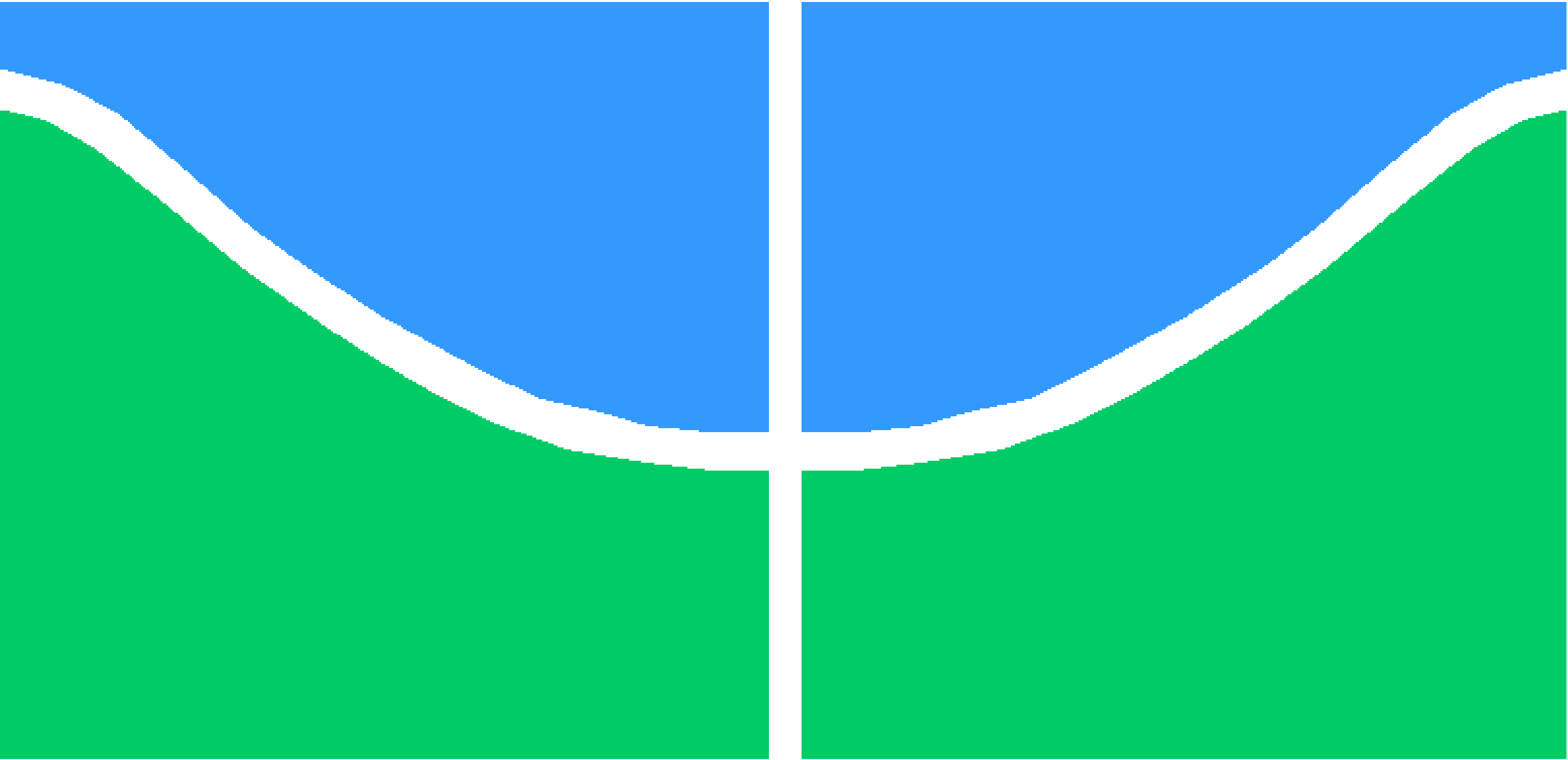


DISSERTAÇÃO DE MESTRADO

ALGORITMO DE SÍNTESE DE CIRCUITOS ANALÓGICOS TRANSLINEARES UTILIZANDO DECOMPOSIÇÃO NÃO-PARAMÉTRICA

Diogo Andrade

Brasília, agosto de 2014

UNIVERSIDADE DE BRASÍLIA

FACULDADE DE TECNOLOGIA

UNIVERSIDADE DE BRASÍLIA
FACULDADE DE TECNOLOGIA
DEPARTAMENTO DE ENGENHARIA ELÉTRICA

ALGORITMO DE SÍNTESE DE CIRCUITOS ANALÓGICOS TRANSLINEARES UTILIZANDO DECOMPOSIÇÃO NÃO-PARAMÉTRICA

DIOGO ANDRADE

ORIENTADOR: SANDRO AUGUSTO PAVLIK HADDAD

DISSERTAÇÃO DE MESTRADO EM ENGENHARIA ELÉTRICA

PUBLICAÇÃO: PPGEA.DM – 574/14

BRASÍLIA/DF: AGOSTO – 2014

UNIVERSIDADE DE BRASÍLIA
FACULDADE DE TECNOLOGIA
DEPARTAMENTO DE ENGENHARIA ELÉTRICA

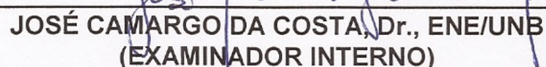
ALGORITMO DE SÍNTESE DE CIRCUITOS ANALÓGICOS TRANSLINEARES UTILIZANDO DECOMPOSIÇÃO NÃO-PARAMÉTRICA

DIOGO ANDRADE

DISSERTAÇÃO DE MESTRADO SUBMETIDA AO DEPARTAMENTO DE ENGENHARIA ELÉTRICA DA FACULDADE DE TECNOLOGIA DA UNIVERSIDADE DE BRASÍLIA, COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE MESTRE.

APROVADA POR:

SANDRO AUGUSTO PAVLIK HADDAD, Dr., FGA/UNB
(ORIENTADOR)



JOSÉ CAMARGO DA COSTA, Dr., ENE/UNB
(EXAMINADOR INTERNO)

FERNANDO RANGEL DE SOUSA, Dr., UFSC
(EXAMINADOR EXTERNO)

Brasília, 05 de agosto de 2014.

FICHA CATALOGRÁFICA

ANDRADE, DIOGO

Algoritmo de Síntese de Circuitos Analógicos Translineares Utilizando Decomposição Não-Paramétrica [Distrito Federal] 1999.

xvii, 157p., 210 x 297 mm (ENE/FT/UnB, Mestre, Dissertação de Mestrado – Universidade de Brasília. Faculdade de Tecnologia.

Departamento de Engenharia Elétrica

1.Síntese de circuitos analógicos	2.Translinear
3.Síntese automática	4.Decomposição não-paramétrica
I. ENE/FT/UnB	II. Título (série)

REFERÊNCIA BIBLIOGRÁFICA

ANDRADE, D. (2014). Algoritmo de Síntese de Circuitos Analógicos Translineares Utilizando Decomposição Não-Paramétrica. Dissertação de Mestrado em Engenharia Elétrica, Publicação PGEA.DM 574/14, Departamento de Engenharia Elétrica, Universidade de Brasília, Brasília, DF, 157p.

CESSÃO DE DIREITOS

AUTOR: Diogo Andrade.

TÍTULO: Algoritmo de Síntese de Circuitos Analógicos Translineares Utilizando Decomposição Não-Paramétrica.

GRAU: Mestre ANO: 2014

É concedida à Universidade de Brasília permissão para reproduzir cópias desta dissertação de mestrado e para emprestar ou vender tais cópias somente para propósitos acadêmicos e científicos. O autor reserva outros direitos de publicação e nenhuma parte dessa dissertação de mestrado pode ser reproduzida sem autorização por escrito do autor.

____

Diogo Andrade
SQS 216, BL I, AP 402.
70.295-090 Brasília – DF – Brasil.

RESUMO

ALGORITMO DE SÍNTESE DE CIRCUITOS ANALÓGICOS TRANSLINEARES UTILIZANDO DECOMPOSIÇÃO NÃO-PARAMÉTRICA

Autor: Diogo Andrade

Orientador: Sandro Augusto Pavlik Haddad

Programa de Pós-graduação em Engenharia de Sistemas Eletrônicos e Automação

Brasília, agosto de 2014

Neste trabalho, foi desenvolvido um algoritmo de síntese de circuitos translineares, útil para converter polinômios multivariável adimensionais, incluindo equações diferenciais lineares e não-lineares, em um ou mais polinômios translineares implementáveis em circuitos translineares. Circuitos translineares podem ser utilizados para realizar processamento de sinais inteiramente no domínio analógico e em modo-corrente, dispensando conversão analógica-digital, permitindo o desenvolvimento de circuitos de baixíssimo consumo de energia e de baixíssima tensão de alimentação.

Este algoritmo e outro encontrado na literatura são implementados em uma ferramenta computacional, e são comparados em termos de eficácia e eficiência. O algoritmo desenvolvido neste trabalho mostrou-se mais eficiente e mais eficaz que o outro em alguns casos. O algoritmo também foi validado ao ser aplicado em polinômios utilizados em circuitos translineares publicados, e as realizações utilizadas nestes trabalhos foram encontradas pelo algoritmo.

A ferramenta computacional foi implementada utilizando linguagem de programação "C", e não depende de pacotes de software proprietários. Seu código fonte foi disponibilizado gratuitamente sob a licença geral pública "GNU v.3"

ABSTRACT

ANALOG TRANSLINEAR CIRCUITS SYNTHESIS ALGORITHM USING NON-PARAMETRIC DECOMPOSITION

Author: Diogo Andrade

Supervisor: Sandro Augusto Pavlik Haddad

Programa de Pós-graduação em Engenharia de Sistemas Eletrônicos e Automação

Brasília, august of 2014

In this work, a Translinear circuit synthesis algorithm, useful to convert a generic dimensionless multivariate polynomial, including linear and non-linear differential equations, into one or more translinear polynomials that can be realized onto a Translinear Circuit was developed. Translinear circuits allow current-mode signal processing entirely into the analog domain, dispensing analog-to-digital conversion, resulting in ultra-low power and ultra low-voltage signal processing circuits.

This algorithm and another one found in the literature are implemented into a computer tool, and they are both compared regarding their efficiency and effectiveness. The developed algorithm was shown to be more effective and more efficient than the other in some cases. The algorithm was also validated by being applied to polynomials used in published Translinear circuits implementations, and the circuit realizations found in those works were also found by this algorithm.

The computer tool was implemented using "C" programming language, and it is not dependent on any proprietary software package. Its source code is freely available under the GNU General Public License v.3.

Dedicatória

Dedico este trabalho à Carmem, minha esposa, e a Raquel, minha filha.

Diogo Andrade

Agradecimentos

Muitas pessoas contribuíram, diretamente ou indiretamente, par a realização deste trabalho. Agradeço a cada uma delas! Neste momento, faz-se necessário reconhecer publicamente a participação específica de algumas.
Agradeço inicialmente ao professor Sandro Haddad, que além de ter sido um ótimo orientador, mostrou-se um grande amigo que sempre esteve me motivando nas horas mais difíceis, e que acreditou em mim quando nem eu acreditava mais. Tudo o que aprendi de microeletrônica desde que nos conhecemos, devo a ele.
A todos os colegas do LDCI, que formam uma família unida. São grandes amigos que me auxiliaram bastante neste trabalho, em especial a Heider Madureira e a José Edil Medeiros.
À minha filha Raquel, que me fez querer desistir tantas vezes deste trabalho para me dedicar exclusivamente ao cuidado dela, e à minha esposa Carmem por não ter deixado isso acontecer com o seu apoio e amor incondicionais.
À minha grande família: meus pais e irmãos, e aos pais e irmãos da Carmem, que também sempre estiveram muito disponíveis para me ajudar nos cuidados da Raquel durante as horas de trabalho mais intensos, e também por me darem apoio e incentivo incondicionais.
A Deus por abençoar minha jornada pela vida e colocar pessoas espetaculares em meu caminho.

Diogo Andrade

SUMÁRIO

1 Introdução ... 1

1.1 Contextualização ... 1
1.2 Definição do problema ... 2
1.3 Objetivos ... 3
1.4 Metodologia ... 3
1.5 Apresentação do manuscrito ... 4

2 Revisão Bibliográfica ... 5

2.1 Introdução ... 5
2.2 Metodologia de síntese de circuitos translineares utilizada ... 5
2.3 Princípio Translinear Estático ... 7
2.4 Decomposição Translinear não-paramétrica ... 10
2.4.1 Geração de Polinômios-Base ... 11
2.4.2 Transformação de Variáveis Estáticas ... 12
2.5 Decomposição Translinear Paramétrica ... 14
2.6 Princípio Translinear Dinâmico [38] ... 15
2.6.1 Transformação de Variáveis Dinâmicas ... 17
2.7 Exemplo de Síntese: Circuito RMS-DC [26] ... 19
2.7.1 Circuito da função Módulo [27] ... 21
2.8 Outras Metodologias de Síntese Translinear ... 23
2.8.1 Algoritmo de Grafos de David Ilsen [25] ... 24

3 Algoritmos de Decomposição Translinear não-paramétrica ... 25

3.1 Introdução ... 25
3.2 Análise Combinatória de um Algoritmo Trivial ... 25
3.3 Algoritmo de decomposição não-paramétrica original [23] ... 26
3.3.1 Geração do vetor de Polinômios-Base ... 28
3.3.2 Divisão recursiva por pares de polinômios-base, versão otimizada para minimizar tempo de execução ... 31
3.3.3 Verificação Final ... 36
3.3.4 Divisão recursiva por pares de polinômios-base, versão otimizada para minimizar a memória utilizada ... 38
3.4 Novo Algoritmo de decomposição não-paramétrica ... 39

3.4.1 Geração do Vetor de Polinômios-Base ... 42
3.4.2 Divisão Recursiva por Pares de Polinômios-Base e verificação final 48
3.5 Eliminação de polinômios translineares redundantes ... 52

4 Procedimento de validação e comparação entre algoritmos ... 55
4.1 Introdução ... 55
4.2 Eficácia do algoritmo implementado ... 55
4.3 Medida de Eficiência entre algoritmos ... 56
4.3.1 Geração de vetores de polinômios extremamente reduzidos e nova medida de eficiência ... 56
4.4 Polinômios utilizados para testar o algoritmo ... 57
4.4.1 Polinômios aleatórios ... 57
4.4.2 Realizações de circuitos Translineares publicados ... 58

5 Resultados ... 59
5.1 Introdução ... 59
5.2 Comparativo de esforço computacional entre algoritmos ... 59
5.3 Aplicação do algoritmo desenvolvido em trabalhos publicados ... 61
5.3.1 Oscilador de segunda ordem ... 61
5.3.2 Conversor RMS-DC ... 64
5.3.3 Circuito da função Módulo ... 65
5.3.4 Circuito de Seno ... 66

6 Conclusões ... 68
6.1 Proposta de Trabalhos Futuros ... 69

REFERÊNCIAS BIBLIOGRÁFICAS ... 70

Anexos ... 74

I Código fonte do aplicativo ... 75
I.1 arquivo “main.c’ ... 75
I.2 arquivo “main.h” ... 80

LISTA DE FIGURAS

1.1 Comparação entre consumo de potência por pólo entre filtros analógicos e digitais [15] 2

2.1 Fluxo do método de síntese de decomposição translinear. 6
2.2 Malha Translinear de 4 transistores.......... 7
2.3 Malha Translinear de 4 diodos com seguidores de tensão 9
2.4 Princípio Translinear Dinâmico [38] 15
2.5 Arranjos distintos de correntes de capacitor. [38] 16
2.6 Diagrama de blocos de um conversor RMS-DC [26] 19
2.7 Malha translinear dinâmica RMS-DC [26] 19
2.8 Circuito RMS-DC [26]. 20
2.9 Circuito retificador de onda completa. 22
2.10 Topologias de circuitos translineares contendo até seis transistores [25].......... 24

3.1 Dias necessários para execução do algoritmo trivial. 27
3.2 Estrutura de topo da implementação do algoritmo original de decomposição não-paramétrica [23].......... 28
3.3 Obtenção de Polinômios-Base (PB's) estritamente positivos (vetor β_1) 29
3.4 Divisão por um Polinômio-Base (PB) e fatoração do Resto (vetor β_2.......... 30
3.5 Divisão por pares de Polinômios-base e execução de decomposições parciais e Verificação final, versão otimizada para tempo de execução. 33
3.6 Sub-rotina de geração da lista de decomposições parciais $\alpha_{x,n}$.......... 35
3.7 Divisão por pares de Polinômios-base e execução de decomposições primárias, versão otimizada para minimizar a memória utilizada. 39
3.8 Sub-rotina de geração de decomposições primarias δ_x.......... 40
3.9 Sub-rotina de geração de decomposições secundarias δ_y. 41
3.10 Estrutura do novo algoritmo de decomposição não-paramétrica 42
3.11 Geração do Vetor de Polinômios-Base.......... 42
3.12 Algoritmo de eliminação de polinômios-base estritamente negativos (vetorτ_1). 43
3.13 Algoritmo de eliminação de polinômios-base (PB) redundantes(vetor τ_2). 45
3.14 Algoritmo de geração do vetor τ_3 de pares de polinômios-base (PB).......... 47
3.15 Algoritmo de geração do vetor τ_4 de polinômios-base (PB).......... 50
3.16 Divisão Recursiva por Pares de Polinômios Simultâneos. 53
3.17 Eliminação de polinômios translineares redundantes 54

4.1 Geração do vetor de polinômios-base extremamente reduzido τ_5 57

5.1 Circuito oscilador de segunda ordem [49] 62
5.2 Circuito Seno compacto 67
5.3 Saída do circuito Seno e seu valor ideal 67

LISTA DE TABELAS

2.1 Exemplo de geração de polinômios-base 12
2.2 Número de topologias de circuitos translineares para um dado número de transistores [25] 24

3.1 Exemplo de eliminação de Polinômis-Base estritamente negativos........ 44
3.2 Exemplo de vetor τ_2 47
3.3 Vetor τ_3 gerado a partir de vetor τ_2 reduzido 49
3.4 Vetor reduzido τ_3 51

4.1 Vetor de Polinômios de Entrada para avaliação dos algoritmos 58

5.1 Comparativo de eficiência computacional entre algoritmos com coeficientes $[-1, \ldots, +1]$. 60
5.2 Comparativo de eficiência computacional entre algoritmos com coeficientes $[-3, \ldots, +3]$ 61
5.3 Comparativo de eficiência computacional entre algoritmos com coeficientes $[-3, \ldots, +3]$ e vetores de polinômios-base extremamente reduzidos 62

LISTA DE SÍMBOLOS

Símbolos Latinos

GHz	Giga-Hertz	$[10^9 \mathrm{s}^{-1}]$
V_{BE}	Tensão Base-Emissor	[V]
V_{GS}	Tensão *Gate-Source*	[V]
V_{SB}	Tensão *Source-Bulk*	[V]
V_T	Tensão Térmica	[V]
I_S	Corrente de saturação	[A]
N_A	concentração de átomos receptores de elétrons	$[\mathrm{m}^{-3}]$
W	Largura de um transistor MOS	[m]
L	Comprimento do canal do transistor MOS	[m]
q	Carga do elétron	[C]

Símbolos Gregos

ϕ_F	Tensão de banda	[V]
ϵ_S	Permissividade dietética do semicondutor	$[\mathrm{Nm}^2/\mathrm{C}^2]$
μ	mobilidade de portadores de carga	$[\mathrm{V}^{-1}/\mathrm{s}]$
η_x	fator de escala do transistor "x"	adimensional
λ_x	combinação de vários fatores de escala de transistores	adimensional

Sobrescritos

$\cdot$	Variação temporal

Siglas

A/D	Analógico-Digital
BiCMOS	Bipolar-CMOS
BJT	*Bipolar Junction Transistor*
CMOS	*Complementary Metal-Oxide-Semiconductor*
DC	*Direct Current*
DTL	*Dynamic Translinear*
DR	*Dynamic Range*
ECG	Eletrocardiograma
EMG	Eletromiografia
FGMOS	*Floating Gate Metal-Oxide-Semiconductor*
FPAA	*Field Programmable Analog Array*
GB	*Giga Byte*
GNU	*GNU is Not Unix*
GPL	*General Public License*
LC	Indutor-Capacitor
LPF	*Low-Pass Filter*
LTK	Lei das Tensões de Kirchoff
MB	*Mega Byte*
MOS	*Metal-Oxide- Semiconductor*
PB	Polinômio-Base
PTAT	*Proportional To Absolute Temperature*
RAM	*Random Access Memory*
RFID	*Radio Frequency Identification*
RMS	*root mean square*
SAH	Sentido Anti-Horário
SH	Sentido Horário
SNR	*Signal to Noise Ratio*
STL	*Static Translinear*

Capítulo 1

Introdução

1.1 Contextualização

Um grande desafio no desenvolvimento de circuitos integrados para dispositivos embarcados é o consumo de potência. Estes dispositivos, de natureza portátil, são alimentados por baterias, ou por sistemas de coleta de energia do ambiente [1,2], ou por transmissão de potência sem fio [3–10], ou por uma combinação destas tecnologias. No caso de dispositivos embarcados implantados em seres vivos, um baixo consumo de potência talvez seja a restrição de projeto mais relevante, pois uma troca de bateria implica em um procedimento cirúrgico, o que traz riscos de infecções e custos elevados para a manutenção destes dispositivos.

A maioria dos dispositivos implantados realizam aquisição de sinais vitais, que pode ser acompanhada de processamento destes sinais no próprio dispositivo [11–14]. O paradigma de aquisição mais utilizado nestes dispositivos é o da amplificação e discretização do sinal analógico medido utilizando um conversor Analógico-Digital(A/D). Entretanto, o consumo de potência relacionado à amplificação e conversão A/D pode ser relativamente elevado, tornando necessária a utilização de tecnologias mais sofisticadas de fornecimento de potência.

Um paradigma alternativo de aquisição e processamento de sinais analógicos é o dos filtros translineares[1]. Esta classe de filtros é utilizada para fazer processamento de sinais inteiramente no domínio analógico e em modo-corrente. Em [15], foi realizado um estudo comparativo entre o consumo de potência utilizando filtros digitais e filtros analógicos translineares, levando-se em consideração o consumo de potência da conversão A/D. Neste estudo, considerou-se o consumo de potência de conversores A/D no estado da arte, bem como o de um conversor A/D ideal. Foi constatado que, para uma boa precisão no processamento de sinais (faixa dinâmica de 90 dB), a razão entre o consumo de potência em um filtro translinear e em um filtro digital com conversão A/D ideal pode chegar a 10^7, conforme visto na Fig. 1.1, onde a curva "A/D P_{min}" mostra o limite teórico mínimo de consumo de potência por pólo em um conversor A/D.

