.

The rest of the report is organized as follows. Chapter 2 gives the high-level algorithm descriptions in pseudo code. After this, we present the design details: data type definitions used throughout the whole package are given in Chapter 3.1; it also constitutes discussions of lower-level implementation details, including subroutine descriptions and comments on differing implementations of the core engine. Chapter 4 concludes the manual, and provides support contacts.

Chapter 2

High-Level Design Description

After a brief introduction to the proposed generalized Y-Δ transformation in the previous chapter, we describe the high-level algorithms in this chapter. The design is divided into four major design blocks, each of which contains some more detailed function blocks. In Fig. 2.1, we give an overview of the relationships among users and the four design blocks, i. e., the reduction engine and the three applications.

High-level document in the chapter for each major block will start with a design flow describing the interrelationships of the functions within it, followed by pseudo code of the functions themselves. Before you continue, make sure that you are familar with the following conventions. We will type words in various fonts to discriminate their associated categories, i. e.,

- **top-level functions**;
- *bottom-level functions*;
- `variables`. Local variables begin with an underline (_), except some mathematical characters;
- comments;
- keywords are in CAPITAL letters.

Particularly, if a function call is written in **bold face**, then you will be able to find that specific function in pseudo code as well in the context. Otherwise the function call will be written in *italic shape*.

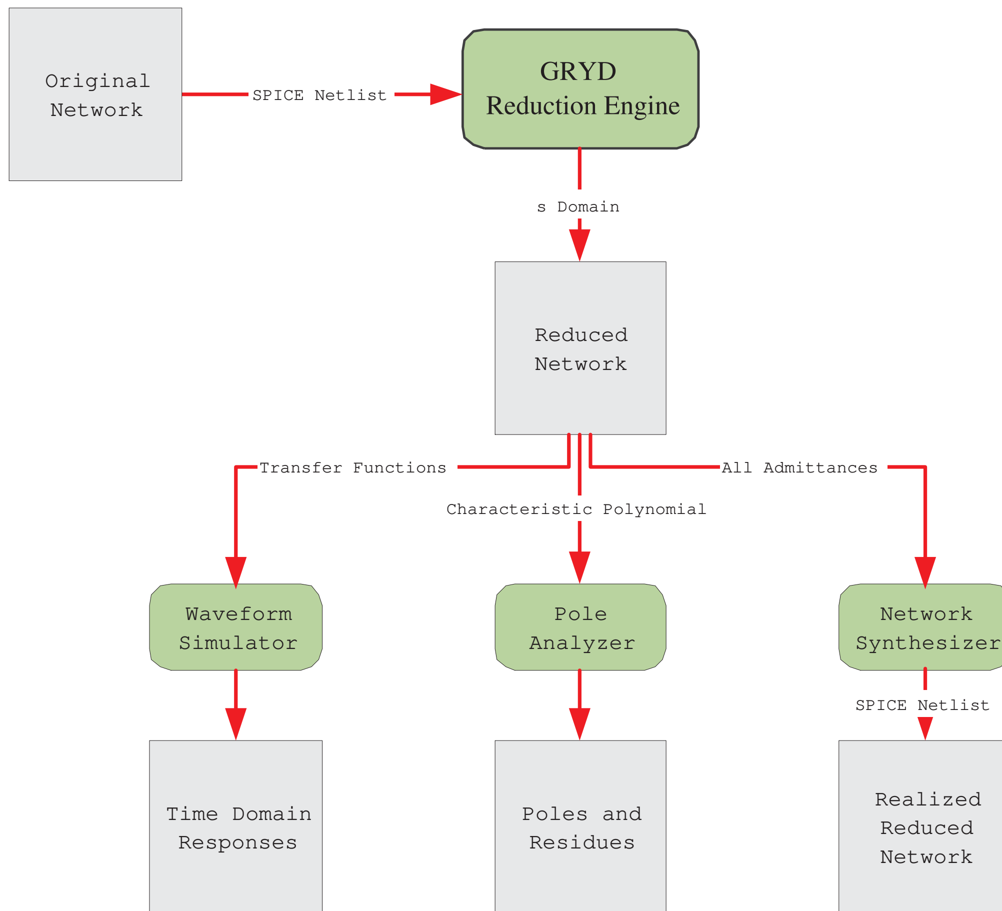


Figure 2.1: Design flow in GRYD program.

2.1 GRYD Reduction Engine

Driver of Reduction Engine

```
PROCEDURE ReductionDriver(InFile, β, OutFile)
// InFile is a SPICE input netlist file;
// β is the given threshold of order of polynomial truncations;
// OutFile is an order-reduced output netlist file for internal data exchange;
// The procedure is the driver of GRYD reduction engine. It takes as input
// an SPICE netlist file, and generates as output an order-reduced netlist
// file, an intermediate result to be used by various applications.

1    New(Ckt, ExtNodeList);
2    SpiceParser(Ckt, InFile);
3    Reduction(Ckt, β, ExtNodeList);
4    CktDump(Ckt, β, OutFile).
```

Input-Netlist Parser (SPICE Compatible)

```
PROCEDURE SpiceParser(Ckt, InFile)
// InFile is the given SPICE input netlist file;
// Ckt is the pointer to the internal data structure;
// The procedure is to take as input a SPICE input netlist file, and load
// data into the internal data structures defined.

1    Initialize hash tables _nhash and _bhash for nodes and branches;
2    Skip the first line (.title line) in InFile;
3    WHILE (_line=fgets(InFile)) DO BEGIN
4        parse _line:
5            IF (_line is an output deck, e. g. .print or .plot) THEN
6                save node IDs in output-node list;
7            _s1, _s2 ← node names in net _line;
8            _branchInfo ← type & value of net _line;
9
10       IF (_s1, _s2∈ _nhash) THEN
11           _n1=hash_lookup(_nhash, _s1);
12       ELSE
13           _n1=node_new(_s1);
14           _n1=hash_insert(_nhash, _n1);
15       Repeat 9 to 13 for _s2;
15
16       IF (∃ branch between _n1 and _n2 in _bhash) THEN
17           _oldBranch=hash_search(_bhash, _n1, _n2);
18           _oldBranch=BranchMerging(_oldBranch, _branchInfo);
19       ELSE
20           _branch=branch_new(_branchInfo);
21   END WHILE
22
23   Recompile output-node list to link pointers to the nodes;
24   Convert voltage srcs to current srcs;
25   Convert current srcs from time to s domain by Padé approximatin;
26   Decouple floating current srcs, if any.
```

Branch Merging

```
PROCEDURE BranchMerging(OldBranch, NewBranch)
// OldBranch is the existing branch;
// NewBranch is a new branch;
// The procedure is to merge the existing branch with a new one. Enumerous
// combinations of different types of branches cause the nontriviality of
// this procedure. Basically three types of elements are defined:
// iSRC (curent source), vSRC (voltage source), and adm (admittance).

1     SWITCH (types of OldBranch and NewBranch) BEGIN
2         CASE vSRC & vSRC
3             TrapError("voltage sources in loop");
4         CASE vSRC & (iSRC or adm)
5             IF (OldBranch.type==iSRC) THEN
6                 Replace OldBranch with NewBranch;
7             ELSE
8                 Warning("existing voltage source dominates");
9         OTHERWISE:
10            Merge the two admittances;
11            Merge the two current sources:
12                IF (both are PWL fn) THEN merge to one PWL;
13    END SWITCH
```

```
                         Reduction Core Engine

PROCEDURE Reduction(Ckt, β, ExtNodeList)
// Ckt is the network given;
// β is the given threshold of order of polynomial truncations;
// ExtNodeList is an array containing pointers to external terminals;
// The procedure is to eliminate all the nodes in Ckt,
// except for those in ExtNodeList, using Y-Δ transformation.

1     CALL _seq=NodeOrdering(Ckt);
2     FOR (_node=First(_seq)) DO BEGIN
3         _degree=degree of _node;
4         Γ = all the neighbors of _node;
5         CALL WyeDeltaTransform(_node, Γ, _degree, β);
6         IF (_node has a current src, i. e., _iSrc≠ 0) THEN
7             CALL CurrentSrcTransform(_node, _iSrc, Γ, _degree);
8         Update Ckt:
8             Remove _node and the branches it touches from Ckt;
9             Update the degree of the neighbors of _node;
10        Delete _node from _seq.
11    END FOR
```

Node Ordering

```
PROCEDURE NodeOrdering(Ckt, ExtNodeList, Seq)
// Ckt is a graph given. It will not be changed upon return;
// ExtNodeList is the external terminal list;
// Seq is the resultant elimination sequence (output);
// The procedure is to perform symbolic factorization on Ckt to minimize
// non-zero fill-ins.

1    Initialize s= {n_i|n_i ∈ Ckt and n_i ∉ ExtNodeList};
2    Compute the degree of all the nodes in s;
3    Initialize the node sequence Seq= ∅;
4    WHILE (s≠ ∅) DO BEGIN
5        Pick a node _node with the minimum degree;
6        Number _node and those indistinguishable from it;
7        Append them at the end of Seq and remove them from s;
8        Update Ckt:
9            Remove them and the branches they touched from Ckt;
10           Add branches to Ckt (symbolic LU decompositions);
11           Update the degree of their neighbors;
12   END WHILE
13   RETURN(Seq).
```

Y-Δ Transformation

```
PROCEDURE WyeDeltaTransform(Node, Γ, m, β)
// Node is the node to be eliminated;
// Γ is the set of neighbors of Node;
// m is the number of nodes in Γ;
// β is the given threshold of order of polynomial truncations;
// The procedure is to evaluate Y-Δ transformation on Node. Each
// admittance is computed up to the truncation threshold β. Admittances
// of incident branches of Node are denoted as A_1/B_1, A_2/B_2, ··· A_m/B_m.

1   Compute W_k using (1.8);
2   Compute ω_k using (1.7);
3   FOR (∀n_i, n_j ∈ the set of the neighbors of Node and n_i ≠ n_j) DO
4   BEGIN
5       Y_new = (Y_ik^(k) Y_jk^(k) / ω_k) × (B_1 B_2 ... B_m / W_k);
6       IF (∃ branch between n_i and n_j already) THEN
7           _oldBranch=hash_search(_bhash, n_i, n_j);
8           Y_ij^(k+1) = Y_ij^(k) + Y_new;
9           _oldBranch=BranchMerging(_oldBranch, Y_ij^(k+1));
10          FOR (∀ factor ω_l in both B_i and B_j) DO
11              Cancel ω_l in the numerator and denominator of Y_ij^(k+1).
12      ELSE
13          CALL branch_new(Y_new);
14  END FOR // Loop utill m(m-1)/2 branches are generated.
14  Remove Node and the branches it touches out of Ckt.
```

Current Source Transformation (in the s Domain)

PROCEDURE **CurrentSrcTransform**(`Node`, `ISrc`, Γ, m)
// `Node` is the node to be eliminated;
// `ISrc` is `Node`'s current source;
// Γ is `Node`'s neighbors;
// m is the number of nodes in Γ;
// The procedure is to evaluate current source transformation on `Node`.
// Admittances of incident branches of `Node` are denoted as $\frac{A_1}{B_1}, \frac{A_2}{B_2}, \cdots, \frac{A_m}{B_m}$.

1 Use W_k and ω_k in 1 and 2 of **WyeDeltaTransform**();
2 FOR $\forall n_i \in \Gamma$ DO BEGIN
3 $I_{\text{new}} = \frac{Y_{ik}^{(k)}\texttt{ISrc}}{\omega_k} \times \frac{B_1 B_2 \ldots B_m}{W_k}$;
4 IF (n_i has a current source already, i. e., $I_i^{(k)} \neq 0$) THEN
5 $I_i^{(k+1)} = I_i^{(k)} + I_{\text{new}}$;
6 FOR ($\forall$ factor ω_l in the denominators of `ISrc` and $I_i^{(k)}$) DO
7 Cancel ω_l in the numerator and denominator of $I_i^{(k)}$.
8 ELSE
9 Generate a new current source $I_i^{(k+1)} = I_{\text{new}}$;
10 END FOR // Loop for generating m current sources.

2.2 GRYD Simulator

Waveform Evaluator

PROCEDURE **WvfmEvl**(`Src`, `Rpoly`, `Wvfm`)
// `Src` is the given input source;
// `Rpoly` is the given transfer function (for voltage inputs) or the grounded admittance;
// `Wvfm`, the output, is the response waveforms in Gnu Plot format;
// The procedure takes as input a voltage or current source, and a reduced network. The output is the waveform in time domain. The basic technique is partial fraction decomposition and inverse Laplace transform.

```
1    _response = Src * Rpoly;
2    PartialFracDecomp(_pFracs, _response);
3    FOR (each partial fraction in _pFracs) DO
4        Do inverse Laplace transform to get waveforms in time domain;
5    Overall waveform ← superposition of all the waveforms.
```

Partial Fraction of Rational Function

PROCEDURE **PartialFracDecomp**(`PFracs`, `SRpoly`)
// `PFracs` is the resultant partial fractions (array of rational functions);
// `SRpoly` is the given rational function of s;
// This procedure is to do partial fraction decomposition[a] to `SRpoly`.

```
1    _numerator=SRpoly.Numerator;
2    _denominator=SRpoly.Denominator;
3    Validate assertions:
4        _numerator.NumOfTerms<_denominator.NumOfTerms;
5
6    WHILE (1) DO BEGIN
7        _converge = drpoly(SRpoly.Denominator); // solve for poles
8        IF (NOT _converge) THEN
9            Scale frequency s in SRpoly.Denominator up by 10 times;
10       ELSE BREAK;
11   END WHILE
12
13   Scale SRpoly.Denominator back if necessary;
14   Build a set of linear equations for residues of all partial fractions;
15   Solve for the residues.
```

[a] For a rational function of s: $\frac{a_0+a_1s+\cdots+a_ms^m}{b_0+b_1s+\cdots+b_ns^n}$, where $m \leq n$, its partial fraction can be represented by $\frac{r_0}{s+p_0} + \frac{r_1}{s+p_1} + \cdots + \frac{r_n}{s+p_n}$. p_i is called a pole of the rational function, and r_i is the corresponding residue.

2.3 GRYD Pole Analyzer

Pole Analysis Driver

PROCEDURE **PoleDriver**(β, `InFile`)
// `InFile` is the netlist file of a reduced network;
// β is the given threshold of order of polynomial truncations;
// This procedure is to provide an interactive text-based interface between
// users and background procedures. It provides a set of commands to
// print or save transfer functions from sources to sinks, system natural
// frequencies, system poles, and stabilized counterparts.

1 **Reconstruct**(`_ckt`, `_bhash`, `_nhash`, β, `InFile`);
2 Compute transfer functions $F(s)$ using Y-Δ reductions;
3 Compute system charateristic polynomial $P(s)$;
4 WHILE (`_input`) DO BEGIN
5 CASE pole:
6 *PrintPoles*($P(s)$);
7 CASE stablize pole:
8 **StablizePoly**($Q(s)$, $P(s)$);
9 *PrintPoles*($Q(s)$);
10 CASE transfer function $F(s)$:
11 *PrintTransfer*($F(s)$);
12 CASE stablize transfer function:
13 **ContFrac**($G(s)$, $F(s)$);
14 *PrintTransfer*($G(s)$);
15 CASE natural frequency:
16 *PrintNaturalFreq*($P(s)$);
17 END WHILE

Transfer Function (Model) Stabilization

PROCEDURE **ContFrac**(`TRpoly`, `SRpoly`)
// `TRpoly` is the storage of a stabilized tranfer function (output);
// `SRpoly` is the storage of the given rational function (input);
// For a given transfer function in s, the procedure is to give a stabilized
// rational function. The stabilized one has all the poles on the left-half
// complex plane, or simple roots on the imaginary axis. i. e., it meets the
// Routh-Hurwitz Criteria.

1 CALL **StablizePoly**(`TRpoly.Denominator`, `Spoly.Denominator`);
2 CALL `_moments`=*GetRationalMoments*(TRpoly);
3 Solve for `TRpoly.Numerator`, so that `TRpoly`'s moments agree with
4 `_moments`;

Pole Stabilization

PROCEDURE **StablizePoly**(`Tpoly`, `Spoly`)
// `Tpoly` is the storage of the stabilized polynomial (output);
// `Spoly` is the storage of the given polynomial (input);
// For a given polynomial in s, the procedure is to give a stabilized
// polynomial. The stabilized polynomial has all the roots on the left-half
// complex plane, or simple roots on the imaginary axis, i. e., it meets the
// Routh-Hurwitz Criteria.

1 Build a rational function $f(s) \equiv \frac{sA_1}{B_1}$, where A_1 and B_1 are
2 even polynomials, i. e., polynomials in s^2, and $sA_1 + B_1 =$`Spoly`;
3 Do continuous fraction on $f(s)$ and retrieve adjacent non-negative
4 quotients from the beginning;
5 Construct a rational function of s, $g(s) \equiv \frac{sA_2}{B_2}$, whose quotients of
6 continous fraction is equal to the non-negative ones of $f(s)$;
7 `Tpoly.Denominator`$=B_2 + sA_2$;

2.4 GRYD Network Synthesizer

Admittance Realization

PROCEDURE **Back2SPICE**(`Adm`)
// `Adm` is the given admittance, a rational function of s;
// The procedure is to realize the admittance using a set of pre-defined
// templates. Since this is an optimization process, the template with the
// smallest error will be selected.