Um nível tão baixo de consumo de potência permite o desenvolvimento de dispositivos implan-

[1] também chamados *log-domain*

tados que possam funcionar com uma única bateria durante toda a expectativa de vida de um indivíduo. Assim, o processamento de sinais analógicos por meio de circuitos translineares tem uma aplicação muito relevante em sistemas embarcados e em sistemas biomédicos implantados. Combinando este paradigma de processamento de sinais com tecnologias de coleta de energia do ambiente, como em [1, 2], abrem-se inúmeras possibilidades de novos circuitos em que o uso de baterias ou de tecnologias de transmissão de potência sem fio sejam dispensáveis.

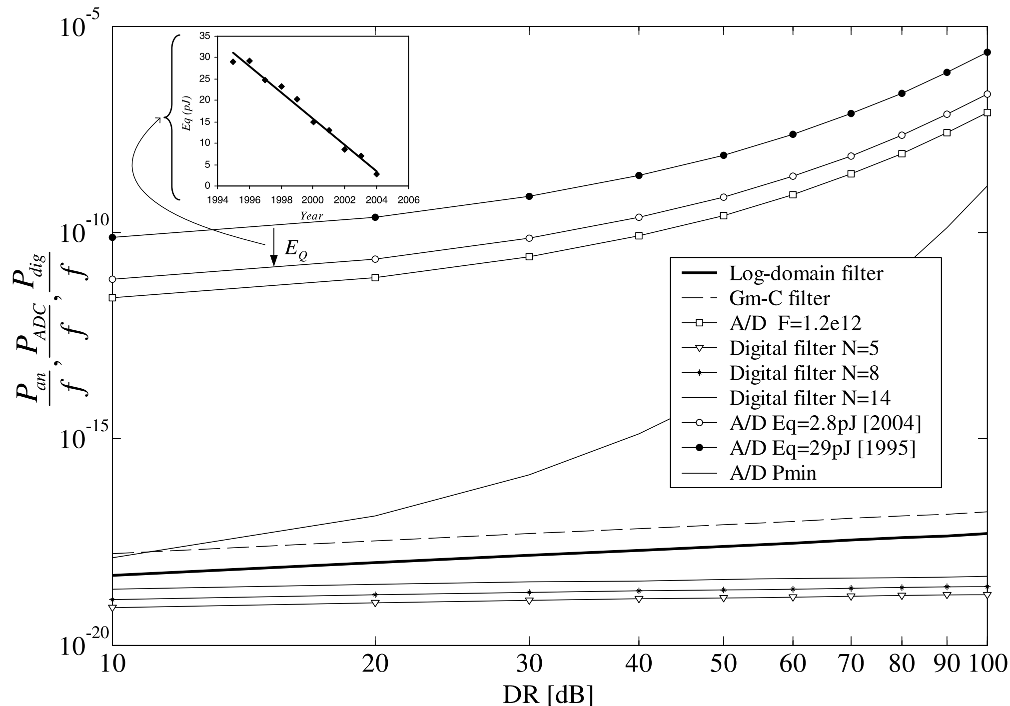


Figura 1.1: Comparação entre consumo de potência por pólo entre filtros analógicos e digitais [15]

Os circuitos translineares não tem sua aplicabilidade limitada à realização de filtros. Podem ser utilizados para realizar um processamento de sinal avançado, incluindo qualquer função matemática descrita por um polinômio, como funções estáticas[2] e equações diferenciais lineares e não-lineares. Como estes circuitos operam em modo corrente, a tensão de alimentação do circuito deixa de ser um fator limitante [16], possibilitando assim, circuitos processadores de sinais de baixíssima tensão e baixíssimo consumo de potência, como nos trabalhos [17–21].

1.2 Definição do problema

Partindo do fato de que circuitos translineares podem ser utilizados para construir circuitos de baixíssimo consumo de potência, há o problema de complexidade do processo de síntese destes circuitos. Como será visto no capítulo 2, existem várias metodologias na literatura, e as mais sistematizadas tendem a utilizar blocos fundamentais na construção dos circuitos, o que tende a torná-los super-dimensionados. Dentre as metodologias a serem apresentadas, a metodologia que

[2]instantâneas no tempo

utiliza o processo de “decomposição não-paramétrica” é a que produz os circuitos mais compactos, ou seja, com o mínimo de transistores e capacitores possível. Entretanto, este processo depende muito da “criatividade algébrica” do projetista para encontrar uma realização de circuito [22]. Assim, existe um *“trade-off”* entre a facilidade de sintetizar um circuito translinear e o tamanho do circuito gerado pelo método escolhido.

Em [23] um algoritmo de decomposição translinear não-paramétrica automatizada foi publicado, e representou a primeira metodologia de síntese automática de circuitos translineares desenvolvida, mas sua implementação não foi encontrada nas bases de dados pesquisadas. Em [24], o autor afirma que implementou este algoritmo, mas sua implementação também não foi encontrada. Já em [25], o autor propõe um algoritmo diferente, baseado em bancos de malhas translineares e classificação de padrões utilizando teoria dos grafos, mas não utiliza o processo de decomposição não-paramétrica, e utiliza pacotes de software de alto custo em sua implementação. Assim, não se encontra ainda uma implementação auto-contida[3] de um processo de decomposição paramétrica automatizado disponibilizada publicamente, e tampouco alguma análise e proposta de melhoria em relação ao algoritmo apresentado em [23].

Define-se assim o problema a ser resolvido neste trabalho como ausência de uma ferramenta computacional gratuita e auto-contida que implemente um algoritmo de decomposição translinear não-paramétrica automatizada. Este é um problema que se encontra na fronteira entre a teoria de circuitos translineares, as teorias de álgebra combinatória, e ciências da computação.

1.3 Objetivos

Este trabalho tem como objetivos analisar e implementar o algoritmo de decomposição translinear não-paramétrica descrito em [23], e apresentar mudanças a este algoritmo que o tornam mais eficaz, com a finalidade de produzir mais resultados, e mais eficiente, com a finalidade de reduzir o tempo de computação necessário para encontrar todas os resultados possíveis. Esta implementação deverá ser auto-contida, ou seja, não dependerá do uso de pacotes de software proprietários.

Outro objetivo é publicar o código-fonte da Implementação resultante, tornando-a software livre, como uma forma de se estimular o desenvolvimento de circuitos translineares.

1.4 Metodologia

Este trabalho iniciou-se com o estudo dos princípios translineares estático e dinâmico, e em seguida foi realizada uma pesquisa a respeito de metodologias de síntese destes circuitos. Os documentos pesquisados foram: livros-texto, artigos e periódicos científicos relacionados a circuitos e sistemas. Sítios eletrônicos na internet foram utilizados de forma complementar.

Escolheu-se o algoritmo de decomposição não-paramétrica de [23], por ser uma metodologia de síntese automática, para realizar um estudo mais profundo, e assim foi formulado um novo

[3]não depende de pacotes de software proprietários

algoritmo. Para validar e comparar os dois algoritmos, ambos foram implementados em linguagem de programação "C". Esta linguagem foi escolhida porque o autor do algoritmo de [23] afirma que implementou o seu algoritmo nesta linguagem, e decidiu-se seguir a mesma metodologia.

Definiu-se, então uma metodologia de validação e comparação dos algoritmos. Executou-se esta metodologia e os resultados foram colhidos e analisados.

1.5 Apresentação do manuscrito

No capítulo 2 é feita uma revisão bibliográfica sobre princípio translinear, e a descrição da metodologia de síntese de circuitos translineares que utiliza o processo de decomposição translinear. No 3, três algoritmos de decomposição translinear não paramétrica são mostrados: Um algoritmo trivial, o algoritmo original de [23], e o novo algoritmo desenvolvido neste trabalho. A metodologia para se comparar os algoritmos é descrita no capítulo 4, e os resultados da comparação e validação são descritos no capítulo 5. No capítulo 6, são apresentadas as conclusões e são propostos trabalhos futuros a respeito do tema de decomposição translinear não-paramétrica automatizada. O Anexo contém o código-fonte da ferramenta que implementa tanto o novo algoritmo quanto o algoritmo original.

Capítulo 2

Revisão Bibliográfica

2.1 Introdução

Neste capítulo, é apresentada a metodologia de síntese de circuitos translineares utilizada neste trabalho: decomposição translinear. Em seguida são apresentados os conceitos necessários para a aplicação do método: o princípio transliner estático, o conceito de decomposição translinear paramétrica e não-paramétrica, e o princípio translinear dinâmico. Como exemplo, a síntese de um circuito RMS-DC [26] e de um circuito da função módulo [27] são mostradas. Em seguida, um algoritmo de decomposição não-paramétrica trivial é mostrado como justificativa para que se desenvolva algoritmos mais eficientes. O algoritmo de decomposição não-paramétrica de [23], que é a referência para o algoritmo desenvolvido neste trabalho, é apresentado, e finalmente, outras metodologias de síntese são brevemente apresentadas.

A metodologia de síntese de decomposição translinear utilizada neste trabalho é uma extensão das apresentadas em [23,27]. Nestes trabalhos a parte de implementação em hardware dos circuitos e métodos de representação em espaço de estados de filtros transllineares são abordados com detalhes, enquanto que neste trabalho, apenas conceitos básicos de transformação de equações diferenciais são apresentados, e os detalhes a respeito de implementação em hardware não são apresentados. A pesquisa realizada não encontrou publicações recentes a respeito de detalhes de implementação de hardware. Entretanto, as metodologias de representação de filtros translineares são bem detalhadas em [15].

A metodologia de análise de circuitos Translineares fica implícita dentro do método de síntese apresentado. Metodologias de análise detalhadas podem ser encontradas em [23, 27, 28].

2.2 Metodologia de síntese de circuitos translineares utilizada

O método de decomposição translinear consiste em encontrar uma topologia de circuito analógico que realize um processamento de sinal contínuo no tempo dado por uma equação algébrica adimensional, que por sua vez pode ser uma equação diferencial linear ou não-linear. O fluxo deste

método pode ser visto na Fig. 2.1:

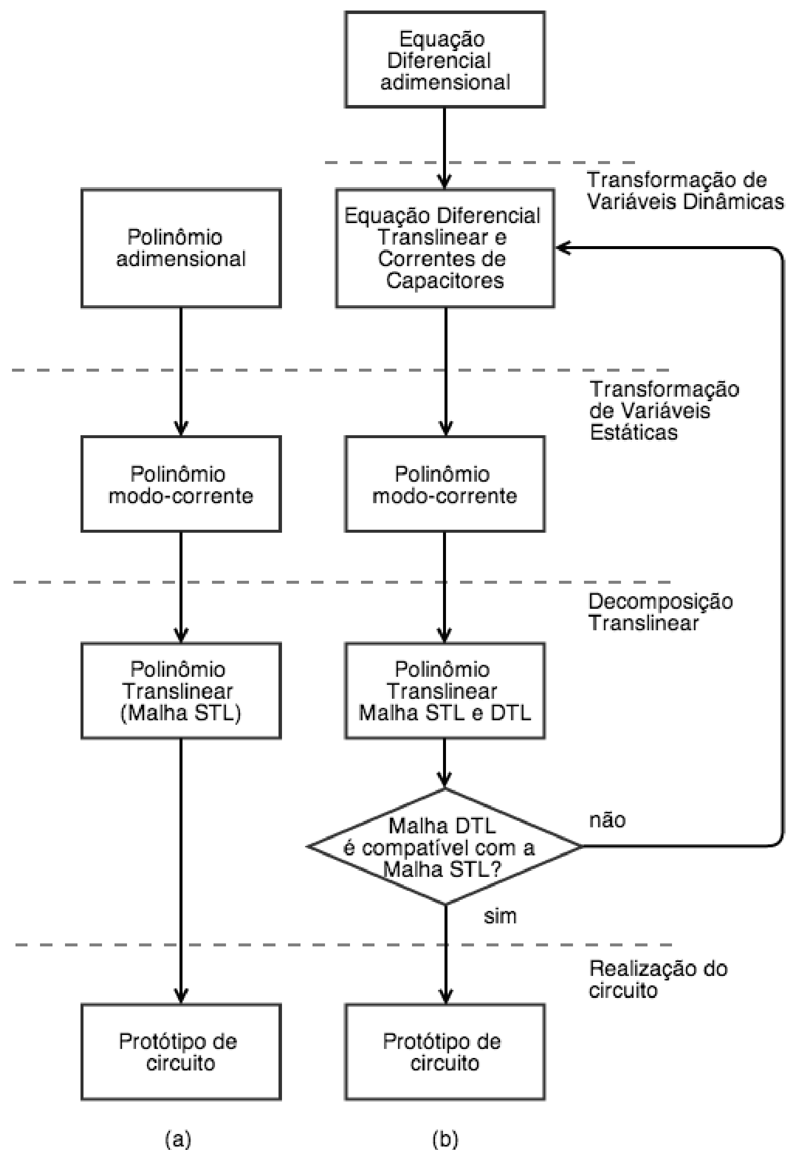


Figura 2.1: Fluxo do método de síntese de decomposição translinear: (a): circuitos estáticos, (b) circuitos dinâmicos, adaptado de [23]

Um ponto importante da metodologia apresentada é que, após ter sido obtido um polinômio modo-corrente, a trajetória da síntese é igual para polinômios estáticos e para equações diferenciais. Uma diferença significativa é que, no método de síntese de circuitos dinâmicos, pode ser que as correntes de capacitores definidas não sejam compatíveis com os polinômios translineares encontrados, fazendo com que o projetista tenha que voltar ao passo inicial.

Exceto pela etapa de realização do circuito, que está fora do escopo deste trabalho, todas as outras são descritas nas seções a seguir. A etapa de decomposição translinear pode ser realizada por decomposição paramétrica ou não-paramétrica. O algoritmo de decomposição não paramétrica de [23] é descrito com detalhes na seção 3.3, já da forma que foi implementada na ferramenta

desenvolvida neste trabalho, pois o autor especificou apenas o algoritmo abstrato, sem qualquer estratégia de implementação.

Este método é divergente, pois várias escolhas diferentes podem ser feitas no caminho, e múltiplas soluções para se implementar o mesmo polinômio de entrada podem ser alcançadas, e elas vão ter desempenho diferente quanto a: razão sinal ruído (SNR), faixa dinâmica (DR), tensão de alimentação mínima, consumo de energia, largura de banda, sensibilidade a parâmetros de componentes, distorção harmônica, etc. Este tipo de análise está fora do escopo deste trabalho, ou seja, a busca por uma realização de circuito translinear de *alto desempenho* deve ser feita de forma subjetiva dentre as decomposições não-paramétricas encontradas pela ferramenta.

2.3 Princípio Translinear Estático

O princípio Translinear Estático (STL —*Static Translinear*) foi proposto por Gilbert em 1975 [29]. Considerando um transistor bipolar de junção (BJT) como o dispositivo de referência para esta definição, este princípio pode ser enunciado da seguinte forma:

Princípio STL: *Dada uma malha que passa pelas junções base-emissor de um conjunto de transistores com a mesma tensão térmica V_T, e a mesma corrente de saturação I_S, sendo que o número de transistores com a junção base-emissor no sentido horário (SH) da malha é igual ao número de transistores com suas junções no sentido anti-horário (SAH), o produto das correntes de coletor dos transistores com suas junções base-emissor no sentido horário é igual ao produto das correntes dos transistores com suas junções base-emissor no sentido anti-horário. A esta malha dá-se o nome de "malha translinear".*[1]

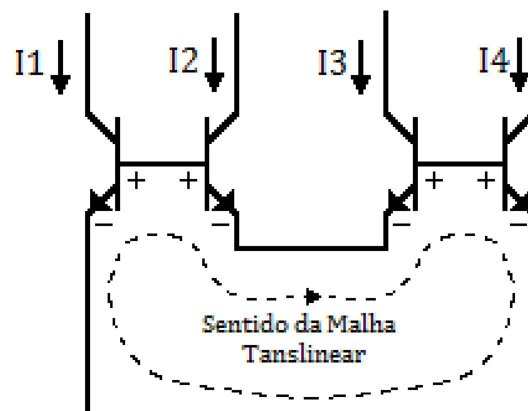


Figura 2.2: Malha Translinear de 4 transistores

A Fig. 2.2 mostra uma malha translinear de quatro transistores. Aplicando a lei das tensões de Kirchoff (LTK) à esta malha translinear, a relação entre as tensões base-emissor é dada pela

[1] Esta é a versão do autor baseada em outras formulações do princípio STL encontradas na literatura [27, 30–32]

Eq. (2.1)

$$-V_{BE1}+V_{BE2}-V_{BE3}+V_{BE4}=0 \tag{2.1}$$

Considerando a equação do BJT ideal dado pela Eq. (2.2), onde η_x é o fator de escala do BJT, V_T é a tensão térmica, I_S a corrente de saturação do transistor, e V_{BEx} é a tensão base-emissor do transistor por onde flui a corrente de coletor I_x, pode-se Isolar V_{BEx} na Eq. (2.2) e obtém-se a sua expressão dada pela Eq. (2.3). Substituindo-o na Eq. (2.1), obtém-se uma nova expressão, dada pela Eq. (2.4)

$$I_x=\eta_x I_s e^{\frac{V_{BEx}}{V_T}} \tag{2.2}$$

$$V_{BEx}=V_T\ln\left(\frac{I_x}{\eta_x I_S}\right) \tag{2.3}$$

$$V_T\ln\left(\frac{I_1}{\eta_1 I_S}\right)-V_T\ln\left(\frac{I_2}{\eta_2 I_S}\right)+V_T\ln\left(\frac{I_3}{\eta_3 I_S}\right)-V_T\ln\left(\frac{I_4}{\eta_4 I_S}\right)=0 \tag{2.4}$$

Assumindo que os transistores operam à mesma temperatura e que são iguais[2], pode-se eliminar I_S e V_T da Eq. (2.4), o que possibilita eliminar o logaritmo da equação, e o resultado é mostrado na Eq. (**??**).

$$\begin{aligned}
V_T\ln\left(\frac{I_1}{\eta_1 I_S}\right)+V_T\ln\left(\frac{I_3}{\eta_3 I_S}\right)&=V_T\ln\left(\frac{I_2}{\eta_2 I_S}\right)+V_T\ln\left(\frac{I_4}{\eta_4 I_S}\right)\\
\ln\left(\frac{I_1 I_3}{\eta_1\eta_3}\right)&=\ln\left(\frac{I_2 I_4}{\eta_2\eta_4}\right)\\
\frac{I_1 I_3}{\eta_1\eta_3}&=\frac{I_2 I_4}{\eta_2\eta_4}\\
\lambda_{SAH}I_1I_3-\lambda_{SH}I_2I_4&=0
\end{aligned} \tag{2.5}$$

Assim, uma malha translinear em um circuito implementa uma diferença de produtos de correntes, e esta relação pode ser generalizada na Eq. (2.6). Onde os fatores de escala η_x de cada produto são combinados nas constantes λ_{SAH} e λ_{SH}, referentes aos transistores conectados no sentido anti-horário e sentido horário respectivamente. Uma característica relevante do circuito translinear estático, é que o seu comportamento dado pela Eq. (2.5) é invariante com a temperatura, pois os termos V_T são iguais para todos os transistores.[3].

$$\lambda_{SAH}\prod_{j\in \text{SAH}} I_j-\lambda_{SH}\prod_{j\in \text{SH}} I_j=0 \tag{2.6}$$

O princípio é válido não apenas para o BJT, mas também para qualquer dispositivo que apresente uma relação $I-V$ exponencial, como o transistor MOS em inversão fraca [33], diodos com seguidores de tensão [34, 35], BJT lateral do transistor MOS [36], e transistores MOS com *gate* flutuante (FGMOS) também em inversão fraca [37]. Como exemplo, mostra-se o transistor MOS e os diodos com seguidores de tensão.

[2] Assume-se que os dispositivos possuem o mesmo modelo de grandes sinais e são bem-casados.

[3] Para maior clareza no desenvolvimento deste trabalho, as correntes $I_j, j\in$ SH serão denominadas "correntes conectadas no SH", enquanto que as correntes $I_j, j\in$ SAH serão denominadas "correntes conectadas no SAH". Um par de correntes em que uma é conectada no sentido SH e a outra no sentido SAH serão chamadas de "correntes opostamente conectadas".

O transistor MOS em inversão fraca tem a corrente de dreno I_{DS} dada pela Eq. (2.7), e sua corrente de saturação, dada pela Eq.(2.8), depende apenas da tensão entre a fonte e o corpo V_{SB}, sendo os outros parâmetros constantes dadas pelo processo de fabricação [33].

$$I_{DS} = \frac{W}{L} I_M(V_{SB}) e^{\frac{V_{GS}}{nV_T}} (1 - e^{\frac{-V_{DS}}{nV_T}}) \tag{2.7}$$

$$I_M(V_{SB}) = \mu V_T^2 e^{\frac{-V_M}{nV_T}} \frac{\sqrt{2q\epsilon_S N_A}}{2\sqrt{2\phi_F + V_{SB}}} \tag{2.8}$$

Se V_{SB} é igual para todos os transistores, então $I_M(V_{SB}) = I'_M$ é uma constante, se V_{GS} é menor que a tensão limiar de entrada na região de inversão moderada V_M, e se $V_{DS} \geq 3V_T$, o que faz o termo entre parênteses da Eq. (2.7) seja aproximadamente 1, então a Eq. (2.7) pode ser rescrita na Eq. (2.9), que tem o mesmo formato da Eq. (2.2), que modela o BJT ideal:

$$I_{DS} = \frac{W}{L} I'_M e^{\frac{V_{GS}}{nV_T}} \tag{2.9}$$

Assim, tomando os devidos cuidados, os transistores MOS podem ser utilizados em realizações de circuitos translineares. Entretanto, para manter o transistor MOS em inversão fraca para correntes mais altas, da ordem de μA, as dimensões dos transistores ficam muito grandes, o que limita severamente a operabilidade em altas frequências devido às capacitâncias parasitas tomarem proporções significativas. Entretanto, há uma vantagem em se utilizar transistores MOS: tecnologias que implementam CMOS exclusivamente são baratas, bem como as que implementam exclusivamente BJT, mas a tecnologia BiCMOS, que implementam ambos, são mais caras. Como o uso de CMOS para circuitos digitais é mais econômico que o uso da tecnologia de BJT em termos de consumo de energia, a tecnologia CMOS é mais adequada para circuitos de sinais mistos.