1 FOR (template t_i) DO BEGIN
2 TemplateGP(`_err`, t_i, `Adm`);
3 Pick the realized template with the smallest `_err`.

Template Realization via Geometric Programming

PROCEDURE **TemplateGP**(`Err`, `Template`, `Adm`)
// `Err` is the mismatch error from the Geomatric Programming optimization
// process;
// `Template` is the given template circuit;
// `Adm` is the given admittance, a rational function of s;
// Given a template with unknown element values, the procedure is to
// assign values to the elements, under the constraints that all the values
// have to be non-negative, and the resultant admittance should match
// `Adm` as much as possible.

1 Evaluate the symbolic input admittance, $Y_{\texttt{in}}(s)$, of `Template`;
2 Parse the symbolic $Y_{\texttt{in}}(s)$ for product terms $p_1, p_2, \cdots, p_m$, and
3 coefficents of powers of s, $t_1, t_2, \cdots, t_n$;
4 Define the object function: $f = \min \sum_{i=1}^{n} \frac{1}{p_i}$;
5 Define the constraint conditions (1): $g_i = t_i \leq \tilde{t}_i$;
6 Define the constraint conditions (2): all elements have to be
7 non-negative;
8 `Err`=$GPEngine(f,\ g_1,\ g_2,\ \cdots,\ g_n)$.

2.5 Utilities

Circuit Dumping Utility

```
PROCEDURE CktDump(Ckt, β, OutFile)
// Ckt is the circuit to be dumped;
// β is the given threshold of order of polynomial truncations;
// OutFile is the output netlist file;
// The procedure is to dump the current circuit to a file. The dump format
// is defined internally. Correspondingly, we have a reconstruct procedure to
// reload the dump file.

1     Print signature (format tag):
2     Dump branches:
3         IDs of the two end nodes;
4         Types of the branch;
5         PolyDump(branch admittance);
6     Dump nodes;
7         Node ID;
8         IF (node has a current source) THEN dump the source;
9         Neighbor list of the node;
10        ω of the node.
```

```
                         Circuit Reloading Utility

PROCEDURE Reconstruct(Ckt, Bhash, Nhash, β, InFile)
// Ckt is the circuit to be dumped;
// Bhash and Nhash are branch and node hash tables, respectively;
// β is the given threshold of order of polynomial truncations;
// InFile is the input netlist file;
// The procedure is to reload circuit from a file. The dump format is defined
// internally. Correspondingly, we have a dump procedure to create the
// dump file.

1    Read file header of InFile to confirm the signature (file type);
2    Initialize node and branch hash tables Bhash and Nhash;
3    FOR (each branch br in InFile) DO BEGIN
4        Construct branch structure;
5        Link the branch into branch double-linked list;
6        hash_insert(_bhash, br);
7        IF (hash_lookup(_bhash, n1)) THEN
8            Construct node structure;
9            Link the node into node double-linked list;
10           hash_insert(_nhash, n1);
11       Repeat 7 to 10 for n2;
12   END FOR
13   Restore current sources attached to nodes.
```

Polynomial-Expression Arithmatic Operation – Multiplication

```
PROCEDURE PolyProduct(Op1, Op2, Op3, β)
// Op2 and Op3 are two operands;
// Op1 is equal to Op2*Op3;
// β is the given threshold of order of polynomial truncations;
// The procedure is to compute the product of Op2 and Op3. Coefficients
// of orders higher than β will not be computed.

1     /* Reset TermArray of t */
2     memset(t->TermArray, 0);
3
4     /* Compute Op2*Op3 */
5     FOR (i = 0; i < Op2->NumOfTerms; i++) DO BEGIN
6         w = i;
7         IF ((k = β - w) < 0) THEN BREAK;
8         k = MIN(k, Op3->NumOfTerms);
9         _tmp = Op2->TermArray[i];
10        FOR (j = 0; j < k; j++)
11            t->TermArray[w++] += _tmp * Op3->TermArray[j];
12    END FOR
13
14    t->SOrder = Op2->SOrder + Op3->SOrder;
15    t->ExactOrder = Op2->ExactOrder + Op3->ExactOrder - 1;
16
17    a = Op2->NumOfTerms + Op3->NumOfTerms - 1;
18    IF (a > β) THEN
19        t->NumOfTerms = β;
20    ELSE
21        t->NumOfTerms = a;
22    memcpy(Op1, t).
```

Polynomial-Expression Arithmatic Operation – Division

```
PROCEDURE PolyDivide(Op1, Op2, Op3, β)
// Op2 and Op3 are two operands;
// Op1 is equal to Op2/Op3;
// β is the given threshold of order of polynomial truncations;
// The procedure is to compute the quotients of Op2 over Op3. Coefficients
// of orders higher than β will not be computed.

1    j = Op2->SOrder - Op3->SOrder;
2    IF (j < 0) THEN
3        TrapError("SOrder of divider is larger than that of dividend");
4    ELSE Op1->SOrder = j;
5
6    memcpy(t, Op2->TermArray);
7    C = Op3->TermArray;
8    b = Op2->NumOfTerms;   c = Op3->NumOfTerms;
9   d = Op2->ExactOrder - Op3->ExactOrder + 1;
10   IF (d < β) THEN b = d;
11   IF (d ≤ 0) THEN TrapError("Can't divide completely");
12   Op1->NumOfTerms = b;
13   Op1->ExactOrder = d;
14   _effDividendLength = b;
15
16   FOR (i = 0; i < b; i++) DO BEGIN
17       q = t[i]/C[0];
19       Op1->TermArray[i] = q;
20       _dividerTail = MIN(_ffDividendLength, c);
21       _effDividendLength–;
22       FOR (j = i+1, w = 1; w < _dividerTail; j++, w++) DO
23           r = t[j] - q*C[w];
24           IF (r * 1.0e+5 < t[j]) THEN
25               r = 0.0;
26               Warning("Possible round-off error detected");
27           t[j]=r;
28       END FOR
29   END FOR
```

Polynomial-Expression Arithmatic Operation – Addition

```
PROCEDURE PolyAddition(Op1, Op2, Op3, β)
// Op2 and Op3 are two operands;
// Op1 is equal to Op2+Op3;
// β is the given threshold of order of polynomial truncations;
// The procedure is to compute the addition of Op2 and Op3. Coefficients
// of orders higher than β will not be computed.

1     IF (Op2->SOrder < Op3->SOrder) THEN
2         memcpy(_tmp, Op2);
3         h = Op3->SOrder - Op2->SOrder;
4         j = min(β, h+Op3->NumOfTerms);
5         B = Op3->TermArray;
6         t.ExactOrder=max(Op2->ExactOrder, h+Op3->ExactOrder);
7         t.NumOfTerms = max(j, Op2->NumOfTerms);
8     ELSE
9         memcpy(t, Op3);
10        h = Op2->SOrder - Op3->SOrder;
11        j = min(β, h+Op2->NumOfTerms);
12        B = Op2->TermArray;
13        t.ExactOrder=max(Op3->ExactOrder, h+Op2->ExactOrder);
14        t.NumOfTerms = max(j, Op3->NumOfTerms);
15
16    A = t.TermArray;
17    FOR (w = 0, i = h; i < j; i++, w++)
18        A[i] += B[w];
19
20    memcpy(Op1, t).
```

Chapter 3

Data Type and Subroutine Description

Following high-level design document in the previous chapters, we will give the detailed design information in this chapter. The first part is the data structures, which include node and branch structures, node and branch double-linked lists, etc. The second part is to describe major subroutines used in funcation blocks.