Uma alternativa para operar em altas frequências com correntes mais altas, é utilizar diodos passivos como dispositivo translinear e transistores MOS para construir amplificadores operacionais configurados como seguidores de tensão. Assim, pode-se ter tensões iguais nos terminais p dos diodos por meio do curto-circuito virtual, enquanto correntes distintas fluem em cada diodo. A Figura 2.3 mostra um exemplo deste arranjo.

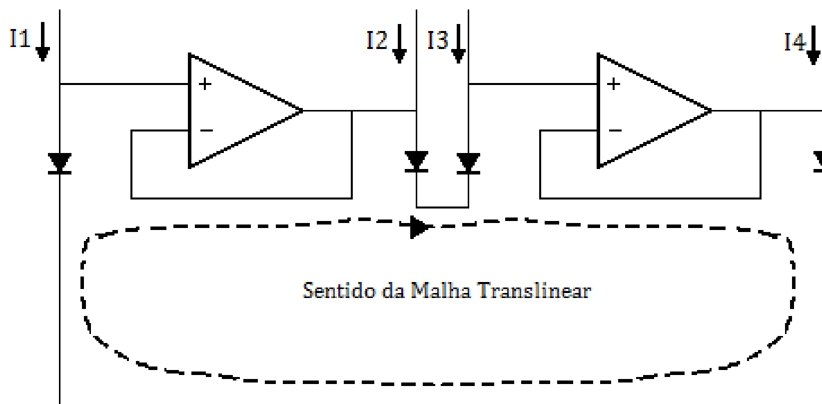


Figura 2.3: Malha Translinear de 4 diodos com seguidores de tensão

Para uma correta operação de um circuito translinear, os dispositivos devem ser devidamente polarizados de modo que o comportamento exponencial seja mantido, e as devidas correntes devem

ser forçadas nos dispositivos apropriados para que as correntes resultantes desejadas sejam geradas. Um exemplo de polarização de um circuito translinear será mostrado mais adiante no Capítulo 5

Este princípio translinear é chamado *estático* porque não são levados em consideração comportamentos transientes do circuito em sua formulação. Neste princípio, apenas o comportamento estático, ou seja o comportamento DC do circuito, é modelado.

O estudo a respeito de efeitos de segunda-ordem como distorção, ruído, etc. estão fora do escopo deste trabalho, pois estes efeitos são específicos da tecnologia utilizada para implementar o circuito desejado.

2.4 Decomposição Translinear não-paramétrica

A forma da Eq. (2.6) permite utilizar circuitos translineares para processamento de sinais modo-corrente no domínio analógico. Como as correntes de coletor I_j podem ser forçadas, elas podem ser formadas por combinações lineares de diferentes correntes. Como exemplo, considerando o circuito da Fig. 2.2, se $I_1 = I_3 = I_a$, $I_2 = I_y + I_b$, $I_4 = I_y - I_b$, e $\lambda_{SH} = \lambda_{SAH} = 1$, tem-se um circuito translinear dado pela Eq. (2.10).

$$(I_a)(I_a) - (I_y + I_b)(I_y - I_b) = 0 \tag{2.10}$$

Desenvolvendo a Eq. (2.10), encontra-se a expressão da média geométrica, dada pela Eq. (2.11):

$$\begin{aligned} &I_a^2 - (I_y^2 + I_b^2) = 0 \\ &I_y^2 = I_b^2 + I_a^2 \\ &I_y = \sqrt{I_b^2 + I_a^2} \end{aligned} \tag{2.11}$$

Assim, pode-se implementar, com um circuito simples, o cálculo da média geométrica entre dois sinais sem utilizar conversão A/D e sem ter que utilizar um microprocessador. Esta é a grande vantagem de se utilizar circuitos translineares para processamento de sinais: *Realizar um processamento de sinal avançado no domínio analógico, com baixas tensões e baixas correntes, e consequentemente, com baixo consumo de energia.*

No entanto, encontrar uma realização translinear para um dado polinômio de entrada não é uma tarefa trivial. Ainda que encontrar a Eq. (2.10) a partir da Eq. (2.11) possa não parecer tão difícil, encontrar uma realização de um polinômio mais complexo, com mutilas variáveis e de grau elevado exigiria um certo nível de “criatividade algébrica”. Nas palavras de Frey e Drakkis[4]: *“O princípio translinear estático sugere uma implementação de circuito se um único produto de correntes do lado esquerdo* [da equação] *for igual a outro único produto de correntes no lado direito* [da equação]. *Isto pode ser realizado com um algumas operações de rearranjo e fatoração baseados na etapa de síntese chamada de ‘decomposição translinear’. Encontrar o rearranjo apropriado é limitado pelo nível de criatividade e experiência que dão o devido valor a este método. Entretanto, alguns podem considerar este processo como um desafio..”* [22]. Assim, define-se o processo de

[4]tradução do autor

decomposição translinear não-paramétrica como a reescrita de um dado polinômio modo-corrente em uma diferença de produtos, na forma da Eq. (2.6). É chamada não-paramétrica pois procura rescrever um polinômio de entrada em uma única malha translinear, sem introduzir novos parâmetros. O processo de decomposição paramétrica é descrito na seção 2.5.

O processo de decomposição translinear não-paramétrica é puramente algébrico. O objetivo é tentar combinar, de alguma forma, *polinômios-base* de forma a obter uma representação do polinômio de entrada na forma da Eq. (2.6). A essa representação dá-se o nome de "polinômio translinear". Como é simples, por meio de espelhos de correntes, somar e subtrair as correntes que serão forçadas no coletor de um transistor qualquer da malha translinear, esta corrente de coletor é representada no domínio algébrico por um polinômio-base, que é uma combinação linear das variáveis que compõe o polinômio de entrada E_r, r sendo o grau do polinômio. Chamando os polinômio-base de P_x, uma polinômio translinear pode ser escrito na forma da Eq. (2.12). Pode-se notar que, devido ao número de termos de cada produto da Eq. (2.12) conter um número de polinômios lineares igual ao grau do polinômio de entrada E_r, este deverá ser *homogêneo*, ou seja, todos so monômios resultantes de sua expansão deverão ser de grau r. Caso E_r não seja homogêneo, uma transformação de variáveis adequada deverá ser feita, conforme será mostrado nas seções 2.4.2 e 2.6.1[5].

$$\lambda_1 P_1 P_3 \ldots P_{2r-1} - \lambda_2 P_2 P_4 \ldots P_{2r} \tag{2.12}$$

2.4.1 Geração de Polinômios-Base

Conforme visto na seção anterior, um polinômio-base é uma combinação linear das variáveis que compõe o polinômio de entrada E_r, sendo r o grau do polinômio. Para se desenvolver um algoritmo de decomposição não-paramétrica qualquer, é necessário primeiramente gerar um vetor de polinômios-base, a partir dos quais o algoritmo tentará encontrar os possíveis polinômios translineares. Para gerar este vetor, deve-se escolher um intervalo de coeficientes que serão combinados com as variáveis. O número de polinômios-base é função do número de variáveis v e do intervalo de coeficientes $[-N, \ldots, +N]$ utilizados. Como o número zero faz parte do intervalo, o número de coeficientes é dado por $(2N + 1)$. Considerando que cada variável deve ser multiplicada por algum coeficiente numérico arbitrário, o numero total de polinômios-base N_{β_0} é uma permutação com repetição, de $(2N + 1)$ tomados v a v, dado ela Eq (2.13). Por exemplo, considerando um polinômio de duas variáveis (chamadas aqui de x e y), e coeficientes inteiros entre $[-1, \ldots, +1]$, ou seja, $N = 1$, independentemente do grau deste polinômio, os seus polinômios-base são mostrados na Tabela 2.1.

$$N_{\beta_0} = (2N + 1)^v \tag{2.13}$$

[5]Para tornar o desenvolvimento nas seções seguintes mais claro, o produto $\lambda_1 P_1 P_3 \ldots P_{2r-1}$, por ter os índices ímpares, será denominado "lado ímpar", e os polinômios-base P_1, P_3, $\ldots$, P_{2r-1} serão denominados "polinômios ímpares". De forma análoga, o produto $\lambda_2 P_2 P_4 \ldots P_{2r}$, será denominado "lado par", e os polinômios-base P_2, P_4, $\ldots$, P_{2r} serão denominados "polinômios pares", sendo que o lado ímpar e o lado par serão chamados de "lados opostos". O termo "polinômios opostos" se refere a um par de polinômios em que um é um polinômio ímpar e o outro é um polinômio par. Por exemplo, P_1 e P_2 pertencentes à Eq. (2.12) são polinômios opostos.

Tabela 2.1: Exemplo de geração de polinômios-base

x	y	PB
-1	-1	$(-x-y)$
-1	0	$(-x)$
-1	1	$(-x+y)$
0	-1	$(-y)$
0	0	0
0	1	(y)
1	-1	$(x-y)$
1	0	(x)
1	1	$(x+y)$

Colocando os valores de $v = 2$ e $N = 1$ na fórmula da Eq. (2.13), $N_{\beta_0} = (2 * 1 + 1)^2 = 9$, exatamente o número de polinômios-base da Tabela 2.1. A este conjunto completo de polinômios-base dá-se o nome de β_0, e será referenciado nos algoritmos de decomposição translinear não-paramétrica descritos neste trabalho.

Os coeficientes das variáveis no domínio algébrico representam espelhos de corrente na implementação do circuito, então apenas coeficientes inteiros ou fracionários podem ser utilizados, para que se possa obter cópias precisas das correntes. É importante ressaltar que, sempre existirá um polinômio translinear de E_r, bastando que, para isso, possa-se utilizar todos os números fracionários, ou seja, todos os números do conjunto-universo $\mathbb{Q}$ como possíveis coeficientes das variáveis nos polinômios-base. No entanto, se este intervalo fosse utilizado, o número de combinações possíveis, até mesmo para um algoritmo eficiente, seria infinito, logo a escolha destes coeficientes não seria prática. Assim, escolhe-se apenas coeficientes inteiros, e geralmente com $N < 10$. Sendo os coeficientes limitados, as soluções são finitas, logo é possível implementar um algoritmo para encontrá-las.

2.4.2 Transformação de Variáveis Estáticas

Para que um polinômio de grau r seja realizável em uma malha translinear conforme mostrado na seção 2.4, ele deve ser *modo-corrente*, ou seja, todas as suas variáveis devem descrever correntes elétricas em um circuito. Se deseja-se obter um polinômio translinear a partir de um polinômio adimensional, primeiro é necessário transformar suas variáveis adimensionais, que são função de uma variável implícita adimensional τ em variáveis de correntes no tempo. Entretanto, isso deve ser feito sem alterar o "comportamento" do polinômio, ou seja, as transformações devem manter a admiensionalidade da variável a ser transformada. Para isso, ao transformar uma variável adimensional em uma variável de corrente, introduzem-se novas correntes de referência do tipo $I_{0x}(t)$:

$$x(\tau) = \frac{I_x(t)}{I_{0x}(t)} \tag{2.14}$$

Assim, na transformação da variável $x(\tau)$ em $I_x(t)$, os dois lados da Eq. (2.14) são mantidos adimensionais, mas agora é a corrente $I_x(t)$ que carrega a informação da variável $x(\tau)$, e a variável introduzida $I_{0x}(t)$ vai definir, de alguma forma os limites da excursão de sinal de $I_x(t)$. Como exemplo, faz-se a transformação de variáveis do polinômio que representa a aproximação da função $\text{sen}(\pi x)$ [29], omitindo (τ) e (t) para melhor legibilidade:

$$\frac{\text{sen}(\pi x)}{\pi} \approx \frac{x - x^3}{1 + x^2} \tag{2.15}$$

Igualando a variável y ao lado direito da aproximação dada pelos Eq. (2.15), obtém-se:

$$y = \frac{x - x^3}{1 + x^2} \tag{2.16}$$

Desenvolvendo-se a Eq. (2.16) obtêm-se o polinômio adimensional dado pelo lado esquerdo da igualdade da Eq. (2.17)

$$x^3 + x^2 y - x + y = 0 \tag{2.17}$$

Fazendo as transformações das variáveis y e x para correntes I_{out} e I_{in} respectivamente, e substituindo no polinômio da Eq. (2.17) obtém-se:

$$y = \frac{I_{out}}{I_0} \tag{2.18}$$

$$x = \frac{I_{in}}{I_0} \tag{2.19}$$

$$\frac{{I_{in}}^3}{{I_0}^3} + \frac{{I_{in}}^2 I_{out}}{{I_0}^3} - \frac{I_{in}}{I_0} + \frac{I_{out}}{I_0} = 0 \tag{2.20}$$

Removendo os denominadores obtém-se:

$${I_{in}}^3 + {I_{in}}^2 I_{out} - {I_0}^2 I_{in} + {I_0}^2 I_{out} = 0 \tag{2.21}$$

O lado esquerdo da igualdade na Eq. (2.21) é um polinômio modo-corrente de grau 3 equivalente ao polinômio adimensional do lado esquerdo da igualdade na Eq. (2.17). Nota-se que o polinômio da Eq. (2.21) é homogêneo, ou seja, todos os seus monômios possuem o mesmo grau. Outro aspecto importante é que cada variável pode ser transformada por uma corrente de referência diferente, ou seja, não é necessário que todas as variáveis sejam transformadas por meio da mesma corrente como foi feito no exemplo acima. Por exemplo, ao escolher as transformações:

$$y = \frac{I_{out}}{I_2} \tag{2.22}$$

$$x = \frac{I_{in}}{I_1} \tag{2.23}$$

O polinômio modo-corrente resultante possui um grau a mais:

$$I_2 {I_{in}}^3 + I_1 {I_{in}}^2 I_{out} - {I_1}^2 I_2 I_{in} + {I_1}^3 I_{out} = 0 \tag{2.24}$$

Assim, existem várias formas de se transformar um polinômio adimensional em um polinômio modo-corrente homogêneo. Por ser um método direto e discricionário, não é necessário um algoritmo para esta etapa.

2.5 Decomposição Translinear Paramétrica

Nem sempre é possível encontrar um polinômio translinear dentro do intervalo de coeficientes inteiros especificado. Entretanto, é possível dividir o polinômio de entrada E_r em sub-polinômios menores utilizando variáveis intermediárias, ou seja, *parâmetros*. Em [27], o método de decomposição paramétrica descrito nesta seção é proposto. Dado o polinômio de entrada na forma $P(I_1, I_2, \ldots, I_n)$, sendo n o número de variáveis chamadas I_n, e I_p uma variável intermediária, pode-se efetuar a transformação das Eqs. (2.25)

$$P(I_1, I_2, \ldots, I_n) = P(I_1, I_2, \ldots, I_n, I_p) \quad (2.25a)$$

$$P(I_1, I_2, \ldots, I_n, I_p) = 0 \quad (2.25b)$$

Cada uma das das Eqs. (2.25) formará uma malha translinear distinta. Deste modo, a variável intermediária I_p não poderá ser uma combinação linear das variáveis I_n, ou a Eq. (2.25b) não formará um polinômio homogêneo. Para ilustrar melhor o método e esta afirmação, suponha que deseja-se realizar decomposição paramétrica no polinômio homogêneo E_3 dado pela Eq. (**??**):

$$E_3 = a^2b + ab^2 + ya^2 + yab + y^3 \quad (2.26)$$

$$a^2b + ab^2 + ya^2 + yab + y^3 = 0 \quad (2.27)$$

Seguindo a forma das Eqs. (2.25), divide-se este polinômio em dois usando a variável I_p, conforme ilustrado na Eq. (2.28):

$$a^2b + ab^2 + ya^2 + yab + y^3 = a^2b + ab^2 + ya^2 + I_pab \quad (2.28a)$$

$$a^2b + ab^2 + ya^2 + I_pab = 0 \quad (2.28b)$$

Simplificando a Eq.(2.28a) e a Eq. (2.29a), obtém-se um novo conjunto de polinômios dados pela Eq. (2.29):

$$yab + y^3 - I_pab = 0 \quad (2.29a)$$

$$ab + b^2 + ya + I_pb = 0 \quad (2.29b)$$

Estes polinômios são mais simples que o da Eq. (2.26), e se for conveniente, pode-se reduzir qualquer um deles ainda mais repetindo o processo, introduzindo outras variáveis intermediárias. Isolando I_p na Eq. (2.29b), obtém-se:

$$I_p = a + b + \frac{ya}{b} \quad (2.30)$$

A Eq. (2.30) mostra que não há como a variável I_p ser uma combinação linear das variáveis $[a, b, y]$ do polinômio E_3, caso contrário o polinômio dado pela Eq. (2.28b) não seria homogêneo. A variável intermediária deve sempre formar um monômio do mesmo grau do polinômio de entrada.

Em [23], o autor afirma que o algoritmo de decomposição não-paramétrica formulado por ele (a ser visto na seção 3.3.1.1) não pode ser utilizado para encontrar os polinômios translineares dos polinômios modo-corrente gerados por este processo de decomposição paramétrica. Já o algoritmo de decomposição não-paramétrica desenvolvido neste trabalho pode ser utilizado no auxílio deste

processo de decomposição paramétrica (a ser visto na seção 3.4.1.2). Aplicando este algoritmo ao conjunto da Eq. (2.29) obtém-se o conjunto de decomposições não-paramétricas dado pela Eq. (2.31)

$$y^3 - ab(I_p - y) = 0 \tag{2.31a}$$

$$(a + b)(I_p + b) - a(I_p - y) = 0 \tag{2.31b}$$

De acordo com a pesquisa realizada, não foi encontrado algoritmo de decomposição paramétrica automatizado algum. Provavelmente porque há infinitas formas de se adicionar variáveis intermediárias [25].

2.6 Princípio Translinear Dinâmico [38]

O princípio STL permite transformar equações algébricas estáticas em circuitos translineares, e o princípio DTL (*Dynamic Trasnlinear*) permite transformar equações diferenciais, lineares ou não, também em circuitos translineares. Para realizar esta transformação, basta associar um capacitor a um dispositivo de comportamento exponencial, dado pela Eq. (2.2), como visto na seção 2.3. A Fig. 2.4 mostra como é feita a associação: Sendo I_{cap} a corrente que flui dentro do

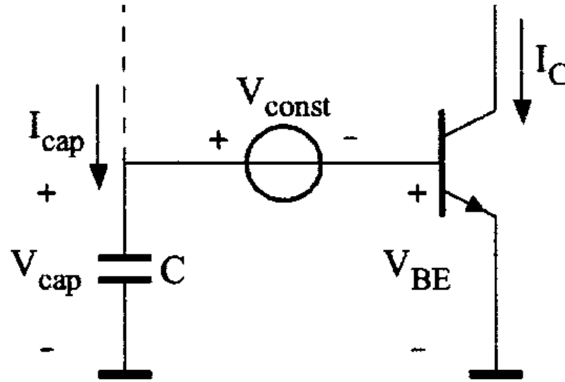


Figura 2.4: Princípio Translinear Dinâmico [38]

capacitor C, I_C a corrente de coletor do transistor, e V_{BE} a tensão base-emissor do transistor. V_{const} representa qualquer dispositivo que tenha uma tensão constante entre seus terminais que ficam entre o capacitor e a base do transistor. Como o emissor do transistor e a placa inferior do capacitor estão conectadas ao mesmo nível de referência, o capacitor e o transistor formam uma *malha translinear dinâmica*. Aplicando a LTK nesta malha, obtêm-se a Eq. (2.32):

$$-V_{cap} + V_{const} + V_{BE} = 0 \tag{2.32}$$

Substituido a expressão de V_{BE}, dada pela Eq. (2.3), na Eq. (2.32), obtém-se:

$$-V_{cap} + V_{const} + V_T \ln\left(\frac{I_C}{\lambda I_S}\right) = 0 \tag{2.33}$$

Diferenciando a Eq. (2.33) obtém-se:

$$-\frac{dV_{cap}}{dt}+\frac{d}{dt}\left(V_T\ln\left(\frac{I_C}{\lambda I_S}\right)\right)=0 \tag{2.34}$$

$$-\frac{dV_{cap}}{dt}+V_T\left(\frac{\lambda I_S}{I_C}\right)\frac{d}{dt}\left(\frac{I_C}{\lambda I_S}\right)=0 \tag{2.35}$$

$$\frac{dV_{cap}}{dt}=V_T\left(\frac{1}{I_C}\right)\frac{dI_C}{dt} \tag{2.36}$$

Entretanto,

$$I_{cap}=C\frac{dV_{cap}}{dt} \tag{2.37}$$

Substituindo a expressão de I_{cap} na Eq. (2.36), e multiplicando-a pela capacitância C, tem-se a Eq. (2.38):

$$I_{cap}=CV_T\left(\frac{1}{I_C}\right)\frac{dI_C}{dt} \tag{2.38}$$

Mudando a notação de derivada para um ponto sobrescrito, a Eq. (2.38) pode ser rescrita na forma da Eq. (2.39) :

$$\begin{aligned} I_{cap} &= CV_T\frac{\dot{I_C}}{I_C} \\ CV_T\dot{I_C} &= I_{cap}I_C \end{aligned} \tag{2.39}$$

O princípio DTL é uma conseqüência direta da Eq. (2.39): *"A derivada no tempo de uma corrente pode ser mapeada em um produto de correntes"* [38]. O lado direito da Eq. (2.39) pode ser realizado facilmente utilizando o princípio STL, ou seja, a implementação de partes de uma equação diferencial é equivalente à implementação de um produto de correntes.