3.1 Data Type Definition

3.2 Subroutine Description

```
void
SpiceParser(
  InFp, NodeList, ArcList, SizeOfNodeListPtr, SizeOfArcListPtr,
  SizeOfOutputNodeListPtr, OutputNodeListPtr, SpiceRelicBuffer,
  ArcTable, GndNode)

FILE *          InFp;
NODEPTR         NodeList;
ARCPTR          ArcList;
int *           SizeOfNodeListPtr;
```

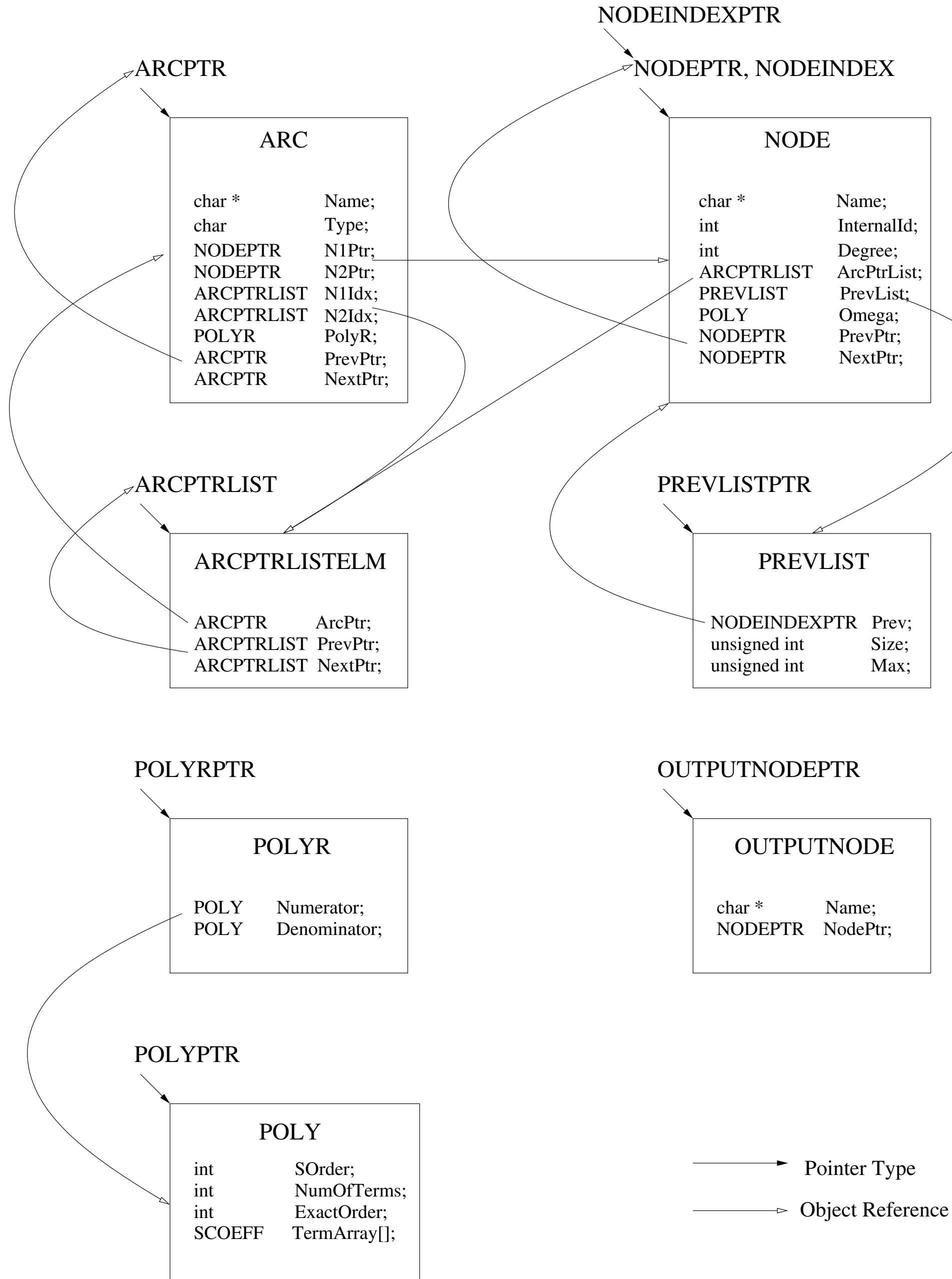


Figure 3.1: Data types in GRYD program.

```
int *           SizeOfArcListPtr;
int *           SizeOfOutputNodeListPtr;
OUTPUTNODEPTR * OutputNodeListPtr;
char **         SpiceRelicBuffer;
st_table *      ArcTable;
NODEPTR *       GndNode;
```

Functionality:
This function is to take as input a SPICE input netlist file and load data into the data structures defined under 'include' directory. Current version supports only basic cards in SPICE, such as resistors(r), capacitors(c), inductors(l), independent voltage sources(v), independent current sources(i), dc analysis(.dc), transient analysis (.tran), simulation output(.print, .probe and .plot) are supported. Nodes (for printing voltages, e.g. V(n1)) and nodes of elements (for printing currents, I(r1)) specified in output cards will be considered as external nodes and will be kept intact. All other nodes will be eliminated in reduction step. Particularly, in this version, we integrate mutual inductanceby introducingK-element into our simullation models. But because partial self and mutual inductance in PEEC model are transformed into K-elements at the same time, thus we could not support partial self inductance (L) and K-elements at the same time.
Interface specifications:
Input:
InFp - handler of input SPICE netlist file.
Inout:
NodeList - double-linked list of nodes, with dummy element at the beginning.
ArcList - double-linked list of arcs, with dummy element at the beginning.
Output:
SizeOfNodeListPtr -
pointer to SizeOfNodeList, whose value will be updated according to the actual size of NodeList upon return.
SizeOfArcListPtr -
pointer to SizeOfArcList, whose value will be updated according to the actual size of ArcList upon return.
SizeOfOutputNodeListPtr -
pointer to SizeOfOutputNodeList, which indicates #entries in OutputNodeList.

OutputNodeListPtr -
If simulation output is specified in the given SPICE netlist file, this array will be pointing a valid address where these simulation output requests are stored.
SpiceRelicbuffer -
This is a buffer storing some uninterpreted lines in the SPICE netlist file, such as the stimuli, the simulation output, .tran or .dc cards, etc.
ArcTable -
an arc hash table to index arcs by names later on.
GndNode -
address to the pointer to ground node.

```
void
_MergeISrcs(s1, s2, sign)

char * s1;
char * s2;
SCOEFF sign;
```

Functionality:
This procedure merges two (2) piece-wise linear functions.
Interface specifications:
Input:
s1 - input string 1.
s2 - input string 2.
sign - direction of the second source
Output:
s1 - synthesized PWL function.

```
void
_NewArc(
    Name, ArcPtr, N1Ptr, N2Ptr, ElmType, SubElmType, FuncTag)

char *   Name;
```

```
ARCPTR   ArcPtr;
NODEPTR  N1Ptr;
NODEPTR  N2Ptr;
int      ElmType;
int      SubElmType;
char *   FuncTag;
```

Functionality:
This procedure fills in the contents of a new arc structure.
Interface specifications:
Input:
Name - Name of the netlist (arc name);
ArcPtr - pointer to the arc construct;
N1Ptr - pointer of the first node;
N2Ptr - pointer of the second node;
ElmType - type of the element. It could be ADMITTANCE, CEXACT, and VEXACT only.
SubElmType - subtype of elements such as RESISTOR, INDUCTOR AND CAPACITOR.
FuncTag - function string.

```
SCOEFF
_CalDet(A, n)

SCOEFFPTR A;
int n;
```

Functionality:
This procedure evaluates of a square matrix a with dimension n.
Interface specifications:
Input:
A - matrix.
n - dimension of A and X.

```
void
_PWL2Pade(m, k, x, y, RationalPtr)

SCOEFFPTR       m;
int             k;
int             x, y;
RATIONALPTR     RationalPtr;
```

Functionality:
This procedure converts a PWL function in time domain into $[x, y]$ Pade approximant. Moments of PWL is evaluated outside the procedure, but as we assume we are processing PWL functions only, there implies a $1/s^2$ term as a factor of result.
Interface specifications:
Input:
m - moment array.
k - #moments given.
$[x, y]$ pade approximant.
Output:
RationalPtr - pointer to the RATIONAL construct saving the approximant.

```
void
_CalPWLMoments(str, m, k)

char * str;
SCOEFFPTR  m;
int k;
```

Functionality:
This procedure evaluates moments of given PWL function.
Interface specifications:
Input:
str - string containing the function description (spice format).
Output:
m - moment array.
k - #moments requested.

```
ARCPTR
_VoltageArcCreation(RefArcPtr, N1Ptr, N2Ptr, nArc, ArcTable)

ARCPTR RefArcPtr;
NODEPTR N1Ptr, N2Ptr;
int * nArc;
st_table * ArcTable;
```

Functionality:
This procedure creates a new voltage source attached to the branch between N1Ptr and N2Ptr. The source is created because a voltage source path exists between nodes N1Ptr and N2Ptr, and one of the sources on the path is to be eliminated (transformed).
Interface specifications:
Input:
RefArcPtr - the insertion point of the branch double-linked list;
N1Ptr, N2Ptr - the two nodes between which a voltage source path can be found;
Inout:
ArcTable - branch hash table;
nArc - #arcs;
Return:
the branch being added.

```
ARCPTR
_CurrentArcCreation(
    OldArcPtr, RefArcPtr, N1Ptr, N2Ptr, nArc, ArcTable)

ARCPTR OldArcPtr, RefArcPtr;
NODEPTR N1Ptr, N2Ptr;
int * nArc;
st_table * ArcTable;
```

Functionality:
This procedure creates a new curent source attached to the branch between N1Ptr and N2Ptr. The source is created because voltage source branch OldArcPtr is to be eliminated (transformed).