Pode ser que haja mais transistores na malha translinear dinâmica nos quais as tensões V_{BE} de suas junções não são constantes. A Fig. 2.5 mostra dois exemplos onde esta situação ocorre. Aplicando os mesmos cálculos das Eqs. (2.32) a Eq. (2.39) aos arranjos da Fig. 2.5 (a) e (b),

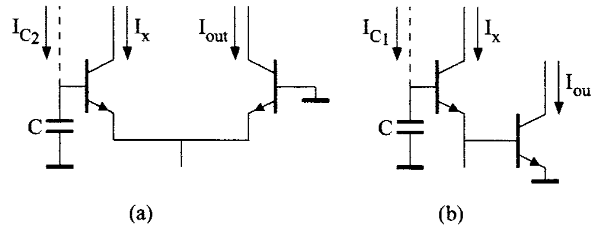


Figura 2.5: Arranjos distintos de correntes de capacitor. [38]

obtém-se, respectivamente, as Eqs. (2.40a) e (2.40b)

$$I_{C_2}=CV_T\left(\frac{\dot{I_X}}{I_X}-\frac{\dot{I_{out}}}{I_{out}}\right) \tag{2.40a}$$

$$I_{C_1}=CV_T\left(\frac{\dot{I_X}}{I_X}+\frac{\dot{I_{out}}}{I_{out}}\right) \tag{2.40b}$$

Assim, utilizando os índices SH e SAH para denominar os transistores com suas junções base-emissor no sentido horário e no sentido anti-horário de uma malha translinear dinâmica, I_j a corrente de coletor que flui pelo transistor j, e I_C a corrente do capacitor, o princípio DTL pode ser generalizado na forma da Eq. (2.41):

$$I_C = CV_T \left(\sum_{j \in \mathrm{SH}} \frac{\dot{I}_j}{I_j} - \sum_{j \in \mathrm{SAH}} \frac{\dot{I}_j}{I_j} \right) \tag{2.41}$$

2.6.1 Transformação de Variáveis Dinâmicas

Para implementar uma equação diferencial adimensional em um circuito translinear, além da transformação das variáveis estáticas vista na seção 2.4.2, deve-se transformar as derivadas adimensionais para derivadas no tempo, para que o princípio DTL possa ser aplicado. Assim, considera-se que as derivadas adimensionais são realizadas em relação à variável implícita τ. Como exemplo, a equação diferencial de primeira ordem dada pela Eq. (2.42) pode ser escrita na forma da Eq. (2.43):

$$\dot{y} + y = x \tag{2.42}$$

$$\frac{d}{d\tau} y(\tau) + y(\tau) = x(\tau) \tag{2.43}$$

Deve-se aplicar ao termo $d/d\tau$ uma transformação que gere d/dt na forma da Eq. (2.44), para manter os dois lados adimensionais:

$$\frac{d}{d\tau} = [\text{tempo}] \times \frac{d}{dt} \tag{2.44}$$

Entretanto, deve-se gerar "tempo" combinando unidades de variáveis de circuito (tensões, correntes, capacitâncias, resistências, etc.). Como isso pode ser feito de várias formas, deve-se procurar fazê-lo de uma forma que seja vantajosa para a realização do circuito. Como deseja-se utilizar o princípio DTL para implementar a derivação em um circuito, a constante CV_T da Eq. (2.39) é um bom ponto de partida. A unidade desta constante é dada pela Eq. (2.45):

$$[C] \times [V_T] = \frac{[\text{carga}]}{[\text{tensão}]} \times [\text{tensão}] = [\text{carga}] \tag{2.45}$$

Assim, basta agora encontrar uma unidade k que relacione carga e tempo:

$$[\text{tempo}] = k \times [\text{carga}] \tag{2.46}$$

$$k = \frac{[\text{tempo}]}{[\text{carga}]} \tag{2.47}$$

$$k = \frac{1}{[\text{corrente}]} \tag{2.48}$$

Assim, a constante CV_T/I_0 equivale a "tempo", sendo I_0 uma corrente constante, e a Eq. (2.49) mostra a transformação dada pela Eq. (2.44) modificada:

$$\frac{d}{d\tau} = \frac{CV_T}{I_0} \frac{d}{dt} \tag{2.49}$$

Uma conclusão direta da Eq. (2.49) é que o funcionamento do circuito agora depende da constante V_T, logo é sensível à variação de temperatura. Uma forma de contornar este problema é fazer a corrente I_0 PTAT[6].

Retomando o desenvolvimento da Eq. (2.43), aplicando a transformação dada pela Eq. (2.49), obtém-se:

$$\frac{CV_T}{I_0}\frac{d}{dt}y(\tau) + y(\tau) = x(\tau) \tag{2.50}$$

Agora deve-se transformar as variáveis adimensionais, em τ, em variáveis de correntes no tempo, conforme mostrado na seção 2.4.2. Utilizando uma única corrente I_{0_1} para realizar esta transformação, a Eq. (2.50) fica:

$$\frac{CV_T}{I_0}\frac{d}{dt}\frac{I_y}{I_{0_1}} + \frac{I_y}{I_{0_1}} = \frac{I_x}{I_{0_1}} \tag{2.51}$$

Eliminando-se I_{0_1} da Eq. (2.51), bem como mudando a notação de derivação para um ponto sobrescrito, a transformação de variáveis da Eq. (2.42) fica:

$$CV_T\dot{I}_y + I_0 I_y = I_0 I_x \tag{2.52}$$

A Eq. (2.52) é uma descrição modo-corrente de um filtro passa-baixa de primeira ordem [34]. Combinando a Eq. (2.52) com a Eq. (2.39), obtém-se o polinômio modo-corrente equivalente deste filtro:

$$I_{cap}I_y + I_0 I_y - I_0 I_x = 0 \tag{2.53}$$

A Eq. (2.53) pode ser realizada em um circuito translinear utilizando o princípio STL. Desta forma, a realização do circuito implica em uma malha STL superposta a uma malha DTL. Ao encontrar possíveis polinômios translineares equivalentes à Eq. (2.53), deve-se verificar se as malhas encontradas são compatíveis. Tomando como exemplo o desenvolvimento do filtro de primeira ordem dado pela Eq. (2.52), foi utilizada a malha DTL da Fig. 2.4 para gerar a Eq. (2.39), que por sua vez foi utilizada para substituir o termo $CV_T\dot{I}_y$ na Eq. (2.52). Se um polinômio translinear equivalente à Eq. (2.53) implicar em introduzir junções na malha DTL entre o capacitor e a base do transistor por onde flui I_y onde as tensões V_{BE} não são constantes, esta malha DTL não será compatível com a malha dada pela Fig. 2.4, logo este polinômio translinear não será válido.

Quando não é possível encontrar um polinômio translinear onde a malha STL seja compatível com a malha DTL, uma nova malha DTL deve ser especificada, e o projeto é reiniciado até que seja encontrado um polinômio translinear que tenha suas malhas STL e DTL compatíveis.

Assim, evidencia-se uma vantagem em se utilizar circuitos translineares: Funções de transferências, como filtros passivos por exemplo, podem ser implementadas em circuitos translineares utilizando apenas transistores e capacitores. Combinado à alta densidade de componentes proporcionado pelas tecnologias CMOS, pode-se criar circuitos de elevada complexidade funcional utilizando transistores MOS. Como os circuitos translineares dispensam o uso de resistores, que tendem a ter dimensões muito grandes quando se utiliza correntes muito pequenas, e quanto maior o resistor, maior tende a ser a sensibilidade do circuito em relação à temperatura, esta é uma vantagem muito importante para circuitos de baixíssimo consumo de energia.

[6] *Proportional To Absolute Temperature*—Proporcional à Temperatura Absoluta

2.7 Exemplo de Síntese: Circuito RMS-DC [26]

Uma forma de se implementar um circuito que fornece um valor DC[7] proporcional ao valor RMS[8] é dada pela Fig. 2.6, onde LPF (*low-pass filter*) se refere a um filtro passa-baixas. A equação

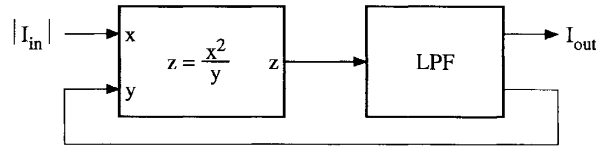


Figura 2.6: Diagrama de blocos de um conversor RMS-DC [26]

diferencial translinear do filtro utilizado é a dada pela Eq. (2.52), desenvolvida na seção anterior. A transformação de variáveis do bloco $z = x^2/y$ é dada por:

$$I_z = \frac{I_{in}^2}{I_y} \tag{2.54}$$

Substituindo $I_y = I_{out}$ e $I_x = I_z$ na Eq. (2.52), obtém-se a equação diferencial translinear, já com as variáveis estáticas transformadas:

$$CV_T \dot{I_{out}} I_{out} + I_0 I_{out}^2 - I_0 I_{in}^2 = 0 \tag{2.55}$$

Para encontrar a corrente de capacitor e transformar a Eq. (2.55) em um polinômio modo-corrente, define-se o circuito da malha translinear dinâmica na Fig. 2.7. Esta malha é definida com dois transistores, pois na Eq. (2.55) aparece I_{out}^2, e este ramo poderá implementar esta parte do circuito estático.

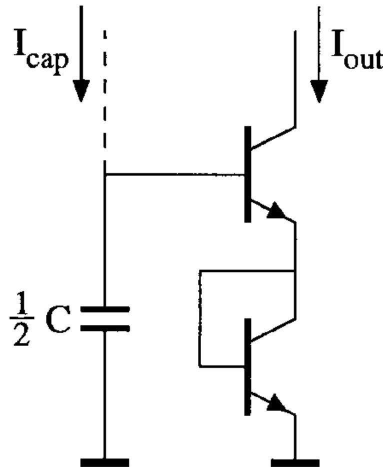


Figura 2.7: Malha translinear dinâmica RMS-DC [26]

Deste modo, aplicando a fórmula geral do princípio DTL dada pela Eq. (2.41), a relação entre

[7] *Direct Current*—Corrente contínua

[8] *Root Mean Square*—Raiz da Média Quadrática

o termo derivativo e a corrente de capacitor é obtida:

$$I_{cap} = \frac{1}{2}CV_T\left(\frac{\dot{I_{out}}}{I_{out}} + \frac{\dot{I_{out}}}{I_{out}}\right)$$

$$I_{cap} = 2\left(\frac{1}{2}C\right)V_T\left(\frac{\dot{I_{out}}}{I_{out}}\right)$$

$$I_{cap}I_{out} = CV_T\dot{I_{out}} \qquad (2.56)$$

Substituindo o resultado da Eq. (2.56) na Eq. (2.55), o polinômio modo-corrente é dado pela Eq. (2.57)

$$I_{cap}I_{out}^2 + I_0I_{out}^2 - I_0I_{in}^2 = 0 \qquad (2.57)$$

Aplicando-se decomposição não-paramétrica no polinômio modo-corrente da Eq. (2.57), obtém-se o polinômio translinear:

$$I_{out}^2(I_{cap} + I_0) - I_0I_{in}^2 = 0 \qquad (2.58)$$

O circuito relativo ao polinômio translinear da Eq. (2.58) é mostrado na Fig. 2.8. A malha estática é formada pelos transistores Q_1 a Q_6, sendo que os transistores conectados no sentido horário são Q_1 e Q_2, que implementam I_{in}, e Q_4, que implementa I_0. Já os transistores conectados no sentido anti-horário são Q_5 e Q_6, que implementam I_{out}, e Q_3 que implementa $(I_{cap} + I_0)$. A malha translinear dinâmica é dada pelo capacitor e os transistores Q_5 e Q_6.

Como o transistor Q_3, é polarizado pela corrente constante I_0, ele é visto pela malha dinâmica como uma fonte de tensão constante, portanto não tem influencia alguma na malha dinâmica. Os transistores Q_7 e Q_8 servem para compensar as correntes de base, e colocar Q_2 e Q_4 no modo *"diode-connected"*. Assim, as correntes em Q_1, Q_2 e Q_4 são fixadas, fazendo com que as correntes em Q_3, Q_5 e Q_6 variem de forma a manter a igualdade do polinômio-translinear implementado pelo circuito, dado pela Eq. (2.58).

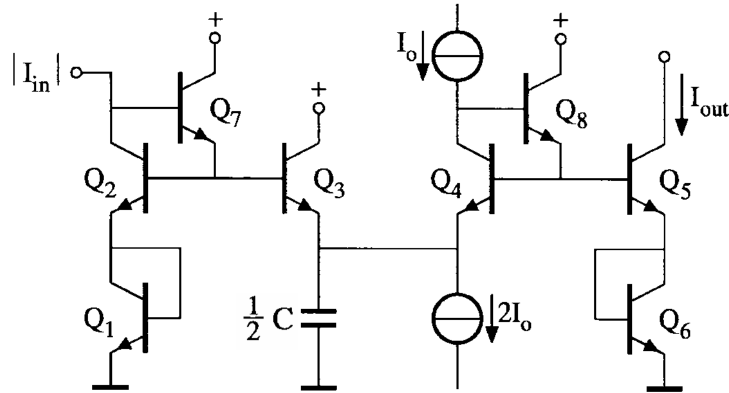


Figura 2.8: Circuito RMS-DC [26]. A malha estática é formada por Q_1 aQ_6; a malha dinâmica é formada por $1/2C$, e Q_5 a Q_6

Como a malha translinear dinâmica é compatível com a malha translinear estática, o projeto está pronto. Um ponto interessante é que a frequência de corte do filtro passa-baixas é dada

pela Eq. (2.59), mostrando que os circuitos translineares podem ter vários de seus parâmetros controláveis por meio das correntes utilizadas nas transformações de variáveis.

$$\omega_c = \frac{I_0}{CV_T} \tag{2.59}$$

2.7.1 Circuito da função Módulo [27]

O circuito RMS-DC assume que a corrente I_{in} está retificada antes de alimentá-lo. Assim, pode-se também realizar um circuito translinear estático que implemente a função módulo com esta finalidade. O polinômio adimensional que representa a função escolhida é dado pela Eq. (2.60)

$$y^2 = x^2 \tag{2.60}$$

$$y = \sqrt{x^2} \tag{2.61}$$

$$y = |x| \tag{2.62}$$

Assim, $y = -x$ se $x < 0$ e $y = x$ se $x > 0$. O polinômio modo-corrente relativo à Eq. (2.60) é dado pelo lado esquerdo da Eq. (2.63):

$$I_y^2 - I_x^2 = 0 \tag{2.63}$$

Entretanto, este polinômio não implementaria o circuito de forma satisfatória, pois se $I_x < 0$, então o transistor entraria em corte. Para solucionar o problema, pode-se adicionar e subtrair o monômio I_0^2 à Eq. (2.63) sem alterar o seu comportamento e mantendo o polinômio homogêneo:

$$-I_0^2 + I_y^2 - I_x^2 + I_0^2 = 0$$

$$(I_0^2 - I_x^2) - (I_0^2 - I_y^2) = 0 \tag{2.64}$$

Aplicando o produto notável $a^2 - b^2 = (a+b)(a-b)$ à Eq. (2.64), obtém-se o polinômio translinear dado pela Eq. (2.65)

$$(I_0 + I_x)(I_0 - I_x) - (I_0 + I_y)(I_0 - I_y) = 0 \tag{2.65}$$

O circuito que implementa a Eq. (2.65) pode ser visto na Fig. 2.9

Para verificar que este circuito de fato implementa a Eq. (2.65), calculam-se as correntes dos transistores que formam a malha estática $Q_2 - Q_5$. As correntes I_2 e I_3, que fluem pelos respectivos transistores Q_2 e Q_3 são analisadas conforme as Eqs. (2.66):

$$I_0 = I_3 + I_1 \tag{2.66a}$$

$$I_1 = I_2 \tag{2.66b}$$

$$I_0 = I_3 + I_2 \tag{2.66c}$$

$$I_2 = I_3 - I_{in} \tag{2.66d}$$

substituindo Eq. (2.66d) na Eq. (2.66c):

$$I_0 = I_3 + I_3 - I_{in} \tag{2.66e}$$

$$I_3 = \frac{1}{2}(I_0 - I_{in}) \tag{2.66f}$$

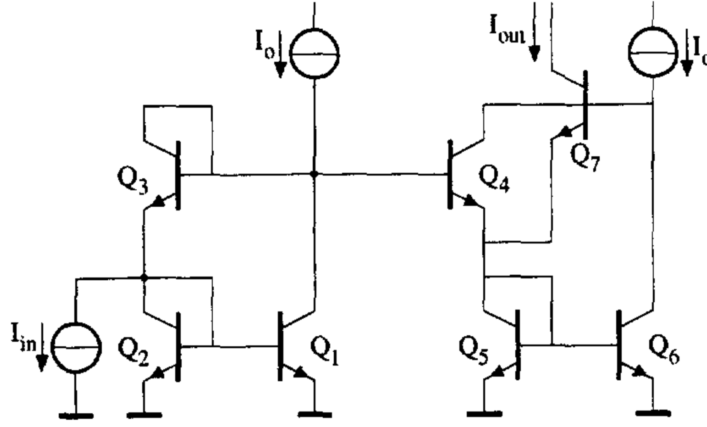


Figura 2.9: Circuito retificador de onda completa, com a malha translinear composta pelos transistores Q_2 a Q_5 [27]

substituindo Eq. (2.66f) na Eq. (2.66c):

$$I_0 = \frac{1}{2}(I_0 - I_{in}) + I_2 \tag{2.66g}$$

$$2I_2 = I_0 + I_{in} \tag{2.66h}$$

$$I_2 = \frac{1}{2}(I_0 + I_{in}) \tag{2.66i}$$

Assim:

$$I_2 = \begin{cases} \frac{1}{2}(I_0 + |I_{in}|) & \text{se } I_{in} > 0 \\ \frac{1}{2}(I_0 - |I_{in}|) & \text{se } I_{in} < 0 \end{cases}$$

e

$$I_3 = \begin{cases} \frac{1}{2}(I_0 - |I_{in}|) & \text{se } I_{in} > 0 \\ \frac{1}{2}(I_0 + |I_{in}|) & \text{se } I_{in} < 0 \end{cases}$$

Desta forma, desde que $|I_{in}| < I_0$, os transistores Q_2 e Q_3 estarão sempre com correntes de coletor positivas.

As correntes I_4 e I_5, que fluem pelos respectivos transistores Q_4 e Q_5 são analisadas conforme as Eqs. (2.67):

$$I_0 = I_4 + I_6 \tag{2.67a}$$

$$I_6 = I_5 \tag{2.67b}$$

$$I_0 = I_4 + I_5 \tag{2.67c}$$

$$I_4 = I_5 - I_{out} \tag{2.67d}$$

substituindo Eq. (2.67d) na Eq. (2.67c):

$$I_0 = I_5 + I_5 - I_{out} \tag{2.67e}$$

$$I_5 = \frac{1}{2}(I_0 - I_{out}) \tag{2.67f}$$

substituindo Eq. (2.67f) na Eq. (2.67c):

$$I_0 = \frac{1}{2}(I_0 - I_{out}) + I_4 \tag{2.67g}$$

$$2I_4 = I_0 + I_{out} \tag{2.67h}$$

$$I_4 = \frac{1}{2}(I_0 + I_{out}) \tag{2.67i}$$

Olhando novamente para o circuito da Fig. 2.9, os transistores Q_2 e Q_3 estão conectados no sentido horário, enquanto que Q_4 e Q_5 estão conectados em sentido anti-horário. Assim, este circuito implementa o polinômio modo corrente:

$$I_2 I_3 - I_4 I_5 = 0 \tag{2.68}$$

Substituindo os valores das correntes dados pelas Eqs. (2.66f), (2.66i), (2.67f), e (2.67i) na Eq. (2.68), obtém-se o polinômio translinear dado pela Eq. (2.69) que é equivalente ao polinômio translinear dado pela Eq. (2.65). Assim, o circuito implementa corretamente a função módulo.

$$\frac{1}{2}(I_0 + I_x)\frac{1}{2}(I_0 - I_x) - \frac{1}{2}(I_0 + I_y)\frac{1}{2}(I_0 - I_y) = 0 \tag{2.69}$$

O transistor Q_7 tem sua tensão base-emissor flutuante, logo fornece I_{out} para o nó entre o emissor de Q_4 e o coletor de Q_5, pois as correntes I_4 e I_5 são forçadas pela malha dada pela Eq. (2.65)

2.8 Outras Metodologias de Síntese Translinear

Em [22], um estudo sobre a convergência dos métodos de síntese de filtros translineares é realizado. As metodologias listadas nesse estudo são: mapeamento de espaço de estados exponenciais [39], a adaptação da técnica de filtros LC em escada [40], e os mesmos fundamentos utilizados na metodologia do trabalho desta dissertação em [28], estendidos com o uso da célula de Bernoulli [41]. O autor afirma que todas essas metodologias produzem uma combinação entre malhas estáticas e dinâmicas, e que, ainda que os autores considerem seus métodos únicos, eles são na verdade semelhantes. Um ponto em comum nos métodos apresentados em [22] é que são baseados em construir um bloco básico de um integrador translinear, e combiná-los de alguma forma para obter a funcionalidade desejada [22]. Além das metodologias descritas em [22], em [15] é realizada uma descrição completa de síntese de filtros translineares em espaço de estados, com detalhes de implementação em hardware.

Entretanto, as metodologias descritas acima limitam-se apenas a síntese de filtros translineares[9].

Assim, a metodologia utilizada nesta dissertação permite encontrar uma realização em circuito de qualquer função matemática, não limitando-se a síntese de filtros. Em relação à síntese de filtros, tem a vantagem de poder produzir circuitos mais compactos, pois busca realizá-los em poucas malhas translineares, preferencialmente em uma única, enquanto que nas outras metodologias citadas acima, o uso de blocos básicos pode resultar em circuitos de maiores dimensões, o que por sua vez pode levar a um desempenho inferior.

[9] Os filtros translineares são também chamados filtros *log-domain*

2.8.1 Algoritmo de Grafos de David Ilsen [25]

Nesse trabalho, um método semelhante ao da Fig. 2.1 é desenvolvido, porém a metodologia da etapa de decomposição polinomial é substituída por um algoritmo de classificação de padrões de polinômios baseado em teoria dos grafos. Este algoritmo cria um catálogo de todas as topologias de circuitos translineares para um dado número de transistores, bem como os polinômios translineares expandidos e simplificados associados a cada uma delas. A partir daí, procura-se ajustar o polinômio de entrada a uma ou mais topologias do catálogo que tenham maior probabilidade de representá-lo, por meio de classificação de padrões. A Fig. 2.10 mostra o catálogo de topologias com até seis transistores, onde cada ramo em um grafo representa um transistor, e a Tabela 2.2 mostra o número de topologias geradas pelo algoritmo para até nove transistores.

A vantagem deste algoritmo é que ele encontra o equivalente a decomposições paramétricas e não paramétricas do polinômio modo-corrente, pois como pode se perceber na Fig. 2.10, topologias com malhas múltiplas são consideradas. No entanto, a principal desvantagem, é que a a sua implementação disponibilizada depende do pacote de software proprietário MATHEMATICA©, não sendo assim, uma implementação amplamente acessível e de baixo custo.