Interface specifications:
Input:
OldArcPtr - branch attached to which a voltage source is to be eliminated;
RefArcPtr - the insertion point of the branch double-linked list;
N1Ptr, N2Ptr - the two nodes between which a voltage source path can be found;
Inout:
ArcTable - branch hash table;
nArc - #arcs;
Return:
branch being added.

```
void
_DecoupleCurrentSource(ArcPtr, GndPtr, ArcTable, nArc)

ARCPTR ArcPtr;
NODEPTR GndPtr;
st_table * ArcTable;
int * nArc;
```

Functionality:
This subroutine is to decouple a floating current source, represented by ArcPtr. After the decoupling, the two nodes incident to the floating current source will be connected to the ground with current sources in two opposite directions of the same absolute value.
Interface specifications:
Input:
GndPtr - the pointer to the ground node.
Inout:
ArcPtr - the arc representing the floating current source;
ArcTable - arc hash table;
nArc - the number of arcs;
Output:
N/A;
Side effect:
Nodes incident to ArcPtr will be affected;

SModel of ArcPtr will be de-allocated.

```
void
_InsertOutputNodeList(
  Name, NodeListPtr, OutputNodeTable, ListSizePtr, ListMaxPtr)

char* Name;
OUTPUTNODEPTR * NodeListPtr;
st_table *OutputNodeTable;
int *ListSizePtr, *ListMaxPtr;
```

Functionality:
This subroutine registers node name in OutputNodeTable and NodeList. If the list has no empty slot left, it will be expanded by NUMOFOUTPUTNODE. values pointed by ListSizePtr and ListMaxPtr will be updated accordingly.
Interface specifications:
Input:
Name - name of the node to be inserted.
Inout:
NodeListPtr - pointer to output node list;
OutputNodeTable - an output node hash table;
ListSizePtr - pointer to an integer ListSize(current size of the node list);
ListMaxPtr - pointer to an integer Listmax(capacity of the node list).

```
void
_InsertOutputNodeListWithNodePtr(
    Name, NodePtr, NodeList, ListSizePtr, ListMaxPtr)

char* Name;
NODEPTR NodePtr;
OUTPUTNODEPTR NodeList;
int *ListSizePtr, *ListMaxPtr;
```

Functionality:
This subroutine registers node name ’Name’ and the node pointer ’NodePtr’

in NodeList only (different from its brother subroutine above). If the list has no empty slot left, it will be expanded by NUMOFOUTPUTNODE. values pointed by ListSizePtr and ListMaxPtr will be updated accordingly.
Interface specifications:
Input:
Name - name of the node to be inserted.
NodePtr - the pointer to the node to be inserted.
Inout:
NodeList - output node list;
ListSizePtr - pointer to an integer ListSize(current size of the node list);
ListMaxPtr - pointer to an integer Listmax(capacity of the node list).

```
void
_AppendRelicBuffer(
    Str, RelicBuffer, nEmptyBufferPtr, nTotalBufferPtr)

char *Str, *RelicBuffer;
int *nEmptyBufferPtr, *nTotalBufferPtr;
```

Functionality:
This subroutine appends a string 'Str' to string buffer 'RelicBuffer'. If the buffer does not have enough empty space, it will be expanded by RELICBUFFERSIZE. values pointed by nEmptyBufferPtr and nTotalBufferPtr will be updated accordingly.
Interface specifications:
Input:
Name - name of the node to be inserted.
NodePtr - the pointer to the node to be inserted.
Inout:
NodeList - output node list;
ListSizePtr - pointer to an integer ListSize(current size of the node list);
ListMaxPtr - pointer to an integer Listmax(capacity of the node list).

```
void _SaveMList(mListPtr, str)
```

```
MLISTPTR mListPtr;
char * str;
```

Functionality:
This subroutine inserts a MLIST element into mList. Memory will be allocated by the subroutine itself.
Interface specifications:
Input:
mListPtr - the dummy head of mList (pointer);
str - the mutual inductor card (spice input line).

```
void
Reduction(
  NodeList, ArcList, SizeOfNodeIndexArray, NodeIndexArray,
  SizeOfNodeList, SizeOfArcList, ArcTable,
  SizeOfOutputNodeList, OutputNodeList, GndPtr)

NODEPTR NodeList;
ARCPTR ArcList;
int SizeOfNodeIndexArray;
NODEINDEXPTR NodeIndexArray;
int * SizeOfNodeList;
int * SizeOfArcList;
st_table * ArcTable;
int SizeOfOutputNodeList;
OUTPUTNODEPTR OutputNodeList;
NODEPTR GndPtr;
```

Functionality:
The procedure is used to apply a series of wye-delta reduction by eliminating nodes in the given circuit. Suppose $y_1(s), ...y_k(s)$ are k given admittance connected from the corresponding k nodes to node j. After eliminating node j, the k nodes form a clique and the admittance from any two nodes x, y among the k nodes are:

$$Y_{xy}(s) = \frac{Y_x(s)Y_y(s)}{Y_1(s) + Y_2(s) + \cdots + y_k(s)}$$

We use pairwise method to compute the summation of $y_i(s)$.
Interface specifications:
Input:
NodeList - linked list of nodes.
ArcList - linked list of arcs. Its size will be compressed dynamically along the reduction process.
SizeOfNodeIndexArray - size of the node index array.
NodeIndexArray - this is actually the reordering vector generated by MMD module. The vector suggests the order in which eligible nodes in the circuit are eliminated.
ArcTable - Hash table of arcs.
SizeOfOutputNodeList - size of the output node array.
OutputNodeList - linked list of output nodes.
GndPtr - pointer to the ground node.
Inout:
SizeOfNodeList - It is adjusted to reflect the change of the size of NodeArray upon return.
SizeOfArcList - It is adjusted to reflect the change of the size of ArcArray upon return.

```
void
_CancelPowerS(PolyRPtr)

POLYRPTR PolyRPtr;
```

Functionality:
Given a rational function of s, this subroutine cancels the redundant zeros or poles at s=0 in the numerator and denominator. For example,

$$\frac{s^n(a_0 + a_1 s + \cdots)}{s^m(b_0 + b_1 s + \cdots)} = \frac{a_0 + a_1 s + \cdots}{s^{m-n}(b_0 + b_1 s + \cdots)}, \text{if } m > n.$$

Interface Specifications:
Input:
PolyRPtr - pointer to the given rational function.

```
char
_LightRationalAddition(
  D, S1, S2, BigOmegaPtr, PrevList, BitWord, nBitWord)

POLYRPTR D, S1, S2;
POLYPTR BigOmegaPtr;
unsigned int * BitWord;
int nBitWord;
PREVLISTPTR PrevList;
```

Funcationality:

$$\frac{a}{w \cdot b} + \frac{c}{w \cdot d} = \frac{a \cdot d + b \cdot c}{w \cdot b \cdot d} = \frac{x}{y}.$$

Interface specifications:
Input:
S1, S2 - pointers to two rational expressions to be added.
PrevList - PrevList of the parent node.
BitWord - array with corresponding bit words set.
nBitWord - size of the array above.
Output:
D - pointer to the resultant rational expression. Memory for the structure has to be allocated OUTSIDE the subroutine.

```
void
_GenCurrentSrc(N1, N2, YNum, YDen, BigOmegaPtr, BitArray, x, y)

NODEPTR N1, N2;
POLYPTR YNum, YDen;
POLYPTR BigOmegaPtr;
unsigned int * BitArray;
int x, y;
```

Functionality:
This subroutine evaluates the current source of N2 resulted from the elimination of N1. The new current source will be merged with the existing one, if any.
Interface Specifications:
Input:
N1 - Node to be eliminated;
N2 - the naboring node;
YNum - Numerator of the scalar $y_i / \sum y$;
YDen - Denominator of the scalar $y_i / \sum y$;
BitArray - an array to check for common factors between two nodes;
x - x dimension of BitArray(# nodes in the system);
y - y dimension of BitArray(dimension of each cell).

```
char
_GetCancellingOmega(OmegaPtr, PrevList, BitWord, BitWordSize)
POLYPTR OmegaPtr;
PREVLISTPTR PrevList;
unsigned int * BitWord;
int BitWordSize;
```

Functionality:
This subroutine multiplies a series of omega's. These omega's are associated with nodes. And they are redundant to newly created arcs.
Interface Specifications:
Input:
PrevList - array of prev nodes.
BitWord - array of markers, with each bit indicating the existence of corresponding omega in Omega.
BitWordSize - the dimension of the array BitWord.
Output:
OmegaPtr - output operand

```
char
```

```
_GetCancellingOmega4ISrc(
  OmegaPtr,ISrcPtr,PrevList,BitWord,BitWordSize)

POLYPTR OmegaPtr, ISrcPtr;
PREVLISTPTR PrevList;
unsigned int * BitWord;
int BitWordSize;
```

Functionality:
This subroutine is similar to _GetCancellingOmega(). The difference is that this subroutine will check if nodes in PrevList had a current source associated with, before collecting their Omegas. Besides, the denominators of the current sources will also be collected.
Interface Specifications:
Input:
PrevList - array of prev nodes.
BitWord - array of markers, with each bit indicating the existence of corresponding omega in Omega.
BitWordSize - the dimension of the array BitWord.
Output:
OmegaPtr - multiplication of all omegas.
ISrcPtr - multiplication of all current sources' denominators.

```
void
_PrintAdmittanceMoments(Op, n)

POLYRPTR Op;
int n;
```

Functionality:
This subroutine is a debug subroutine. It prints time moments of an admittance in rational form.
Interface Specifications:
Input:
Op - the input admittance in rational form.
n - the number of moments required.

```
void
_GenCurrentSrc(N1, N2, YNum, YDen, BigOmegaPtr, BitArray, x, y)

NODEPTR N1, N2;
POLYPTR YNum, YDen;
POLYPTR BigOmegaPtr;
unsigned int * BitArray;
int x, y;
```

Functionality:
This subroutine evaluates the current source of N2 resulted from the elimination of N1. The new current source will be merged with the existing one, if any.
Interface Specifications:
Input:
N1 - Node to be eliminated;
N2 - the naboring node;
YNum - Numerator of the scalar $y_i/\sum y$;
YDen - Denominator of the scalar $y_i/\sum y$;
BitArray - an array to check for common factors between two nodes;
x - x dimension of BitArray(# nodes in the system);
y - y dimension of BitArray(dimension of each cell);

```
void
SndErrMsg(MsgCode, LineNo, Line, OptMsg, Action)

unsigned int MsgCode;
int LineNo;
char * Line;
char * OptMsg;
unsigned int Action;
```

Functionality:
This procedure is to print an error message to stdout.

Interface specifications:
Input:
MsgCode - Error message ID.
LineNo - Line number in the input netlist file. It works only with the parser, when it is larger than zero.
Line - Content of LineNo in the input netlist file. It works only with the parser, when it is not null.
OptMsg - An optional message that replaces default one here.
Action - Action to take upon the error. WARNING means warning, FATAL exit.

```
void
PrintNeighborNodes(NodePtr)

NODEPTR NodePtr;
```

Functionality:
Debug subroutine. This is used to print out the neighborhood nodes (names) of a given node.
Interface specifications:
Input:
NodePtr - Pointer to the node whose neighbors are to be printed.

```
void
PrintPoly(PolyPtr)
POLYPTR PolyPtr;
```

Functionality:
Debug subroutine. This is used to print out the given polynomial function to standard output terminal.
Interface specifications:
Input:
PolyPtr- Pointer to a polynomial function (POLY type).

```
void
PolyNeg(PolyPtr)

POLYPTR PolyPtr;
```

Functionality:
This subroutine is to multiply -1 to the polynomial expression refered to by PolyPtr. The is a unary operation. The operand (input) will be changed when the subroutine returns.
Interface specifications:
Input:
PolyPtr- Pointer to a POLY polynomial function.

```
void
RationalNeg(PolyRPtr)

POLYRPTR PolyRPtr;
```

Functionality:
This subroutine is to multiply -1 to the rational expression refered to by PolyRPtr. The is a unary operation. The operand (input) will be changed when the subroutine returns.
Interface specifications:
Input:
PolyRPtr - Pointer to a POLYR rational function.

```
void
PrintPolyR(PolyRPtr)
POLYRPTR PolyRPtr;
```

Functionality:
Debug subroutine. This is used to print out the given rational function to standard output terminal.

Interface specifications:
Input:
PolyRPtr - Pointer to a rational function (POLYR type).

```
void
PrintPoly2File(OutFp, PolyPtr)

FILE * OutFp;
POLYPTR PolyPtr;
```

Functionality:
This is used to print out the given polynomial function to a file.
Interface specifications:
Input:
OutFp - handle of output file.
PolyPtr - Pointer to a polynomial function (POLY type).

```
void
PrintPolyR2File(OutFp, PolyRPtr)

FILE * OutFp;
POLYRPTR PolyRPtr;
```

Functionality:
This is used to print out the given rational function to a file.
Interface specifications:
Input:
OutFp - handle of output file.
PolyRPtr - Pointer to a rational function (POLYR type).

```
void
BeginTiming(i)

int i;
```

Functionality:
This routine, together with EndTiming, is used to get CPU timing information. Usage:

```
BeginTiming(N);
<your procedure here>
runtime = EndTiming(N);
```

Interface specifications:
Input:
i - Clock ID.

```
void
PntWelcomeMsg(Str)

char *Str;
```

Functionality:
This routine is to print Welcome message to the standard output.
Interface specifications:
Input:
Str - Welcome String (char * type).

```
void
SimpleUpdateNode(WorkingNodePtr, ArcPtr)

NODEPTR WorkingNodePtr;
ARCPTR ArcPtr;
```

Functionality:
This procedure associates an arc pointed by ArcPtr with an node pointed by NodePtr. It is an simple version of UpdateNode(). It is used in parser module. As in parser section, we do not need to keep arcs in order, so node updating is done by simply inserting new arc at the top of the ArcPtrList of the working node.

Interface specifications:
Input:
WorkingNodePtr - Pointer to the working node entry in the node array. This node is the one to be updated.
ArcPtr - Pointer to the arc to which the node pointed by NodePtr is connected. This arc is newly generated.

```
void
UpdateNode(WorkingNodePtr, NewNeighborNodePtr, ArcPtr)

NODEPTR WorkingNodePtr, NewNeighborNodePtr;
ARCPTR ArcPtr;
```

Functionality:
This procedure associates an arc pointed by ArcPtr with an node pointed by NodePtr.
Interface specifications:
Input:
WorkingNodePtr - Pointer to the working node entry in the node array. This node is the one to be updated.
NewNeighborNodePtr - Pointer to the node that is newly hooked up to the working node.
ArcPtr - Pointer to the arc to which the node pointed by NodePtr is connected. This arc is newly generated.

```
void
GraphTraversal(NodeList, SizeOfNodeList)

NODEPTR NodeList;
int SizeOfNodeList;
```

Functionality:
This is a testing subroutine. Use it to traverse the graph represented by the node array and the arc array you have already established. As you might have

imagined, this subroutine checks the validity and correctness by traversing the graph. The final decision maker is YOU. Compare the printout of this subroutine with your SPICE netlist file.
Interface specifications:
Input:
NodeList - pointer to the node list.
SizeOfNodeList - #entries in the node list.

```
void
OutputNodeTraversal(OutputNodeList, ArraySize)

OUTPUTNODEPTR OutputNodeList;
int ArraySize;
```

Functionality:
This is a testing subroutine. It's used to test output node parsing operated in SpiceParser.
Interface specifications:
Input:
OutputNodeList - Array of output nodes;
ArraySize - The size of the array.

```
void
PolyProduct(Op1, Op2, Op3)
POLYPTR Op1, Op2, Op3;
```

Functionality:
This subroutine implements just another type of polynomial product. It is of type:

```
        POLY <- POLY * POLY
```

Interface specifications:
Input:
Op2, Op3 - two operands.