Tabela 2.2: Número de topologias de circuitos translineares para um dado número de transistores [25]

numero de transistores	4	5	6	7	8	9
numero de topologias	2	3	19	39	174	559

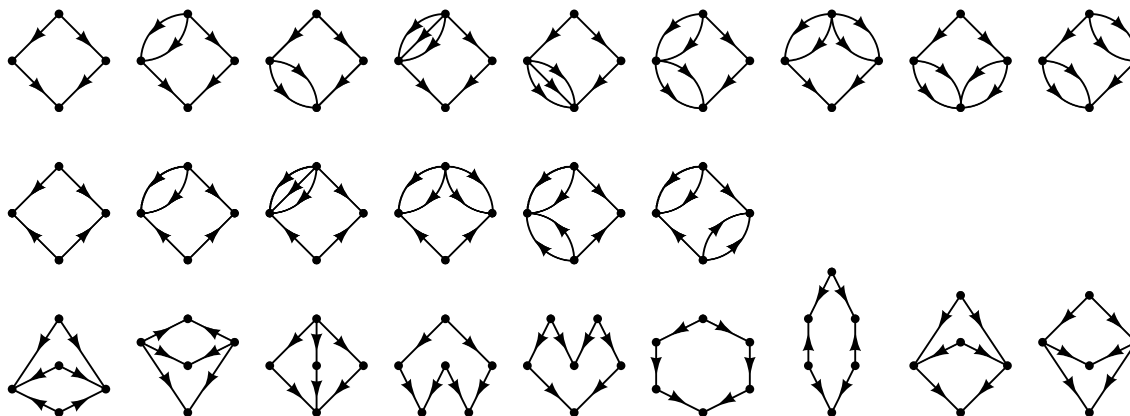


Figura 2.10: Topologias de circuitos translineares contendo até seis transistores [25]

Capítulo 3

Algoritmos de Decomposição Translinear não-paramétrica

3.1 Introdução

Neste capítulo são apresentados três algoritmos de decomposição não-paramétrica: Um algoritmo trivial, o algoritmo de Mulder et. al. [23][1], e o algoritmo desenvolvido neste trabalho[2]. Estes algoritmos são apresentados já da forma que foram implementados nos fluxografos desta seção.

3.2 Análise Combinatória de um Algoritmo Trivial

Para mostrar a necessidade de se criar um algoritmo complexo de decomposição não-paramétrica, primeiro desenvolveu-se um algoritmo simples de pura força-bruta, no qual todas as combinações de polinômios-base na forma da Eq. (2.12) são testadas e comparadas com o polinômio modo-corrente relativo ao polinômio de entrada E_r. Para realizar a análise de viabilidade deste algoritmo, basta calcular quantas combinações serão testadas.

A partir de um vetor de polinômios-base contendo N_{β_0} elementos, N_{β_0} dado pela Eq. (2.13), deve-se calcular quantas combinações destes polinômios-base encaixam-se no modelo da Eq. (2.12). Como a multiplicação possui propriedade comutativa, cada lado da equação pode ter uma combinação entre N_{β_0} polinômios tomados r a r com repetição, pois um polinômio-base pode estar multiplicado mais de uma vez. Define-se então o número $A(N_{\beta_0}, r)$ na Eq. (3.2). Finalmente, como cada produto da Eq. (2.12) pode assumir $A(N_{\beta_0}, r)$ combinações, define-se o número $N_{trivial}$ na Eq. (3.3) como a combinação do número $A(N_{\beta_0}, r)$ tomado 2 a 2.

$$CR_{(n,p)} = \frac{n + p - 1!}{p!(n-1)!} \tag{3.1a}$$

$$C_{(n,p)} = \frac{n!}{p!(n-p)!} \tag{3.1b}$$

[1]Deste ponto em diante este algoritmo será chamado de “algoritmo original”

[2]Deste ponto em diante este algoritmo será chamado de “novo algoritmo”

$$A(N_{\beta_0}, r) = \frac{N_{\beta_0} + r - 1!}{r!(N_{\beta_0} - 1)!} \tag{3.2}$$

$$N_{trivial}(N_{\beta_0}, r) = \frac{\left(\frac{(N_{\beta_0}+r-1)!}{r!(N_{\beta_0}-1)!}\right)!}{2!\left[\left(\frac{(N_{\beta_0}+r-1)!}{r!(N_{\beta_0}-1)!}\right) - 2\right]!} \tag{3.3}$$

Para exemplificar, utilizando a Eq. (3.3), calcula-se o número de combinações possíveis para se encontrar todos os polinômios modo-corrente válidos de um polinômio de terceiro grau, três variáveis e com coeficientes de -5 a 5 ($n = 3, r = 3, N = 5$).

$$\begin{aligned} N_{\beta_0} &= (2 * 5 + 1)^3 = 1331 \\ A(1331, 3) &= \frac{1331 + 3 - 1!}{3!(1331 - 1)!} = 393.877.506 \\ N_{trivial}(1331, 3) &= \frac{393.877.506!}{2!(393.877.506 - 2)!} = 7,75697447 \times 10^{16} \end{aligned} \tag{3.4}$$

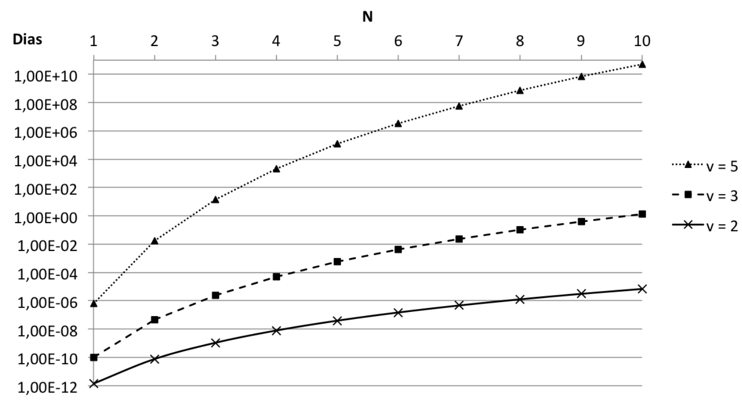
Se fosse tentado encontrar polinômios translineares por meio de um processador hipotético de 8 GHz (recorde de velocidade nesta data), e que toda a operação de multiplicar, expandir, simplificar a combinação de polinômios-base e compará-lo com o polinômio modo-corrente E_r coubesse em apenas um ciclo de processamento, o tempo necessário para encontrar todas os polinômios translineares possíveis seria:

$$t = \frac{7,75697447 \times 10^{16}}{8 \times 10^9} \approx 9.696.218 \text{ segundos} \approx 112 \text{ dias} \tag{3.5}$$

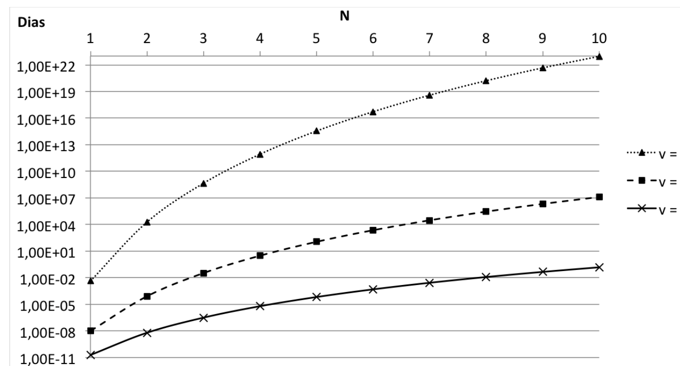
As Figuras 3.1a à 3.1d mostram o número de dias necessários para se testar todas as possibilidades de polinômios translineares utilizando o algoritmo trivial, para diferentes valores de N, r, e v. Pode-se perceber que o uso do algoritmo trivial leva um número elevado de dias para ser executado para os valores $N \geq 4$, $v \geq 3$ e $r \geq 3$. Conclui-se que um algoritmo mais eficiente é necessário.

3.3 Algoritmo de decomposição não-paramétrica original [23]

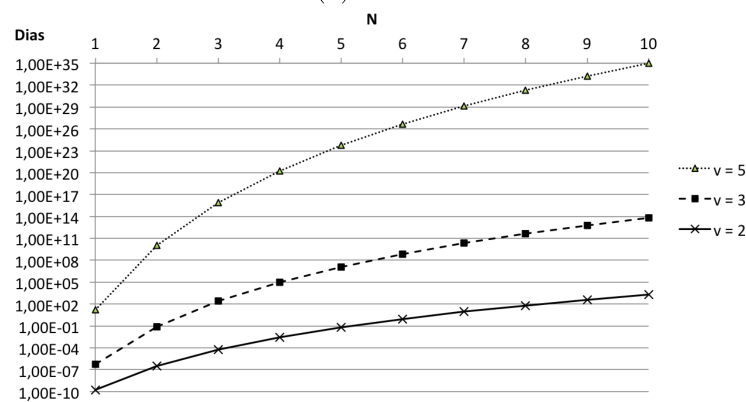
De acordo com a pesquisa realizada, o algoritmo de decomposição não-paramétrica desenvolvido em [23] aparentemente é o único publicado até hoje. Ele explora a forma da decomposição translinear dada pela Eq. (2.12) para usar a operação de divisão por pares de polinômios e expansão em frações parciais. Isso permite encontrar polinômios translineares de uma forma mais inteligente que o algoritmo trivial visto na seção 3.2. O algoritmo é dividido em três etapas: A Geração do Vetor de Polinômios-Base, a Divisão recursiva por pares de polinômios-base e a Verificação final. A Fig. 3.2 mostra um diagrama com estas etapas, e as seções seguintes descrevem cada uma delas. Como o autor do algoritmo original não apresentou a estrutura da implementação de seu algoritmo, duas versões da etapa de “Divisão recursiva por pares de polinômios-base” foram implementadas neste trabalho: uma versão otimizada para minimizar o tempo de execução, e uma versão otimizada para minimizar a memória utilizada.



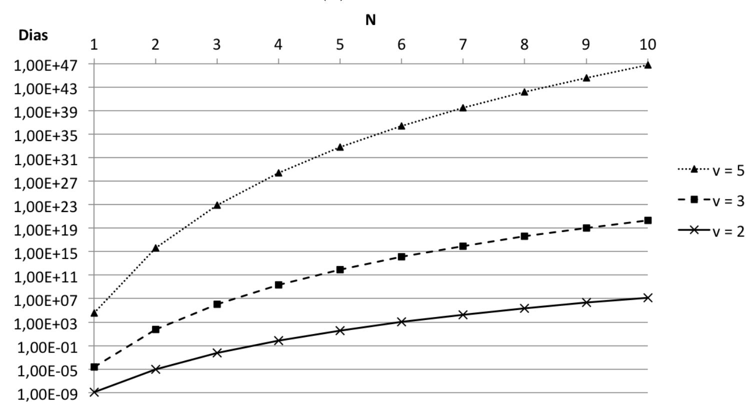
(a) Grau 2



(b) Grau 3



(c) Grau 4



(d) Grau 5

Figura 3.1: Dias necessários para execução do algoritmo trivial para diversos valores de N e v. Cada gráfico corresponde a diferentes graus r do polinômio de entrada E_r

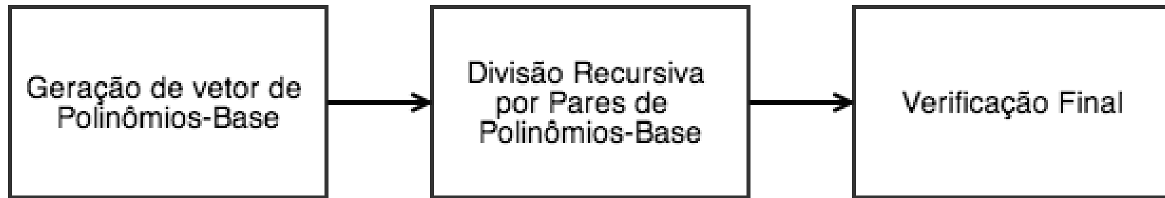


Figura 3.2: Estrutura de topo da implementação do algoritmo original de decomposição não-paramétrica [23]

3.3.1 Geração do vetor de Polinômios-Base

Como visto na seção 3.2, o número de polinômios-base N_{β_0} do vetor β_0 tem um grande impacto no número de computações, não só do algoritmo trivial, mas também de qualquer algoritmo que se possa conceber. Esse algoritmo utiliza duas estratégias para reduzir o número de polinômios-base: A obtenção de polinômios-base estritamente positivos, que gera o vetor β_1 a partir do vetor β_0, e a Divisão por um polinômio base e fatoração do resto, que gera o vetor β_2 a partir do vetor β_1.

3.3.1.1 Obtenção de Polinômios-Base Estritamente Positivos

Como os polinômios-base representam correntes de coletor, a primeira coisa a se fazer é excluir todos os polinômios-base que não são estritamente positivos. Parte-se do princípio que, ao realizar a síntese de um circuito translinear, todos os limites superiores e inferiores das correntes sejam conhecidos. Isso é trivial para variáveis estáticas, mas para variáveis dinâmicas é necessário conhecer as formas de onda e as freqüências das correntes de entrada, e resolver a equação diferencial para obter os limites das correntes de saída. Para determinar se o polinômio-base é estritamente positivo, basta substituir os respectivos limites inferiores nas variáveis com coeficientes positivos, e os respectivos limites superiores nas variáveis com coeficientes negativos, e caso a soma der positiva, o polinômio é estritamente positivo, e é copiado do vetor β_0 para o vetor β_1. A ilustração deste procedimento pode ser vista na Fig. 3.3

3.3.1.2 Divisão por um Polinômio-Base e fatoração do resto

A forma geral de um polinômio translinear dada pela Eq. (2.12) pode ser explorada para verificar se um polinômio-base pode fazer parte de um polinômio translinear. Suponha que um polinômio homogêneo de terceiro grau E_3, seja equivalente ao polinômio translinear dado pela Eq. (3.6). Se dividirmos ambos os lados desta equação por um de seus polinômios-base, P_1 por exemplo, o resultado é dado pela Eq. (3.7). Assim, se o resto da divisão, representado neste exemplo por $P_2P_4P_6$, não for favorável em r polinômios lineares, então não existe polinômio translinear equivalente a E_r na qual P_1 possa fazer parte, logo não deve ser copiado no vetor β_2. Todos os polinômios-base do vetor β_1 são submetidos a este teste, e os polinômios-base não-eliminados são

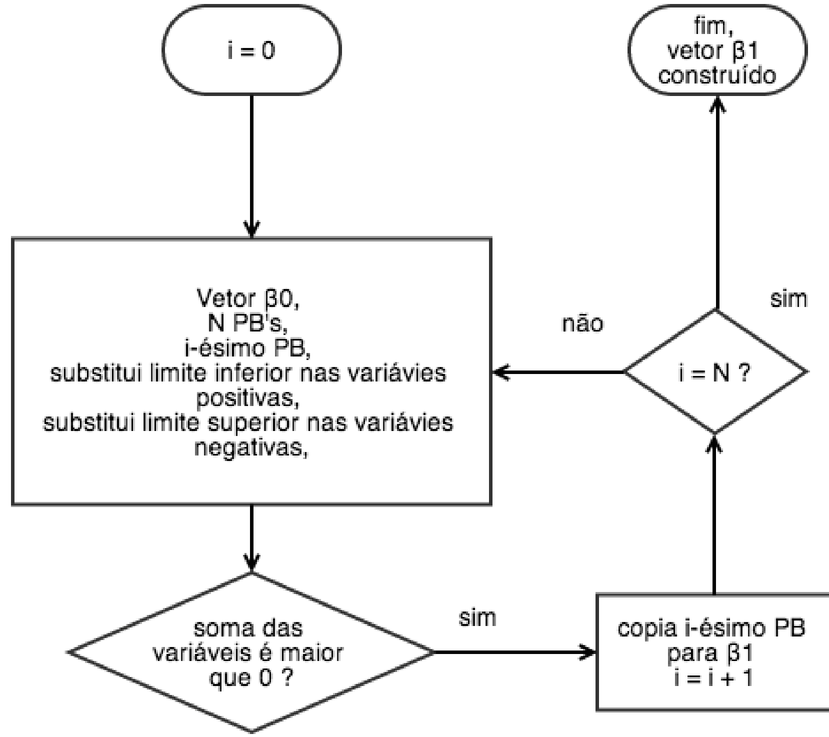


Figura 3.3: Obtenção de Polinômios-Base (PB's) estritamente positivos. O vetor β_1 é obtido a partir do vetor β_0. O índice 'i' é utilizado para percorrer o vetor β_0

copiados no vetor β_2. A Figura 3.4 ilustra este sub-algoritmo.

$$E_3 = \lambda_1 P_1 P_3 P_5 - \lambda_2 P_2 P_4 P_6 \tag{3.6}$$

$$\frac{E_3}{P_1} = \lambda_1 P_3 P_5 - \lambda_2 \frac{P_2 P_4 P_6}{P_1} \tag{3.7}$$

Como exemplo, fazendo E_3 igual ao polinômio da Eq. (2.21), ele é dividido por $P_1 = (6I_{in} + 5I_{out} + 6I_0)$, e o resultado com o resto R obtido fatorado é mostrado na Eq. (3.8). O Quociente Q não é calculado, pois não é relevante neste teste.

$$E_3 = {I_{in}}^3 + {I_{in}}^2 I_{out} - {I_0}^2 I_{in} + {I_0}^2 I_{out}; \qquad P_1 = (6I_{in} + 5I_{out} + 6I_0)$$

$$\frac{{I_{in}}^3 + {I_{in}}^2 I_{out} - {I_0}^2 I_{in} + {I_0}^2 I_{out}}{(6I_{in} + 5I_{out} + 6I_0)} = Q + \frac{1}{216} \frac{I_{out}(5I_{out} - 12I_0)(5I_{out} - 6I_0)}{(6I_{in} + 5I_{out} + 6I_0)} \tag{3.8}$$

$$R = \frac{1}{216} I_{out}(5I_{out} - 12I_0)(5I_{out} - 6I_0) \tag{3.9}$$

O resto dado pela Eq. (3.9) é favorável em 3 polinômios lineares, logo $(6I_{in} + 5I_{out} + 6I_0)$ pode fazer parte de um polinômio translinear equivalente ao polinômio dado pela Eq. (2.21), e deve ser copiado no vetor β_2.

Um outro exemplo é realizado utilizando o mesmo E_3 do exemplo anterior, mas fazendo $P_1 = (-I_{in} + I_{out} + I_0)$. Repete-se o procedimento de divisão por P_1 e fatoração do resto, e o resultado, já com o resto R fatorado, é mostrado na Eq. (3.10).

$$E_3 = {I_{in}}^3 + {I_{in}}^2 I_{out} - {I_0}^2 I_{in} + {I_0}^2 I_{out}; \qquad P_1 = (-I_{in} + I_{out} + I_0)$$

$$\frac{I_{in}{}^3 + I_{in}{}^2 I_{out} - I_0{}^2 I_{in} + I_0{}^2 I_{out}}{(-I_{in} + I_{out} + I_0)} = Q + \frac{I_{out}(2I_{out}^2 + 5I_{out}I_0 + 4I_0{}^2)}{(-I_{in} + I_{out} + I_0)} \tag{3.10}$$

$$R = I_{out}(2I_{out}^2 + 5I_{out}I_0 + 4I_0{}^2) \tag{3.11}$$

O resto dado pela Eq. (3.11) é fatorável em um polinômio linear e um polinômio quadrático irredutível, logo o polinômio $(-I_{in} + I_{out} + I_0)$ não deve ser copiado para o vetor β_2

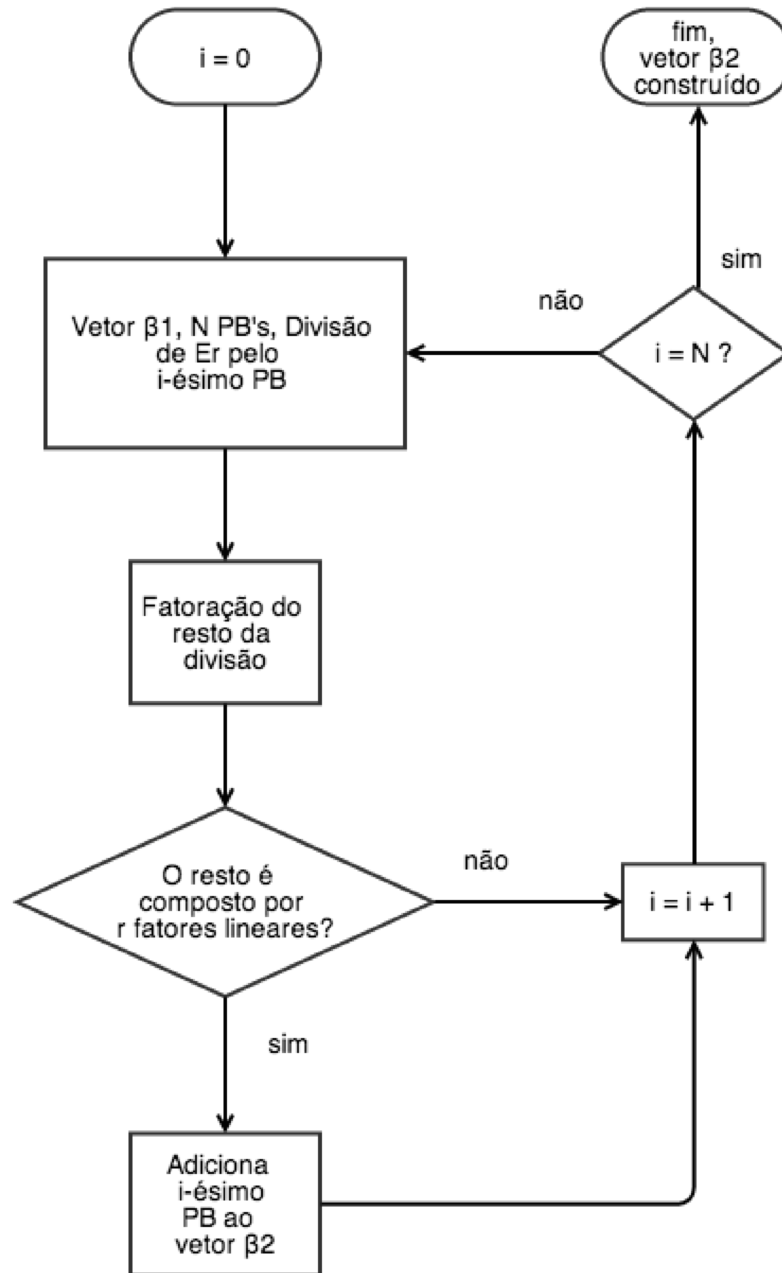


Figura 3.4: Divisão por um Polinômio-Base (PB) e fatoração do Resto. O índice 'i' é utilizado para percorrer o vetor β_1

3.3.2 Divisão recursiva por pares de polinômios-base, versão otimizada para minimizar tempo de execução

Nesta etapa os polinômios-base do vetor β_2 são combinados por meio de divisões polinomiais e expansão em frações parciais dos restos. Este método armazena listas de polinômios-base que são reutilizados em memória, a fim de evitar repetições de rotinas desnecessárias. Entretanto, não há como controlar o tamanho destas listas em memória, tornado esta versão insegura nesse sentido. As seções a seguir detalham as etapas deste sub-algoritmo.

3.3.2.1 Divisão por pares de Polinômios-Base e Expansão do Resto em frações parciais

Uma forma efetiva de verificar quais polinômios-base podem fazer parte de um polinômio translinear juntos, é dividir o polinômio de entrada E_r por um par de polinômios base e expandir o resto em frações parciais. Para ilustrar este exemplo, pode-se assumir um polinômio E_3 que que seja equivalente a um polinômio translinear conforme a Eq.(3.6). A Eq. (3.12) mostra a divisão de E_r por 2 polinômios-base, P_1 e P_2, que fazem parte do polinômio translinear, mas pertencem a produtos opostos. Nota-se que o grau dos restos nos numeradores sobre P_1 e P_2 possuem grau menor que E_3, e podem ser fatorados em $r-1$ polinômios lineares. O Quociente da divisão Q é irrelevante neste teste, logo não é calculado.

$$\frac{E_3}{P_1 P_2} = Q + \lambda_1 \frac{P_3 P_5}{P_2} - \lambda_2 \frac{P_4 P_6}{P_1} \tag{3.12}$$

Deste modo, pode-se generalizar esta etapa baseando-se nas Eqs. (3.13) e (3.14) da seguinte forma: Divide-se o polinômio homogêneo E_r por um par de polinômios-base, P_x e P_y, pertencentes a um vetor de polinômios-base. Expande-se o resto desta divisão em frações parciais em relação a uma variável comum entre P_x e P_y[3]. Se os numeradores desta expansão, R_x e R_y, fore de grau $r-1$ e ambos fatoráveis em $r-1$ polinômios lineares, então P_x e P_y podem, juntos, fazer parte de um polinômio translinear equivalente a E_r como polinômios opostos.

$$\frac{E_r}{P_x P_y} = Q + \frac{R_x}{P_x} + \frac{R_y}{P_y} \tag{3.13}$$

$$E_r = \lambda_1 P_x P_1 \ldots P_{2r-1} - \lambda_2 P_y P_2 \ldots P_{2r} \tag{3.14}$$

Esta etapa do algoritmo divide o polinômio de entrada E_r por todas as combinações entre polinômios-base do vetor β_2, e a cada vez que um resultado satisfatório é encontrado, a próxima etapa, "Divisão recursiva por pares de polinômios-base", é iniciada, e ao final desta, a etapa de "Verificação final" também é executada. Assim, este sub-algoritmo é o núcleo da etapa geral de "Divisão Recursiva por Pares de Polinômios-Base", e seu fluxografo é descrito na Fig. 3.5

Como exemplo, a Eq. (2.21) é dividida e por $(6I_{in} + 5I_{out} + 6I_0)$ junto com $(5I_{in} + 5I_{out} + 4I_0)$ e $(I_{in} + I_{out} + I_0)$ junto com $(-I_{in} + I_{out})$, e os resultados, já com o restos expandidos em frações

[3]Se não houver uma variável em comum, pode-se usar uma variável qualquer de P_x e outra de P_y

parciais e fatorados, são mostrados nas Eqs. (3.15) e (3.16), respectivamente. Os Quocientes Q não são calculados, pois não influenciam neste teste.

$$\frac{{I_{in}}^3 + {I_{in}}^2 I_{out} - {I_0}^2 I_{in} + {I_0}^2 I_{out}}{(6I_{in} + 5I_{out} + 6I_0)(5I_{in} + 5I_{out} + 4I_0)} =$$
$$Q + \frac{1}{36}\left(\frac{I_{out}(5I_{out} - 12I_0)}{6I_{in} + 5I_{out} + 6I_0}\right) + \frac{2}{25}\left(\frac{I_0(10I_{out} + 3I_0)}{5I_{in} + 5I_{out} + 4I_0}\right) \tag{3.15}$$

$$\frac{{I_{in}}^3 + {I_{in}}^2 I_{out} - {I_0}^2 I_{in} + {I_0}^2 I_{out}}{(+I_{in} + I_{out} + I_0)(-I_{in} + I_{out})} =$$
$$Q + \frac{1}{(2I_{out} + I_0)}\left(\frac{-I_{out}^2 I_0}{I_{in} + I_{out} + I_0} + \frac{2I_0^3}{-I_{in} + I_{out}}\right) \tag{3.16}$$

Na Eq. (3.15), os restos numeradores das frações parciais são de grau 2, um a menos que o Polinômio de entrada, e são fatoráveis em 2 polinômios lineares, logo $(6I_{in} + 5I_{out} + 6I_0)$ e $(5I_{in} + 5I_{out} + 4I_0)$ podem formar juntos um polinômio translinear como polinômios opostos. Já na Eq. (3.16), o grau dos numeradores das frações parciais é igual ao grau do polinômio de entrada, então $(I_{in} + I_{out} + I_0)$ e $(-I_{in} + I_{out})$ não podem formar juntos um polinômio translinear como polinômios opostos.

3.3.2.2 Geração Recursiva de decomposições parciais

Já que a divisão de E_r por um par de polinômios-base P_1 e P_2 e a expansão em frações parciais dos restos pode ser usada para verificar se este par pode fazer parte de um polinômio translinear como polinômios opostos, então os numeradores do resultado da expansão em frações parciais dos restos podem ser usados para iniciar um processo recursivo de divisão por 2 polinômios e expansão em frações parciais do novo resto, e assim sucessivamente, até que o resto seja somente uma constante.

Denominando $P_{x,n}$, $Q_{x,n}$, e $R_{x,n}$ Polinômios-base, Quocientes, e Restos de índice x e grau n, Divide-se inicialmente, o polinômio E_r por um par de polinômios-base, P_1 e P_2, e obtém-se:

$$\frac{E_r}{P_1 P_2} = Q_{r,r-2} + \frac{R_{1,r-1}}{P_1} + \frac{R_{2,r-1}}{P_2} \tag{3.17}$$

e os Restos $R_{1,r-1}$ e $R_{2,r-1}$ são usados para iniciar outra divisão com expansão em frações parciais independentes:

$$\frac{R_{1,r-1}}{P_1 P_4} = Q_{1,r-2} + \frac{R_{1,r-2}}{P_1} + \frac{R_{4,r-2}}{P_4} \tag{3.18}$$
$$\frac{R_{2,r-1}}{P_2 P_3} = Q_{2,r-2} + \frac{R_{2,r-2}}{P_2} + \frac{R_{3,r-2}}{P_3} \tag{3.19}$$

e cada polinômio que gerar uma expansão em frações parciais bem-sucedida junto com o seu polinômio-base de origem (P_4 com P_1 e P_3 com P_2) é armazenado em um conjunto temporário de polinômios δ, e o processo recursivo continua dividindo o último numerador sobre o polinômio de

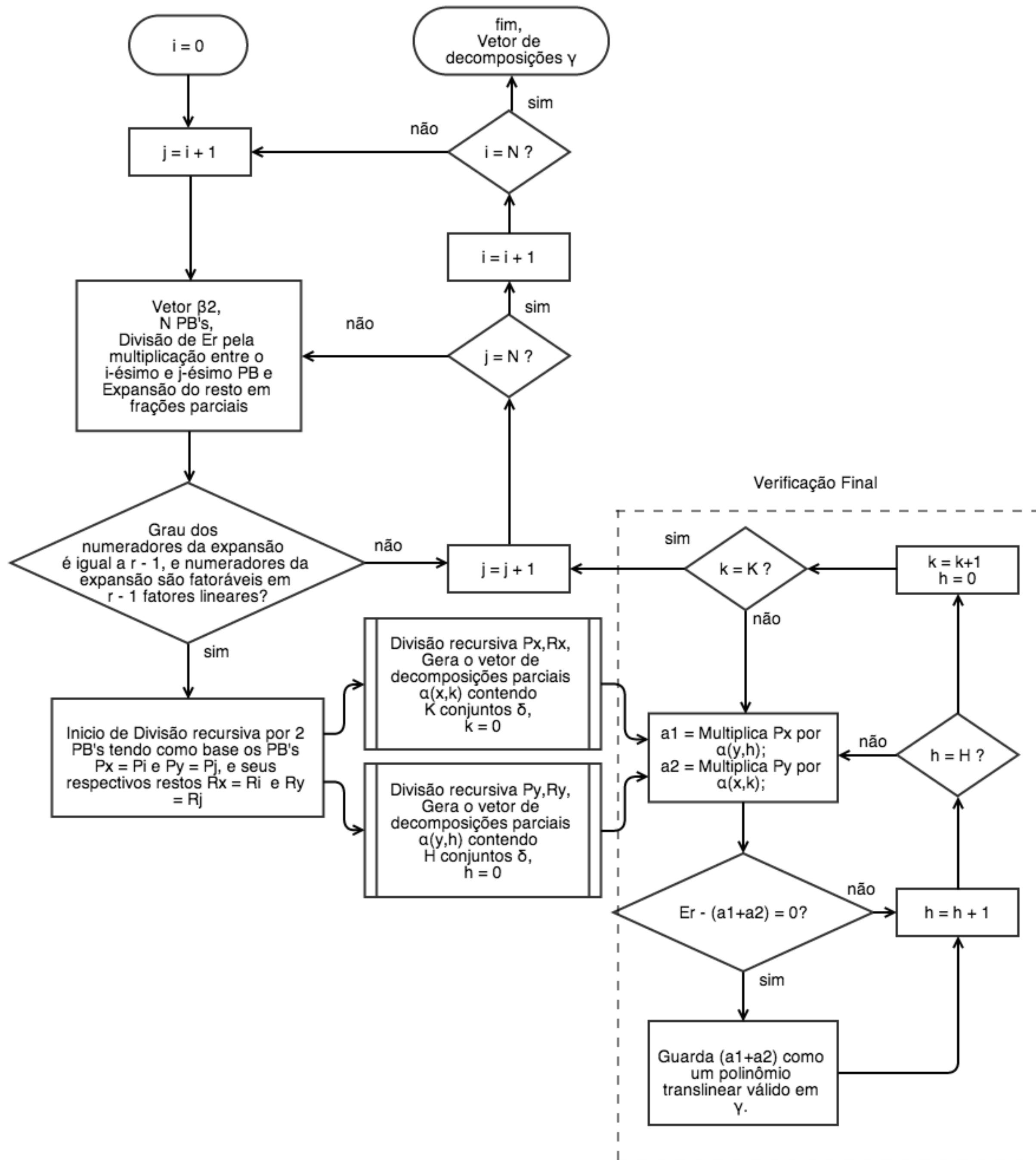


Figura 3.5: Divisão por pares de Polinômios-base e execução de decomposições parciais e Verificação final, versão otimizada para tempo de execução. Os índices 'i' e 'j' são utilizados para percorrer o vetor β_2 e gerar pares de polinômios-base. Os índices 'k' e 'h' ão utilizados para percorrer as listas de decomposições parciais $\alpha_{x,k}$ e $\alpha_{y,h}$. O procedimento de Verificação final está destacado dentro da caixa pontilhada

origem $(R_{1,x}, R_{2,x})$ pelo polinômio de origem multiplicado a um outro polinômio-base de β_2, até que os restos sejam constantes, ou seja, tenham grau 0 $(R_{1,0}, R_{2,0})$.

Neste ponto, δ é copiado para uma lista de decomposições parciais encontradas para o seu

respectivo polinômio-base $\alpha_{x,n}$,onde x refere-se ao índice do polinômio-base de referência, e n é o índice do conjunto δ copiado para a lista $\alpha_{x,n}$. Isso significa que P_1 terá uma lista $\alpha_{1,n}$ de conjuntos de polinômios-base que, quando multiplicados, podem fazer parte de um polinômio translinear tendo P_1 como polinômio oposto, e P_2 também terá sua respectiva lista $\alpha_{2,n}$.

Como o processo recursivo relativo a P_1 não depende dos resultados do processo relativo a P_2, o problema se divide em dois problemas separados. Os quocientes $Q_{x,n}$ e os restos que não são relativos aos polinômios-base de origem são descartados durante este processo. Este sub-algoritmo pode ser melhor visualizado na Fig. 3.6.

Como exemplo, utiliza-se o resultado da expansão em frações parciais da Eq. (3.15) para realizar o procedimento de decomposições parciais, tem-se o primeiro passo tendo P_1 como polinômio de referência:

$$P_1 = (6I_{in} + 5I_{out} + 6I_0), \qquad \delta = [R_{1,2}]$$
$$R_{1,2} = \frac{1}{36} I_{out}(5I_{out} - 12I_0)$$

Escolhe-se $P_4 = (5I_{in} + 5I_{out} + 3I_0)$:

$$\frac{1}{36} \frac{I_{out}(5I_{out} - 12I_0)}{(6I_{in} + 5I_{out} + 6I_0)(5I_{in} + 5I_{out} + 3I_0)} =$$

$$Q_{1,0} + \frac{1}{5}\left(\frac{-I_{in} - I_0}{6I_{in} + 5I_{out} + 6I_0}\right) + \frac{1}{36}\left(\frac{5I_{in} + 3I_0}{5I_{in} + 5I_{out} + 3I_0}\right) \tag{3.20}$$

Como a expansão em frações parciais foi bem-sucedida, adiciona-se P_4 e substitui-se $R_{1,2}$ por $R_{1,1}$ em δ. Segundo passo:

$$P_1 = (6I_{in} + 5I_{out} + 6I_0), \qquad \delta = [R_{1,1}, (5I_{in} + 5I_{out} + 3I_0)]$$
$$R_{1,1} = \frac{1}{5}(-I_{in} - I_0)$$

Escolhe-se $P_6 = I_{out}$:

$$\frac{1}{5} \frac{(-I_{in} - I_0)}{(6I_{in} + 5I_{out} + 6I_0)(I_{out})} =$$

$$\frac{1}{6}\left(\frac{1}{6I_{in} + 5I_{out} + 6I_0}\right) + \frac{1}{30}\left(\frac{-1}{5I_{in} + 5I_{out} + 3I_0}\right) \tag{3.21}$$

Como a expansão em fações parciais foi bem-sucedida, adiciona-se P_6 e substitui-se $R_{1,1}$ por $R_{1,0}$ em δ.

$$P_1 = (6I_{in} + 5I_{out} + 6I_0), \qquad \delta = [\frac{1}{6}, (5I_{in} + 5I_{out} + 3I_0), I_{out}] \tag{3.22}$$
$$R_{1,0} = \lambda_1 = \frac{1}{6}$$

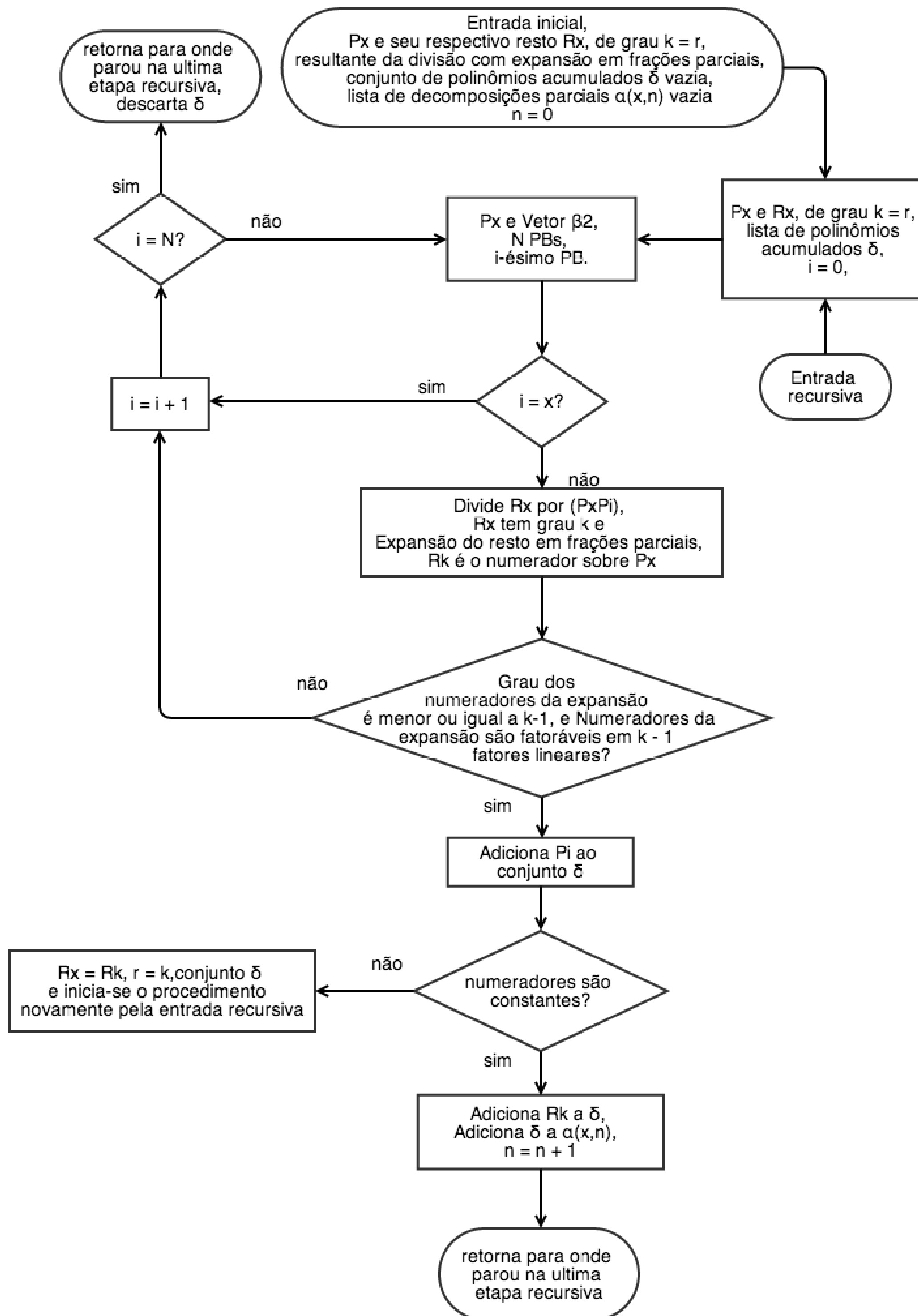


Figura 3.6: Sub-rotina de geração da lista de decomposições parciais $\alpha_{x,n}$. O índice 'i' é utilizado para percorrer o vetor β_2, e o índice 'n' guarda o número de conjuntos δ gravados na lista $\alpha_{x,n}$

Como $R_{1,0}$ é constante, δ é copiada para a lista $\alpha_{1,0}$ de decomposições parciais relativas a P_1.

Agora executando a decomposição parcial para P_2:

$$P_2 = (5I_{in} + 5I_{out} + 4I_0), \qquad \delta = [R_{1,2}]$$
$$R_{2,2} = \frac{2}{25} I_0(10I_{out} + 3I_0)$$

Escolhe-se $P_3 = (-I_{in} + 5I_{out} + I_0)$:

$$\frac{2}{25}\frac{I_0(10I_{out} + 3I_0)}{(5I_{in} + 5I_{out} + 4I_0)(-I_{in} + 5I_{out} + I_0)} =$$

$$Q_{2,0} + \frac{2}{15}\left(\frac{I_0}{5I_{in} + 5I_{out} + 4I_0}\right) + \frac{2}{75}\left(\frac{I_0}{-I_{in} + 5I_{out} + I_0}\right) \tag{3.23}$$

Como a expansão em fações parciais foi bem-sucedida, adiciona-se P_3 e substitui-se $R_{2,2}$ por $R_{2,1}$ em δ. Segundo Passo:

$$P_2 = (5I_{in} + 5I_{out} + 4I_0), \qquad \delta = [R_{2,1}, (-I_{in} + 5I_{out} + I_0)]$$
$$R_{2,1} = \frac{2}{15}(I_0)$$

Escolhe-se $P_5 = (I_{in} + I_{out})$:

$$\frac{2}{15}\frac{I_0}{(5I_{in} + 5I_{out} + 4I_0)(I_{in} + I_{out})} =$$

$$\frac{1}{6}\left(\frac{-1}{5I_{in} + 5I_{out} + 4I_0}\right) + \frac{1}{30}\left(\frac{1}{I_{in} + I_{out}}\right) \tag{3.24}$$

Como a expansão em fações parciais foi bem-sucedida, adiciona-se P_5 e substitui-se $R_{2,1}$ por $R_{2,0}$ em δ.

$$P_2 = (5I_{in} + 5I_{out} + 4I_0), \qquad \delta = [-\frac{1}{6}, (-I_{in} + 5I_{out} + I_0), (I_{in} + I_{out})] \tag{3.25}$$
$$R_{2,0} = \lambda_2 = -\frac{1}{6}$$

Como $R_{2,0}$ é constante, δ é copiada para a lista $\alpha_{2,0}$ de decomposições parciais relativas a P_2.