Output:
Op1 - Where the result of B*C is stored.

```
void
PolyAddition(Op1, Op2, Op3)

POLYPTR Op1, Op2, Op3;
```

Functionality:

```
Op1 = Op2 + Op3
```

where Op2 and Op3 are polynomial expressions.

```
void
RationalAddition(Op1, Op2, Op3)

POLYRPTR Op1, Op2, Op3;
```

Functionality:

$$\frac{A_1}{B_1} = \frac{A_2}{B_2} + \frac{A_3}{B_3} = \frac{A_2B_3 + B_2A_3}{B_2B_3}$$

A_1 and B_1 may have common factors. But the common factors, if any, will not contain s.
Interface specifications:
Input:
A2 - Numerator of the first operand.
a2 - Order of polynomial A2.
B2 - Denominator of the first operand.
b2 - Order of polynomial B2.
A3 - Numerator of the second operand.
a3 - Order of polynomial A3.
B3 - Denominator of the second operand.

b3 - Order of polynomial B3.
Output:
A3 - Numerator of the result.
a3 - Order of polynomial A3.
B3 - Denominator of the result.
b3 - Order of polynomial B3.

```
void
RationalReciprocal(Op1, Op2)
POLYRPTR Op1, Op2;
```

Functionality:
This subroutine computes the reciproacl of a rational function. It's nothing but an exchange of numerator and denominator.
Interface Specifications:
Input:
Op2 - input operand
Output:
Op1 - output operand

```
void
RationalProduct(Op1, Op2, Op3)
POLYRPTR Op1, Op2, Op3;
```

Functionality:
This subroutine computes product of two rational polynomial functions.
Interface Specifications:
Input:
Op2 - input operand
Op3 - input operand
Output:
Op1 - output operand

```
void
PolyPower(Op1, Op2, i)
POLYPTR Op1, Op2;
int i;
```

Functionality:
This subroutine computes the power of polynomials.
Interface Specifications:
Input:
Op2 - input operand
i - order of power
Output:
Op1 - output operand

```
void
PolyDivide(Op1, Op2, Op3)
POLYPTR Op1, Op2, Op3;
```

Functionality:
This subroutine devides the given dividend with the given divider.
Interface Specifications:
Input:
Op2 - input dividend
Op3 - input divider
Output:
Op1 - output quotient

```
int
GetTVPair(sPtr, tPtr, vPtr)

char ** sPtr;
SCOEFFPTR tPtr;
SCOEFFPTR vPtr;
```

Functionality:
Extracts one pair of [time, val] from the string pointed by *sPtr. *sPtr will also be advanced over these two values.
Interface specifications:
Input:
sPtr - *sPtr is the address of the string.
Output:
tPtr - pointer to the time variable.
vPtr - pointer to the val variable.
sPtr - *sPtr will be advanced.
Returns:
0 - failed
1 - succeeded

```
int
IsZeroRational(PolyRPtr)

POLYRPTR PolyRPtr;
```

Functionality:
This subroutine checks if the given rational function used to be reset by ResetRational() or not. If it was, 1 is returned; otherwise, 0 is returned.
Interface Specifications:
Input:
PolyRPtr - A pointer to POLYR structure
Returns:
0 - if PolyRPtr did not use to be reset by ResetRational();
1 - otherwise.

```
int
SetPoly(PolyPtr)

POLYPTR PolyPtr;
```

Functionality:
This subroutine sets the POLY structure refered by PolyPtr to one.
Interface Specifications:
Input:
PolyPtr - A pointer to POLY structure;
Returns:
0 - if PolyPtr is NULL
1 - if succeed.

```
int
ResetRational(PolyRPtr)

POLYRPTR PolyRPtr;
```

Functionality:
This subroutine resets the POLYR structure refered by PolyRPtr. The numerator will be reset to zero, and the denominator, to avoid numerical errors, will be reset to one.
Interface Specifications:
Input:
PolyRPtr - A pointer to POLYR structure
Returns:
0 - if PolyRPtr is NULL
1 - if succeed.

```
void
ArcDeletion(ArcPtr, UpdNodePtr, nArc, ArcTable)

ARCPTR  ArcPtr;
NODEPTR UpdNodePtr;
int * nArc;
st_table * ArcTable;
```

Functionality:
This subroutine deallocates the memory of an arc and delink it from the double link list.

Interface Specifications:
Input:
ArcPtr - pointer to the node to be deallocated.
UpdNodePtr - pointer to the node needed to be updated (degree, arcptrlist, etc.)
Inout:
nArc - pointer to a counter of arcs. Will be updated when the subroutine returns.
Remarks:
Deallocation of arc registry in ArcTable has not been implemented.

```
void
FreeArc(ArcPtr, ArcTable, nArc)

ARCPTR ArcPtr;
st_table ArcTable;
int * nArc;
```

Functionality:
This subroutine deallocates the memory of an arc and delink it from the arc double link list.
Interface Specifications:
Input:
ArcPtr - pointer to the arc to be deallocated.
Inout:
ArcTable - pointer to the arc hash table.
nNode - Pointer to the counter of nodes.

```
void
FreeNode(NodePtr, nNode)

NODEPTR NodePtr;
int * nNode;
```

Functionality:
This subroutine deallocates the memory of a node and delink it from the node double link list.
Interface Specifications:
Input:
NodePtr - pointer to the node to be deallocated.
Inout:
nNode - Pointer to the counter of nodes.

```
void
DelinkArcPtrList( Index )

ARCPTRLIST Index;
```

Functionality:
This subroutine delinks Index from the ArcPtrList that it belongs to, and de-allocates memory of Index.
Interface Specifications:
Input:
Index - the pointer to the cell that needs to be delinked.

```
void
GetRationalMoments(A, M)

POLYRPTR A;
SCOEFFPTR M;
```

Functionality:
This subroutine evaluates moments of a given rational function A, and returns these moments in array M. SOrder fields will be ignored. A will NOT be changed.
Interface Specifications:
Input:
A - input rational function;

Output:
M - moments in the ascendent order.

```
char* GetSpiceValue(S, V)

char* S;
SCOEFFPTR V;
```

Functionality:
This subroutine is to retrieve a value in SPICE format from a string. A SPICE value is usually tailed with a scale factor and/or a dimension. The subroutine retrieves the value and scales it as indicated by the scale factor, if any. The number will be returned as a output parameter.
Interface Specifications:
Input:
S - input string *Output:*
V - value extracted
Return:
Address to which the string 'S' has been read. NULL if no value is extracted.

```
void
Dump( OutFp, NodeList, SizeOfNodeList, ArcList, SizeOfArcList,
      OutputNodeArray, SizeOfOutputNodeArray, SpiceRelic )

FILE *          OutFp;
NODEPTR         NodeList;
int             SizeOfNodeList;
ARCPTR          ArcList;
int             SizeOfArcList;
OUTPUTNODEPTR   OutputNodeArray;
int             SizeOfOutputNodeArray;
char *          SpiceRelic;
```

Functionality:
This procedure scans the reduced network and writes it back into a file. The

content includes all admittance functions (branches) in the network with original node names. And the spice relic buffer is also dumped.
Interface specification:
Input:
OutFp - file handler. The file is open to write SPICE netlist.
NodeList - array of nodes.
SizeOfNodeList -
ArcList - array of arcs.
SizeOfArcList -
SpiceRelic - this buffer stores the SPICE cards that are not interpreted by SpiceParser module. Basically these cards are for simulation output specifications and stimuli.

```
void
_PolyDump(Fp, PolyPtr)

  FILE *        Fp;
  POLYPTR       PolyPtr;
```

Functionality:
This function is to dump the polynomial expression PolyPtr to a file specified by Fp.
Interface specification:
Input:
Fp - Handler to the file where PolyPtr is to be dumped.
PolyPtr - Pointer to the polynomial expression to be dumped.

```
void
SortRoots(RootR, RootI, start, end)

SCOEFFPTR RootR, RootI;
int start, end;
```

Usage:
Classical sorting algorithm used for sorting roots. The sorting keys used are

real parts of the roots only. As distinct roots are assumed, it is your job to validate the assumption. It is easy to be done afer the sorting.

Input:
RootR - Real part of the root array. They are the keys.
RootI - Imaginary part of the root array. Their order will be rearranged according to RootR.
start - starting point of list RootR.
end - ending point of list RootR.
$[start, end]$ specifies the sublist of RootR.
Output:
Sorted RootR and RootI.

```
int
MyPartition(RootR, RootI, start, end)

SCOEFFPTR RootR, RootI;
int start, end;
```

Usage:
Auxiliary function of _Sortroots. Partition the unsorted list RootR into two unsorted ones using the first entry RootR[start] as pivot. Every entry in the first sub-list then will be smaller than the pivot, while every entry in the second sub list is larger than the pivot.

Input:
RootR - unsorted list.
RootI - will be rearranged according to RootR.
start - starting point of list RootR.
end - ending point of list RootR. [start, end] specifies the sublist of RootR.
Output:
function returns the pivot position in RootR.

```
void GetPartialProducts(n, Factors, WorkBuff, PolyArray)

int n;
```

```
POLYPTR Factors;
POLYPTR WorkBuff;
POLYPTR PolyArray;
```

Functionality:
This subroutine delivers an algorithm evaluating n partial products of given n terms with $O(n)$ complexity. We give an example to explain: Given: $B_1, B_2, ..., B_n$ Wanted:

$$B_2 * B_3 * ... * B_n,$$
$$B_1 * B_3 * ... * B_n, ...,$$
$$B_1 * B_2 * ... * B_{n-1}.$$

An naive approach may cause the computation time go up to $O(n^2)$.

For the given sequence $B_1, B_2, B_3, B_4, B_5, B_6, B_7$, and B_8, We compute and save the result of

$$B_1 * B_2, (B_1 * B_2) * B_3 * B_4, (B_1 * B_2 * B_3 * B_4) * B_5 * B_6,$$

and

$$B_7 * B_8, B_5 * B_6 * (B_7 * B_8), B_3 * B_4 * (B_5 * B_6 * B_7 * B_8).$$

It takes us $O(n)$ to get these intermediate results.