3.3.3 Verificação Final

Para entender melhor porque o processo de divisões recursivas e expansões dos restos em frações parciais é eficaz, repete-se abaixo todas as operações de expansão de um polinômio de terceiro grau

E_3:

$$\frac{E_3}{P_1P_2} = Q_{3,1} + \frac{R_{1,2}}{P_1} + \frac{R_{2,2}}{P_2} \tag{3.26a}$$

$$\frac{R_{1,2}}{P_1P_4} = Q_{1,0} + \frac{R_{1,1}}{P_1} + \frac{R_{4,1}}{P_4} \tag{3.26b}$$

$$\frac{R_{1,1}}{P_1P_6} = \frac{R_{1,0}}{P_1} + \frac{R_{6,0}}{P_6} \tag{3.26c}$$

$$\frac{R_{2,2}}{P_2P_3} = Q_{2,0} + \frac{R_{2,1}}{P_2} + \frac{R_{3,1}}{P_3} \tag{3.26d}$$

$$\frac{R_{2,1}}{P_2P_5} = \frac{R_{2,0}}{P_2} + \frac{R_{5,0}}{P_5} \tag{3.26e}$$

Agora, eliminam-se os denominadores, de modo a isolar os termos $R_{1,x}$ e $R_{2,x}$, obtendo-se o novo conjunto:

$$E_3 = P_1P_2Q_{3,1} + P_2R_{1,2} + P_1R_{2,2} \tag{3.27a}$$

$$R_{1,2} = P_1P_4Q_{1,0} + P_4R_{1,1} + P_1R_{4,1} \tag{3.27b}$$

$$R_{1,1} = P_6R_{1,0} + P_1R_{6,0} \tag{3.27c}$$

$$R_{2,2} = P_2P_3Q_{2,0} + P_3R_{2,1} + P_2R_{3,1} \tag{3.27d}$$

$$R_{2,1} = P_5R_{2,0} + P_2R_{5,0} \tag{3.27e}$$

Substituindo o resto $R_{2,1}$ da Eq. (3.27e) na Eq. (3.27d), o resto $R_{1,1}$ da Eq. (3.27c) na Eq. (3.27b), o resto $R_{2,2}$ da Eq. (3.27d) e o resto $R_{1,2}$ da Eq. (3.27b) na Eq. (3.27a), pode-se rescrever E_3 na forma:

$$\begin{aligned} E_3 = {} & P_1P_2Q_{3,1} + P_1P_2P_4Q_{1,0} + P_2P_4P_6R_{1,0} + P_1P_2P_4R_{6,0} \\ & + P_1P_2R_{4,1} + P_1P_2P_3Q_{2,0} + P_1P_3P_5R_{2,0} + P_1P_2P_3R_{5,0} + P_1P_2R_{3,1} \end{aligned} \tag{3.28}$$

Que por sua vez, pode ser reorganizada na forma:

$$\begin{aligned} E_3 = {} & \overbrace{(R_{2,0}P_1P_3P_5 + R_{1,0}P_2P_4P_6)}^{\text{Polinômio translinear encontrado}} \\ & + \underbrace{(P_1P_2[Q_{3,1} + P_4(Q_{1,0} + R_{6,0}) + P_3(Q_{2,0} + R_{5,0}) + R_{3,1} + R_{4,1}])}_{\text{Parte descartada}} \end{aligned} \tag{3.29}$$

Assim, se a parte da Eq. (3.29) referente à "Parte Descartada" for zero, significa que a parte do "Polinômio translinear encontrado" é um polinômio transinear válido. Entretanto, não é necessário guardar nenhum outro resto ou quociente para calcular a "Parte descartada". Basta apenas expandir a parte "Polinômio translinear encontrado" e subtrair de E_3. Se o resultado for zero, implica que a "Parte descartada" também é zero, logo o polinômio translinear é válido.

Assim, o procedimento de Verificação final apenas multiplica P_x por cada elemento da lista de polinômios $\alpha_{y,n}$, e soma a cada resultado de todos os elementos da lista $\alpha_{x,n}$ multiplicados por P_y e subtrai cada resultado de E_r. Cada vez que esta subtração dá zero, o polinômio translinear é guardado no vetor γ. Como exemplo, executa-se a verificação final nos resultados obtidos nas Eqs. (3.22) e (3.25). Esta rotina de verificação final faz parte do fluxografo da Fig. 3.5.

$$\begin{aligned}\alpha_{1,0} =& [\frac{1}{6}, (5I_{in} + 5I_{out} + 3I_0), (I_{out})] \\ \alpha_{2,0} =& [-\frac{1}{6}, (-I_{in} + 5I_{out} + I_0), (I_{in} + I_{out})]\end{aligned}$$

$$\begin{aligned}
&Z = (P_1\alpha_{2,0} + P_2\alpha_{1,0}) \\
&Z = -\frac{1}{6}(6I_{in} + 5I_{out} + 6I_0)(-I_{in} + 5I_{out} + I_0)(I_{in} + I_{out}) \\
&\quad + \frac{1}{6}(5I_{in} + 5I_{out} + 4I_0)(5I_{in} + 5I_{out} + 3I_0)(y) \\
&Z = {I_{in}}^3 + {I_{in}}^2 I_{out} - {I_0}^2 I_{in} + {I_0}^2 I_{out} \\
&K = E_3 - Z \\
&K = ({I_{in}}^3 + {I_{in}}^2 I_{out} - {I_0}^2 I_{in} + {I_0}^2 I_{out}) - ({I_{in}}^3 + {I_{in}}^2 I_{out} - {I_0}^2 I_{in} + {I_0}^2 I_{out}) \\
&K = 0
\end{aligned}$$

Assim, $(P_1\alpha_{2,0} + P_2\alpha_{1,0})$ é um polinômio translinear válido da Eq. (2.21), e deve ser gravado no vetor γ.

3.3.4 Divisão recursiva por pares de polinômios-base, versão otimizada para minimizar a memória utilizada

Pra que o algoritmo apresentado na seção 3.3.2 acima fique seguro em relação ao tamanho de memória utilizado, e que o fundamento de que encontrar os polinômios-base relativos a P_1 e P_2 são problemas separados seja mantido, a *cada* decomposição parcial que seria inserida na lista $\alpha_{x,n}$, todos as decomposições parciais que seriam colocadas na $\alpha_{y,n}$ *são geradas novamente*, o que torna o número de expansões em frações parciais executadas muito maior. Assim o algoritmo da seção anterior é re-implementado na forma do fluxografo da Fig. 3.7. Esta versão do algoritmo executa o sub-algoritmo de "decomposição parcial primária", dado pela Fig. 3.8, que por sua vez executa o sub-algoritmo "decomposição parcial secundária", dado pela Fig. 3.9.

Desta forma, para cada decomposição parcial δ_x, relativa a P_x encontrada, todas as decomposições parciais δ_y relativas a P_y são calculadas, e o procedimento de verificação final é realizado a cada δ_y encontrado, na forma da Eq. (3.30). Como não há criação de listas, não há risco de utilização descontrolada da memória do computador.

$$E_r - (P_x\delta_y + P_y\delta_x) = 0 \tag{3.30}$$

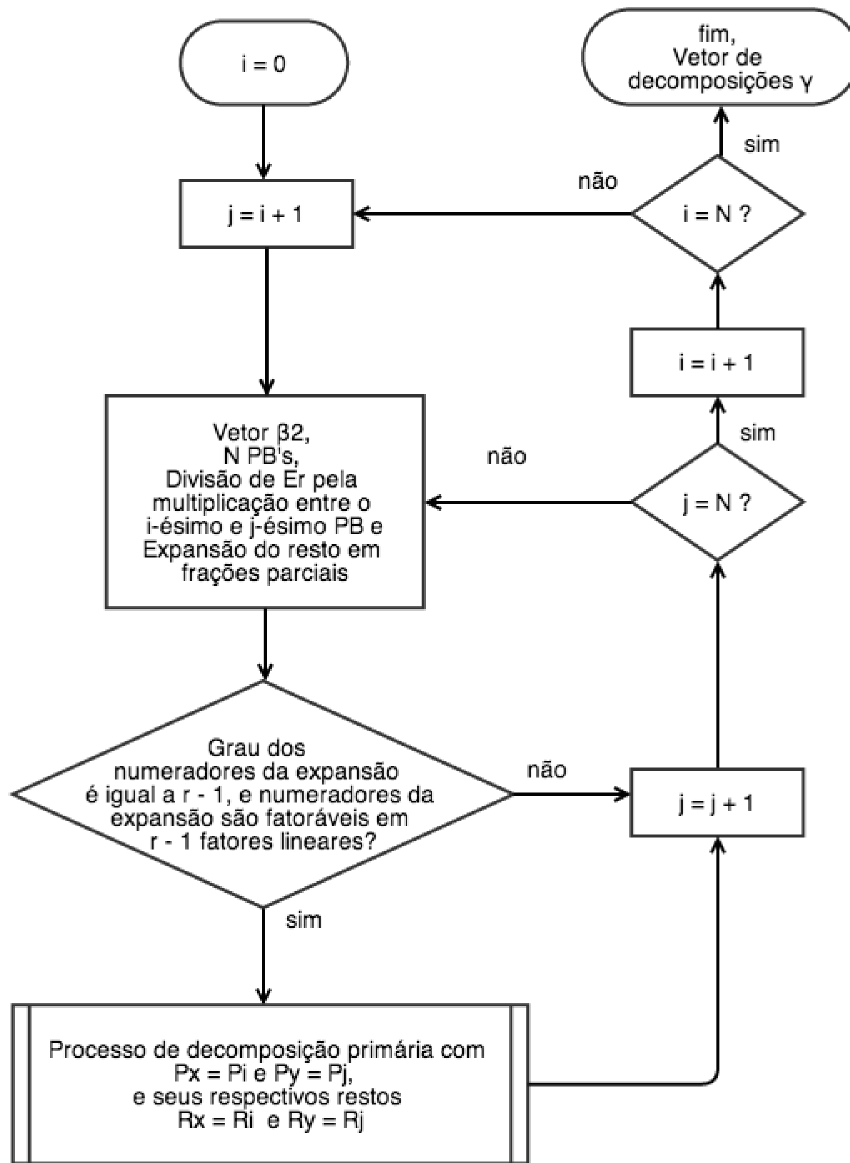


Figura 3.7: Divisão por pares de Polinômios-base e execução de decomposições primárias, versão otimizada para minimizar a memória utilizada. Os índices 'i' e 'j' são utilizados para percorrer o vetor β_2 e gerar pares de polinômios-base.

3.4 Novo Algoritmo de decomposição não-paramétrica

O algoritmo apresentado nesta seção é uma modificação do apresentado na seção 3.3, e é o foco desta dissertação. Assim como no algoritmo de Mulder et. al. [23], este trabalho tem por objetivo encontrar um ou mais polinômios translineares na forma da Eq.(2.6) para um dado polinômio modo-corrente[4] utilizando o processo de decomposição não-paramétrica. A principal mudança introduzida por este trabalho é a tentativa de obter mais soluções que o algoritmo de Mulder et. al., e secundariamente, tentar evitar o aumento de computações, ou seja, uma perda de eficiência

[4]Transformado a partir de um polinômio adimensional, através dos métodos apresentados nas seções, 2.4.2 e 2.6.1

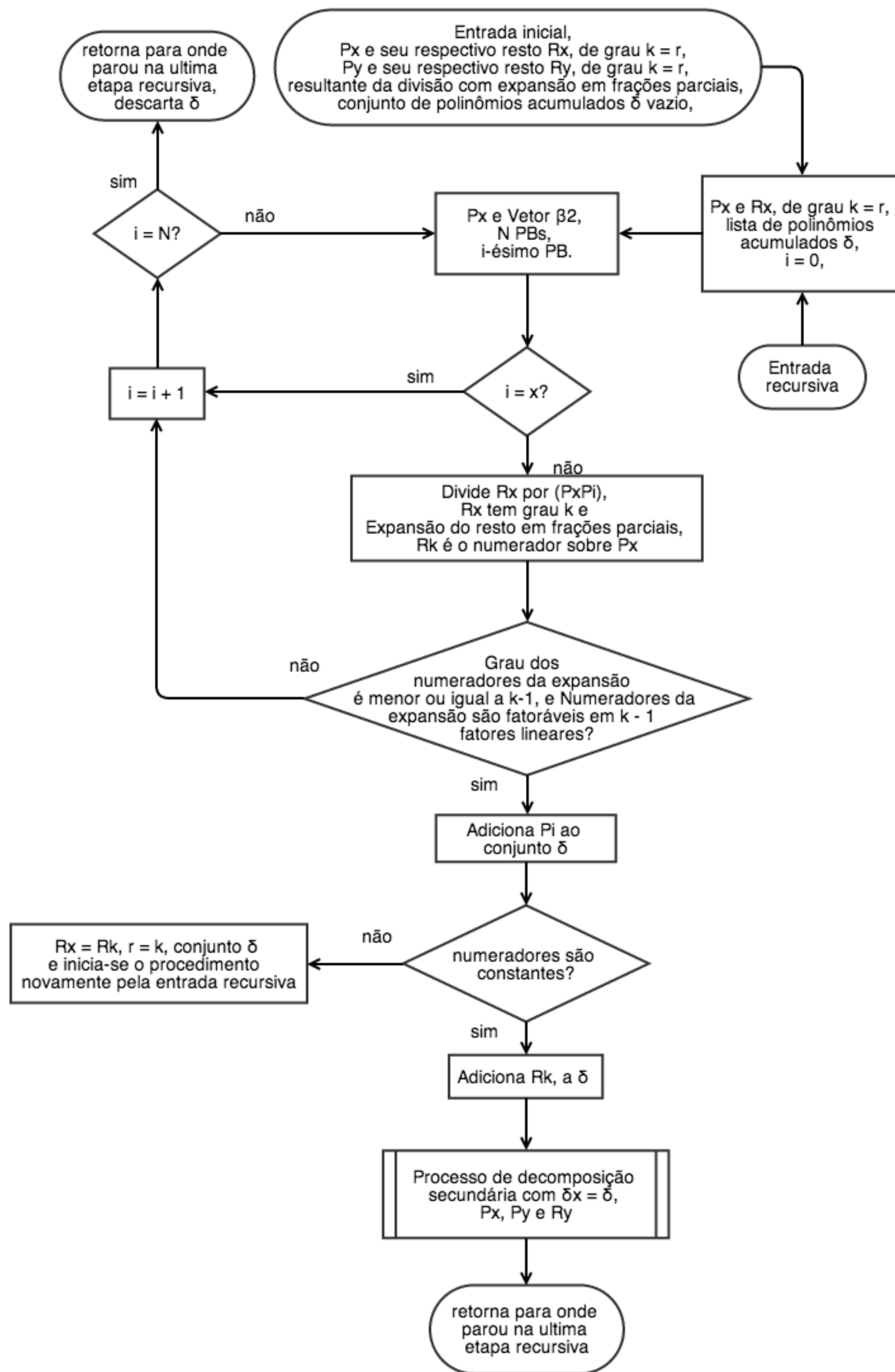


Figura 3.8: Sub-rotina de geração de decomposições primarias δ_x. O índice 'i' é utilizado para percorrer o vetor β_2.

em relação ao algoritmo original. Este algoritmo tem as mesmas etapas estruturais do outro, como pode ser visto na Fig. 3.2, entretanto, adicionou-se a etapa de Eliminação de polinômios translineares repetidos. A Fig. 3.10 mostra a estrutura deste Algoritmo, e cada etapa é descrita nas seções deste capítulo.

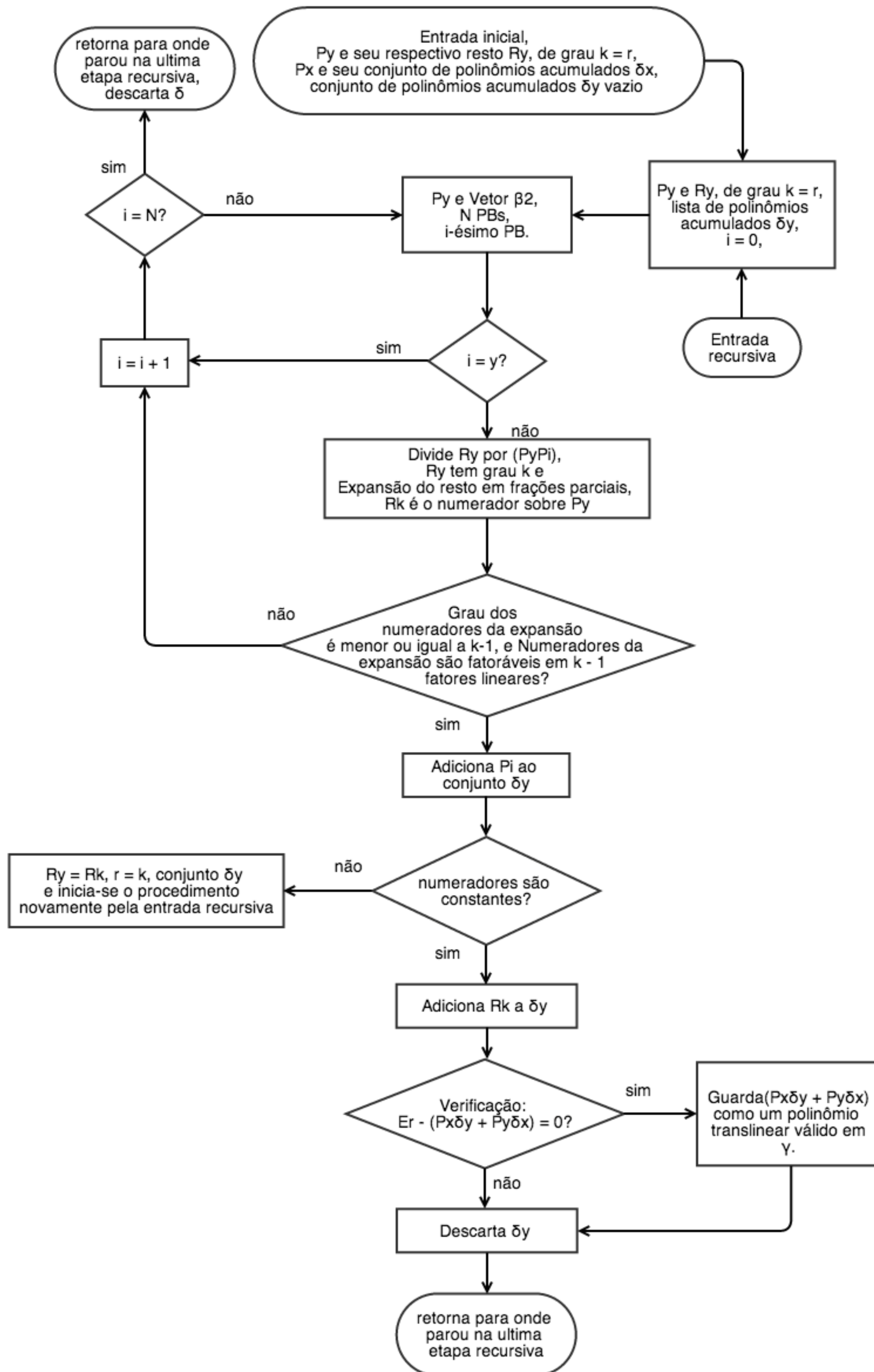


Figura 3.9: Sub-rotina de geração de decomposições secundarias δ_y. O índice 'i' é utilizado para percorrer o vetor β_2.

Deste ponto em diante, o termo "algoritmo" refere-se ao algoritmo desenvolvido neste trabalho, e o termo "algoritmo original" refere-se ao algoritmo de Mulder et. al..

Figura 3.10: Estrutura do novo algoritmo de decomposição não-paramétrica

3.4.1 Geração do Vetor de Polinômios-Base

O algoritmo utiliza o vetor de polinômios-base β_0, definido pela Eq. (2.13), mas este recebe o nome τ_0 nesta seção ($\tau_0 = \beta_0$). As etapas de redução do vetor τ_0 executadas pelo algoritmo podem ser vistas na Fig. 3.11. A etapa de "Eliminação de polinômios-base estritamente negativos" gera o vetor τ_1 a partir do vetor τ_0; a etapa de "Eliminação de polinômios-base redundantes" gera o vetor τ_2 a partir do vetor τ_1; a etapa de "Divisão portares de polinômios base e expansão em frações parciais" gera o vetor τ_3 a partir do vetor τ_2; e a etapa de "Re-contagem de polinômios-base" gera o vetor τ_4 a partir do vetor τ_3.Cada etapa é descrita nas subseções a seguir.

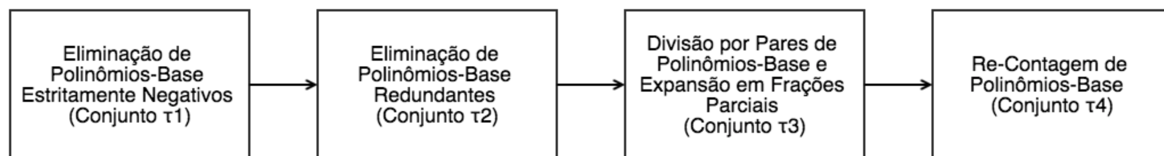


Figura 3.11: Geração do Vetor de Polinômios-Base

3.4.1.1 Eliminação da etapa de divisão por um Polinômio-Base e Fatoração do resto

Apesar da etapa de "Divisão por um polinômo-base e fatoração do resto", apresentada na seção 3.3.1.2, ser efetiva na redução do vetor de polinômios-base, o autor do algoritmo original não especificou qual algoritmo de fatoração polinomial multivariável foi utilizado, e o desenvolvimento destes algoritmos é uma área muito ampla, com muitos trabalhos acadêmicos, praticamente um ramo da matemática simbólica em si. Não há um algoritmo definitivo e absoluto que resolva este problema [42–47].

Assim, como a fatoração polinomial é um passo desejável, mas não necessário para se obter todas os polinômios translineares de um polinômio modo-corrente de entrada, e como o objetivo deste trabalho é encontrar o máximo número de polinômios translineares possível, decidiu-se evitar o risco de ter um polinômio-base adequado excluído por uma fatoração plinomial inadequada, e preferiu-se deixar a escolha de um algoritmo de fatoração adequado, bem como sua implementação ou integração, como propostas para trabalhos futuros.

3.4.1.2 Eliminação de Polinômios-Base Estritamente Negativos

O argumento de que a faixa de variação de todas as variáveis de corrente devem ser conhecidas previamente, como visto na seção 3.3.1.1, nem sempre se aplica. Pode ser que uma restrição de tamanho de circuito, por exemplo, seja mais importante que a faixa de operação das respectivas correntes; ou pode ser que seja mais importante que *haja* pelo menos um polinômio translinear equivalente, e que os limites das correntes possam ser definidos em função do polinômio translinear encontrado, e não o contrário. Esta abordagem oferece ao projetista mais flexibilidade e maior probabilidade de encontrar um polinômio translinear realizável para seu projeto.

Esta modificação também proporciona que este algoritmo possa ser utilizado para encontrar os polinômios translineares do conjunto de polinômios modo-corrente gerados por uma decomposição paramétrica, tornando-o mais poderoso em relação ao algoritmo original.