Then we come to enumerate all the partial terms:

$$(B1 * B2 * B3 * B4 * B5 * B6) * B7, \quad (B1 * B2 * B3 * B4 * B5 * B6) * B8,$$
$$(B1 * B2 * B3 * B4) * B5 * (B7 * B8), (B1 * B2 * B3 * B4) * B6 * (B7 * B8),$$
$$(B1 * B2) * B3 * (B5 * B6 * B7 * B8), (B1 * B2) * B4 * (B5 * B6 * B7 * B8),$$
$$B1 * (B3 * B4 * B5 * B6 * B7 * B8), \quad B2 * (B3 * B4 * B5 * B6 * B7 * B8).$$

There are n partial terms and each of them takes us constant time. So totally we have an $O(n)$ algorithm.
Interface specifications:
Input:
n - Number of factor terms.
Factors - An array of with n factor terms.
WorkBuff - An array of POLY elements, a working buffer with size = 2n.

Output:
PolyArray - An array with n partial product terms.

```
SUBROUTINE DRPOLY(DEGREE,COEFF,ZEROR,ZEROI, RET)

 FINDS THE ZEROS OF A POLYNOMIAL WITH REAL COEFFICIENTS.

  INPUTS -

     COEFF         -  DOUBLE PRECISION VECTOR OF THE COEFF-
                      ICIENTS IN ORDER OF DECREASING POWERS.

     DEGREE        -  INTEGER DEGREE OF POLYNOMIAL.

  OUTPUTS -

     ZEROR, ZEROI  -  DOUBLE PRECISION VECTORS OF REAL
                      AND IMAGINARY PARTS OF THE ZEROS.

 THE SUBROUTINE USES SINGLE PRECISION CALCULATIONS
 FOR SCALING, BOUNDS AND ERROR CALCULATIONS. ALL
 CALCULATIONS FOR THE ITERATIONS ARE DONE IN DOUBLE
 PRECISION.
```

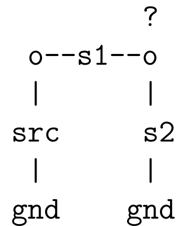
```
void
_GetTransFn( d, s1, s2, Omega)

  POLYRPTR d, s1, s2;
  POLYPTR Omega;
```

Functionality:
This function is to evaluate a transfer function. s1 and s2 are two admittance in series and d is where the transfer function is to be stored. The fomula we used here is simply:

$$d = s1/(s1 + s2)$$

d will not be changed UNTIL the function returns. Please note that the ORDER of series s1 and s2 is important.



Interface Specifications:
Input:
s1 - first admittance.
s2 - second admittance.
Output:
d - Output transfer function.

```
void
_PrnPoles( OutFp, PolyPtr, n)

  FILE * OutFp;
  POLYPTR PolyPtr;
  int n;
```

Functionality:
This routine solves polynomial 'PolyPtr' up to 'n'-th order. You don't need to worry that 'n' is larger than the order of 'PolyPtr', because this is checked before the routine gets called.
Interface Specifications:
Input:
OutFp - Handle of output file.
PolyPtr - Pointer to the polynomial.
n - See 'Functionality' above.

```
int
_IsNeighbor(N1, N2)

NODEPTR N1, N2;
```

Functionality:
This routine is to check if a node N1 is another one N2's nabor.
Specifications:
Input:
N1 - the first node;
N2 - the second node;
Output:
1 - N1 is N2's nabor;
0 - otherwise.

```
void
_GenSumY( Sum, Y1, Y2, Omega )

POLYRPTR Sum, Y1, Y2;
POLYPTR Omega;
```

Functionality:
This subroutins is a similar to a regular rational addition routine. But it considers common-factor existence in the denominators of the two rational functions, and assures that the resultant rational function has no such common factors in the numerator and denominator.
Specifications:
Input:
Y1, Y2 - two input rational functions (points);
Omega - common factor, NULL for its absence.
Output:
Sum - the resultant rational function.

Chapter 4

Summary and Support

This report is a programmer's manual, aimed at keeping on refining the current version of the package and bring it to the next level. The report is devided into 3 chapters, describing the development of the package from high-level document to detailed subroutine interface specifications. We hope the manual is written in such a style, so that readers can fully understand the structure and the details of the implementation.

GRYD package is written and currently maintained by Zhanhai Qin with UC San Diego. To report bugs or send comments, please contact us by email: zqin@cs.ucsd.edu, or by phone: 1-858-534-8174.

Bibliography

[1] Z. Qin, C.-K. Cheng, "Linear network reduction via generalized Y-Δ transformation: theory," technical report No. 2002-0706, University of California, San Diego. May, 2002.

[2] Z. Qin, C.-K. Cheng, "RCLK-VJ network reduction with hurwitz polynomial approximation," proceedings of asia and south pacific design automation conference, pp. 283-91, 2003.

[3] L. W. Nagel, "SPICE2, a computer program to simulate semiconductor circuits," technical report ERL-M520, UC-Berkeley, May 1975.

[4] Z. Qin, C.-K. Cheng, "Linear network reduction via generalized Y-Δ transformation: applications," technical report No. 2003-xxxx, University of California, San Diego. February, 2003.

[5] T.H. Chen, C.P. Chen, " Efficient large-scale power grid analysis based on preconditioned Krylov-subspace iterative methods," *proceedings of design automation conference*, pp.559–62, 2000.

[6] Z. Qin, Z. Zhu, and C.K. Cheng, "Efficient transient analysis for large linear networks," *workshop on synthesis and system integration of mixed information technologies*, pp.293–00, 2001.

[7] J.W. Demmel, J.R. Gulbert, X.S. Li, "SuperLU user's guide," *National Energy Research Scientific Computing Center,* http://www.nersc.gov/ xiaoye/SuperLU, 1999.

[8] S. R. Nassif, J. N. Kozhaya, "Fast power grid simulation," *proceedings of design automation conference*, pp.156–61, 2000.

[9] J. N. Kozhaya, S. R. Nissif, and F. N. Najm, "Multigrid-like technique for power grid analysis," *international conference on computer aided design*, pp.480–7, 2001.

[10] L.T. Pillage and R.A. Rohrer, "Asymptotic waveform evaluation for timing analysis," *IEEE transactions on computer-aided design of integrated circuits and systems*, vol. 9, no. 4, pp.352–66, Apr. 1990.

[11] S. Lin, and E. S. Kuh, "transient simulation of lossy interconnects based on the recursive convolution formulation," *IEEE transactions on Circuits and Systems I: Fundamental Theory and Applications*, vol.39, pp.879–92, Nov. 1992.

[12] H. Liao, W. Dai, R. Wang, and F. Y. Chang, "S-parameter based macro model of distributed-lumped networks using exponentially decayed polynomial function," *proceedings of design automation conference*, pp.726–31, 1993.

[13] H. Liao, W. W.M. Dai, "Partitioning and reduction of RC interconnect networks based on scattering parameter macromodels," *proceedings of design automation conference*, pp.704–9, 1995.

[14] P. Feldmann, R. W. Freund, "Reduced-order modeling of large linear subcircuits via a block Lanczos algorithm," *proceedings of design automation conference*, pp.376–80, 1995.

[15] D. L. Boley, "Krylov space methods on state-space control models," *circuits systems and signal processing*, vol. 13, no.6, pp.733–58, 1994.

[16] A. Odabasioglu, M. Celik, L. T. Pillage, "PRIMA: passive reduced-order interconnect macromodeling algorithm," *IEEE transactions on computer-aided design of integrated circuits and systems*, vol.17, pp.645–54, Aug. 1998.

[17] K. J. Kerns, I. L. Wemple, A. T. Yang, "Stable and efficient reduction of substrate model networks using congruence transforms," *international conference on computer aided design*, pp. 207–14, 1995.

[18] K. J. Kerns, "Accurate and stable reduction of RLC networks using split congruence transformations," Ph.D. dissertation, Univ. Washington, Sept. 1996.

[19] S. Chan, "Introductory topological analysis of electrical networks," *Holt, Rinehart and Winston, Inc.*, 1974.

[20] Y. I. Ismail, E. G. Friedman, "DTT: direct truncation of the transfer function — an alternative to moment matching for tree structured interconnect," *IEEE transactions on computer-aided design of integrated circuits and systems*, vol. 21, No. 2, Feb. 2002.

[21] S. B. Akers, Jr., "The use of wye-delta transformations in network simplification," *operations research*, 8, pp.311-23, 1960.

[22] A. Devgan, H. Ji, W. Dai, "How to efficiently capture on-chip inductance effects: introducing a new circe it element K," *international conference on computer aided design*, pp.150-5, 2000.