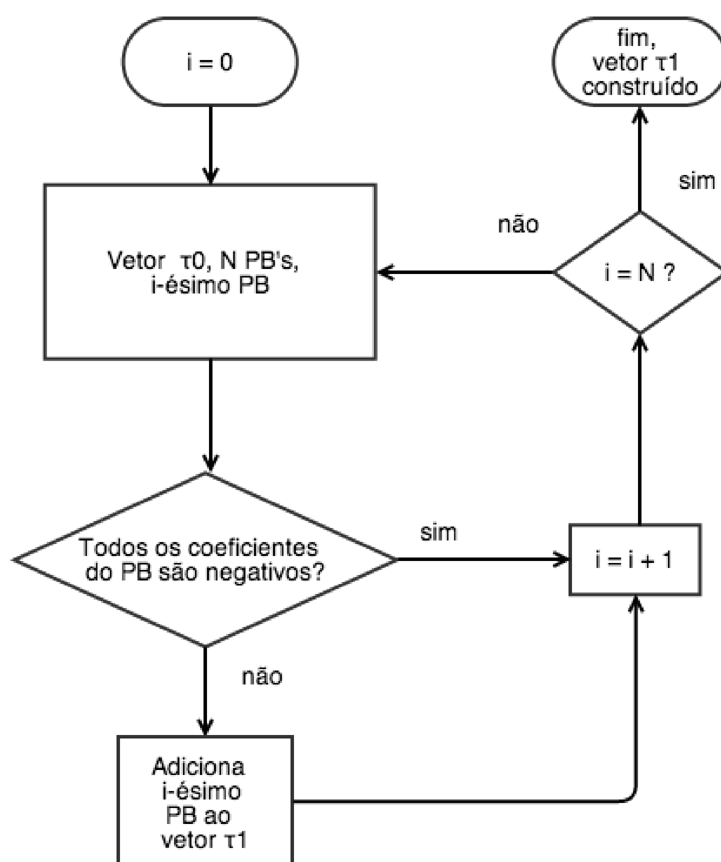


Figura 3.12: Algoritmo de eliminação de polinômios-base estritamente negativos. O índice "i" é utilizado para percorrer o vetor τ_0

Portanto, diferentemente do método apresentado na seção 3.3.1, escolheu-se eliminar os polinômios-base *estritamente negativos* ao invés de manter apenas os estritamente positivos, constituindo assim, o vetor reduzido τ_1. O algoritmo que realiza esta operação pode ser visto na Figura 3.12. O número de polinômios-base excluídos por este procedimento é determinístico, dado pelo número de coeficientes negativos mais um, devido ao zero, elevado ao número de variáveis, ou seja, $(N+1)^v$.

Subtraindo este número de N_{τ_0}, o número de elementos de τ_1, N_{τ_1} é dado pela Eq. (3.31).

$$N_{\tau_1} = N_{\tau_0} - (N+1)^v \tag{3.31}$$

Como exemplo, aplicando esta redução ao conjunto τ_0 dado pela Tabela 2.1, obtém-se o conjunto τ_1 mostrado na Tabela 3.1, cujo número de elementos é $N_{\tau_1} = (3)^2 - (2)^2 = 5$, exatamente o número de elementos, todos estritamente não-negativos, na tabela.

Tabela 3.1: Exemplo de eliminação de Polinômis-Base estritamente negativos

x	y	PB
-1	1	$(-x+y)$
0	1	(y)
1	-1	$(x-y)$
1	0	(x)
1	1	$(x+y)$

Para que haja ainda uma maior flexibilidade nos polinômios translineares que deseja-se encontrar, permite-se definir diferentes limites superiores e inferiores para os coeficientes, na forma $[-N_{inf}, \ldots, +N_{sup}]$ ao invés de usar coeficientes entre $[-N, \ldots, +N]$, como visto na seção 2.5. Assim, os cálculos de N_{τ_0} e N_{τ_1} podem ser re-escritos, respectivamente, nas Eqs. (3.32) e (3.33)

$$N_{\tau_0} = (N_{inf} + N_{sup} + 1)^v \tag{3.32}$$

$$N_{\tau_1} = N_{\tau_0} - (N_{inf} + 1)^v \tag{3.33}$$

3.4.1.3 Eliminação de Polinômios-Base Redundantes

Suponha que para um polinômio qualquer, haja dois polinômios translineares dados pelas Eqs. (3.34) e (3.35).

$$\lambda_1 P_1 P_3 P_5 - \lambda_2 P_2 P_4 P_6 = 0 \tag{3.34}$$

$$\lambda_1 P_1 P_3 P_7 - \lambda_2 P_2 P_4 P_8 = 0 \tag{3.35}$$

Se $P_7 = k_1 P_5$, e $P_8 = k_2 P_6$, k_1 e k_2 constantes, então o polinômio translinear dado pela Eq. (3.35) pode ser rescrito na Eq. (3.36), que é composta pelos mesmos polinômios-base que a Eq. (3.34), portanto os polinômios translineares são equivalentes do ponto de vista do algoritmo.

$$k_1 \lambda_1 P_1 P_3 P_5 - k_2 \lambda_2 P_2 P_4 P_6 = 0 \tag{3.36}$$

Assim, se houver um ou mais polinômios-base que sejam combinação linear de outro qualquer, basta que apenas um deles faça parte do vetor de polinômios-base para que todos os polinômios

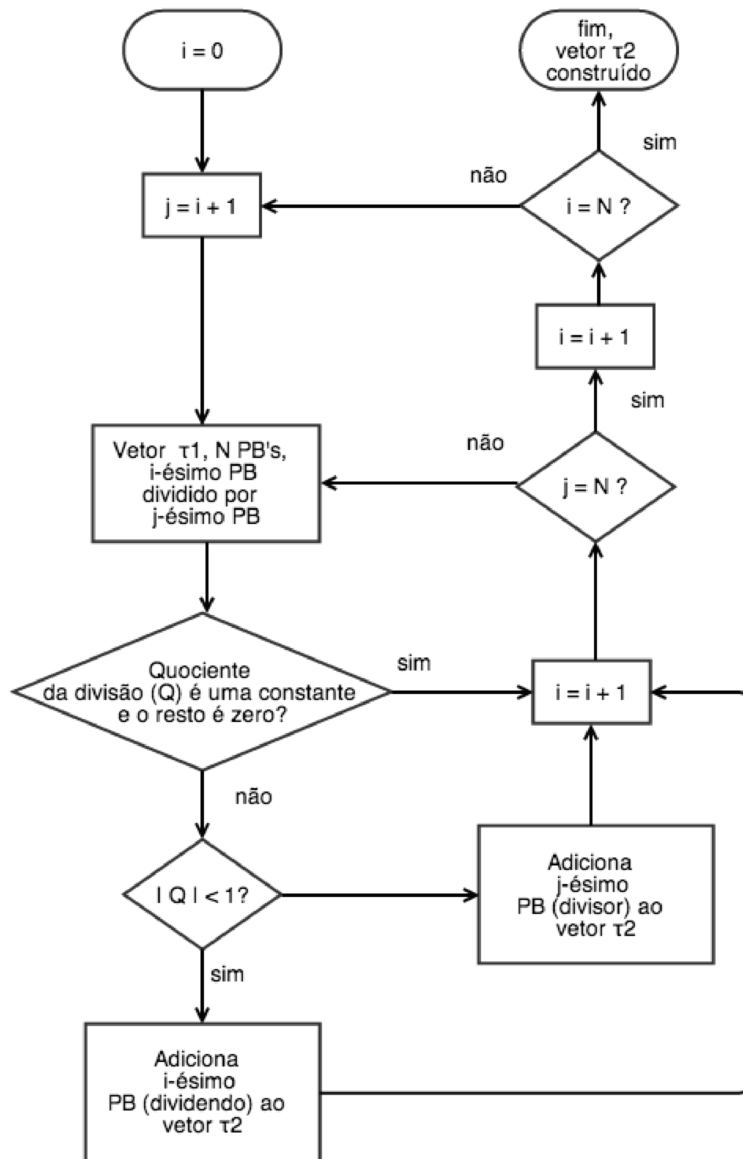


Figura 3.13: Algoritmo de eliminação de polinômios-base (PB) redundantes. O índice "i" refere-se a um polinômio-base de τ_1 que será dividido por outro de índice "j" do mesmo vetor.

translineares sejam encontrados. Para remover estes polinômios-base, faz-se uma divisão simples entre todos os polinômios-base do vetor τ_1, removendo um deles quando o quociente da divisão é uma constante e o resto é zero. A decisão entre remover o polinômio dividendo ou o polinômio divisor depende da constante encontrada no quociente. Se o seu módulo é menor que 1, significa que os coeficientes do divisor são maiores que os do dividendo, portanto elimina-se o polinômio-base divisor. Caso contrário, elimina-se o polinômio-base do dividendo. Isso é feito para que os polinômios-base utilizados tenham coeficientes menores, para melhorar a legibilidade dos resultados. Ao final do processo, o vetor de polinômios-base reduzido τ_2 é gerado. Este procedimento é ilustrado na Figura (3.13)

3.4.1.4 Divisão por Pares de Polinômios-Base e Expansão em Frações Parciais

Como visto na seção 3.3.2.1, a divisão do polinômio de entrada E_r por um par de polinômios-base, P_x e P_y, permite verificar se este par pode fazer parte de um polinômio translinear equivalente. Entretanto, o autor impõe a condição de que os numeradores resultantes da expansão em frações parciais, R_x e R_y devem ter grau $r-1$ e serem fatoráveis em $r-1$ fatores lineares. Como visto na seção 3.4.1.1, utilizar este procedimento com um algoritmo de fatoração polinomial multivariável inadequado pode causar a perda de polinômios-base que poderiam fazer parte de um polinômio translinear válido. Todavia, decidiu-se excluir apenas este requisito de fatorabilidade dos restos, dado pela Eq. (3.7), e o procedimento continua sendo válido para encontrar possíveis pares de polinonômios-base que possam fazer parte de um polinômio translinear válido.

Uma mudança fundamental deste algoritmo é utilizar esta etapa para construir um vetor de *pares de polinômios base*, chamado τ_3, onde cada elemento é um par de polinômios-base com seus respectivos restos gerados pela expansão em frações parciais bem sucedida. Este procedimento é mostrado na Eq. (3.37), e o elemento do vetor τ_3 gerado é $((P_x, R_x), (P_y, R_y))$. Como os elementos deste vetor são os pontos de partida para encontrar as decomposições, chama-se o vetor τ_3 de "vetor sementes".

$$\frac{E_r}{P_x P_y} = Q + \frac{R_x}{P_x} + \frac{R_y}{P_y} \tag{3.37}$$

O fundamento deste procedimento é que, se um polinômio-base faz parte de um polinômio translinear, então ele necessariamente está associado a pelo menos um polinômio-base que faz parte de um mesmo polinômio translinear, opostamente conectado, dado pela Eq. (2.12). Assim, como os polinômios-base devem estar sempre associados a pelo menos um outro, é mais eficiente tratá-los aos pares dentro do algoritmo. Com isso, a afirmação feita pelo autor original na seção 3.3.2.2 de que encontrar os polinômios opostamente conectados a P_x e P_y são problemas separados leva apenas a uma possível perda de eficiência, calculando de forma desnecessária decomposições parciais que com certeza não formarão um polinômio translinear válido, pois fundamentalmente, os problemas não são independentes.

Relembrando que $P_{x,n}$, $Q_{x,n}$, e $R_{x,n}$ são respectivamente: Polinômios-base, Quocientes, e Restos de índice x e grau n, o processo de divisão recursiva mostrado na seção 3.3.2.2 pode ganhar eficiência se, a cada etapa recursiva, um dado resto $R_{x,n}$ for dividido por $P_x P_1$, e $R_{y,n}$ for dividido por $P_y P_2$, como mostrado nas Eqs. (3.38) a (3.39), *apenas* se houver um elemento $((P_1, R_{1,n}), (P_2, R_{2,n}))$ pertencente a τ_3.

$$\frac{R_{x,r-1}}{P_x P_1} = Q_{x,r-2} + \frac{R_{x,r-2}}{P_x} + \frac{R_{1,r-2}}{P_1} \tag{3.38}$$

$$\frac{R_{y,r-1}}{P_y P_2} = Q_{y,r-2} + \frac{R_{y,r-2}}{P_y} + \frac{R_{2,r-2}}{P_2} \tag{3.39}$$

Este processo divide E_r por todos os possíveis pares de polinômios-base do vetor τ_2 para gerar o vetor de pares τ_3, e este procedimento é ilustrado na Fig. 3.14

Como exemplo, é utilizado um vetor de polinômios-base τ_2, gerado a partir dos procedimentos anteriores para o polinômio de entrada E_r dado pelo lado esquerdo da Eq. (2.21). Este vetor

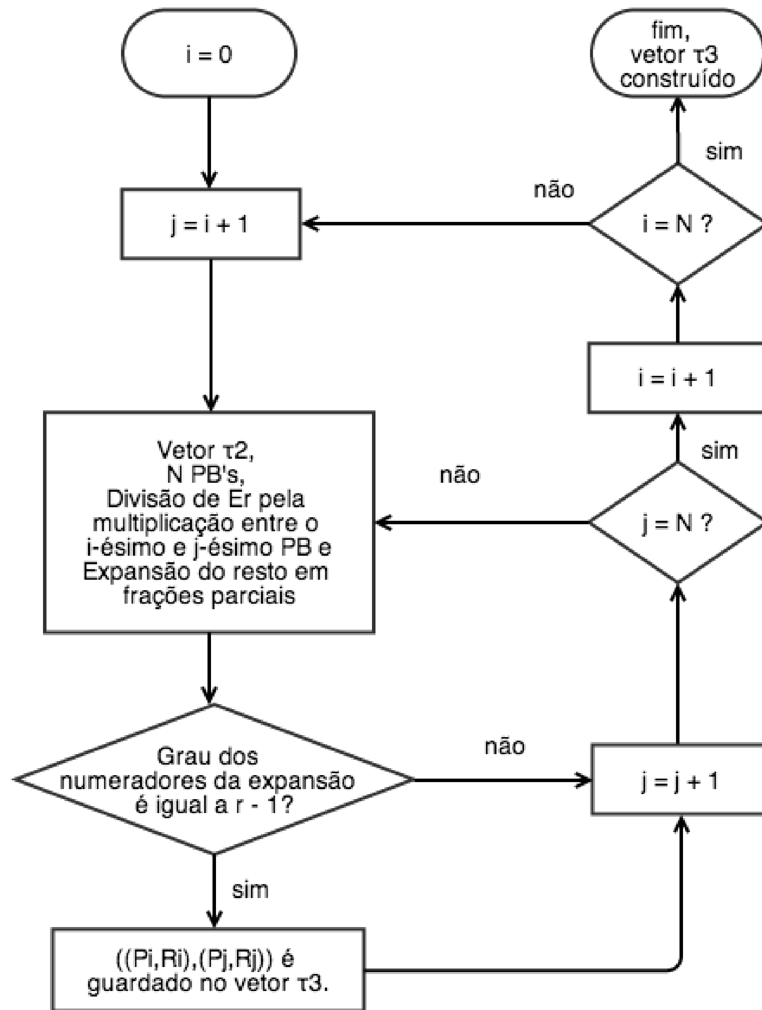


Figura 3.14: Algoritmo de geração do vetor τ_3 de pares de polinômios-base (PB). O índice "i" refere-se a um polinômio-base de τ_2 que, junto outro de índice "j" do mesmo vetor, dividirão o polinômio modo-corrente E_r.

Tabela 3.2: Exemplo de vetor τ_2

elemento	polinômio
P_1	$I_0 + I_{in}$
P_2	$I_0 - I_{in}$
P_3	I_{out}
P_4	I_{in}
P_5	I_{out}
P_6	$I_0 + I_{in} + I_{out}$

τ_2 é mostrado na Tabela 3.2, e alguns polinômios-base gerados foram excluídos para melhorar a legibilidade. O vetor τ_3 gerado a partir deste vetor τ_2 é mostrado na Tabela 3.3. São mostrados exemplos da obtenção dos dois primeiros elementos deste vetor τ_3: $((P_1, P_2), (R_{1,2}, R_{2,2}))$, e

$((P_1, P_4), (R_{1,2}, R_{4,2}))$nas Eqs. (3.40) e (3.41).

$$E_r = {I_{in}}^3 + {I_{in}}^2 I_{out} - {I_0}^2 I_{in} + {I_0}^2 I_{out}$$

Fazendo para $((P_1, P_2), (R_{1,2}, R_{2,2}))$:

$$\begin{aligned} \frac{E_r}{P_1 P_2} &= Q_{3,1} + \frac{R_{1,2}}{P_1} + \frac{R_{2,2}}{P_2} \\ \frac{{I_{in}}^3 + {I_{in}}^2 I_{out} - {I_0}^2 I_{in} + {I_0}^2 I_{out}}{(I_0 + I_{in})(I_0 - I_{in})} &= Q_{3,1} + \frac{I_0 I_{out}}{(I_0 + I_{in})} + \frac{I_0 I_{out}}{(I_0 - I_{in})} \end{aligned} \tag{3.40}$$

obtém-se o primeiro elemento da Tabela 3.3: $(((I_0 + I_{in}), (I_0 - I_{in})), (I_0 I_{out}, I_0 I_{out}))$.

Fazendo agora para $((P_1, P_4), (R_{1,2}, R_{4,2}))$:

$$\begin{aligned} \frac{E_r}{P_1 P_4} &= Q_{3,1} + \frac{R_{1,2}}{P_1} + \frac{R_{4,2}}{P_4} \\ \frac{{I_{in}}^3 + {I_{in}}^2 I_{out} - {I_0}^2 I_{in} + {I_0}^2 I_{out}}{(I_0 + I_{in}) I_{in}} &= Q_{3,1} + \frac{I_0 I_{out}}{(I_0 + I_{in})} + \frac{I_0^2}{I_{in}} \end{aligned} \tag{3.41}$$

obtém-se o segundo elemento da Tabela 3.3: $(((I_0 + I_{in}), (I_{in})), (I_0 I_{out}, I_0^2))$.

Apesar de aparecer $Q_{3,1}$ e $R_{1,2}$ em ambos exemplos, não há problema, pois os índices são particulares de cada elemento de τ_3 gerado.

3.4.1.5 Recontagem de polinômios-Base

O processo de construção do vetor τ_3 compensa, de certa forma, a ausência do processo de fatoração do resto da divisão por um polinômio-base. Após constituído, o vetor τ_3 é percorrido, e cada polinômio-base que faz parte de algum elemento é copiado no vetor τ_4. Assim, os polinômios base de τ_2 que com certeza *não* fazem parte de polinômio translinear algum são eliminados pelo processo de construção do vetor τ_3. Este vetor de polinômios-base reduzido é utilizado para comparação entre algoritmos, e o seu procedimento de construção é mostrado na Fig. 3.15

3.4.2 Divisão Recursiva por Pares de Polinômios-Base e verificação final

Baseado na premissa de que polinômios-base devem ser tratados aos pares, modificou-se o procedimento da seção 3.3.2.2. Ao invés de criar listas $\alpha_{x,n}$ independentes para depois testar se a combinação entre seus conjuntos σ formam um polinômio translinear válido, constrói-se simultaneamente as os conjuntos σ de P_x e P_y a partir de um elemento $((P_x, R_x), (P_y, R_y))$ de τ_3. Assim, estes conjuntos σ são chamados aqui de σ_x e σ_y.

Estes conjuntos σ_x e σ_y são construídos fazendo-se a divisão recursiva dos polinômios do elemento $((P_x, R_x), (P_y, R_y))$ com os polinômios-base de outros elementos de τ_3. Caso uma das divisões (e fatoração do resto) simultâneas venha a falhar, a divisão é refeita invertendo os polinômios-base do elemento testado. Por exemplo, considere os elementos de τ_3 $((P_1, R_1), (P_2, R_2))$ e $((P_3, R_3), (P_4, R_4))$. Realizando o procedimento de divisão e fatoração do resto simultâneos,

Tabela 3.3: Vetor τ_3 gerado a partir de vetor τ_2 reduzido

elemento de $\tau 3$	P_x P_y	$R_{x,2}$ $R_{y,2}$
P_1 P_2	$I_0 + I_{in}$ $I_0 - I_{in}$	$I_0 I_{out}$ $I_0 I_{out}$
P_1 P_4	$I_0 + I_{in}$ I_{in}	$I_0 I_{out}$ ${I_0}^2$
P_1 P_6	$I_0 + I_{in}$ $I_0 + I_{in} + I_{out}$	$I_0 I_{out}$ $-{I_{in}}^2 + 2I_{out}I_0 + {I_0}^2$
P_2 P_3	$I_0 - I_{in}$ I_{out}	$I_0 I_{out}$ ${I_{in}}^2 - {I_0}^2$
P_2 P_4	$I_0 - I_{in}$ I_x	$2I_0 I_{out}$ $I_0 I_{out}$
P_2 P_5	$I_0 - I_{in}$ I_0	$I_0 I_{out}$ ${I_{in}}^2$
P_3 P_4	I_{out} I_{in}	${I_{in}}^2 - {I_0}^2$ ${I_0}^2$
P_5 P_6	I_0 $I_0 + I_{in} + I_{out}$	${I_{in}}^2$ $-{I_{in}}^2 + 2I_{out}I_0 + {I_0}^2$
P_3 P_6	I_{out} $I_0 + I_{in} + I_{out}$	${I_{in}}^2 - {I_0}^2$ $-{I_{in}}^2 + 2I_{out}I_0 + {I_0}^2$
P_4 P_5	I_{in} I_0	${I_0}^2$ ${I_{in}}^2$
P_5 P_6	I_0 $I_0 + I_{in} + I_{out}$	${I_{in}}^2$ $-{I_{in}}^2 + 2I_{out}I_0 + {I_0}^2$

obtém-se:

$$\frac{R_{1,2}}{P_1 P_4} = Q + \frac{R_{1a,2}}{P_1} + \frac{R_{4b,2}}{P_4} \tag{3.42}$$

$$\frac{R_{2,2}}{P_2 P_3} = Q + \frac{R_{2,1}}{P_2} + \frac{R_{3,1}}{P_3} \tag{3.43}$$

Como $R_{1a,2}$ e $R_{4b,2}$ possuem o mesmo grau de $R_{1,2}$, a divisão simultânea falhou, então deve-se testar a divisão combinando P_1 com P_3 e P_2 com P_4, conforme as equações abaixo:

$$\frac{R_{1,2}}{P_1 P_3} = Q + \frac{R_{1,1}}{P_1} + \frac{R_{3,1}}{P_3} \tag{3.44}$$

$$\frac{R_{2,2}}{P_2 P_4} = Q + \frac{R_{2,1}}{P_2} + \frac{R_{4,1}}{P_4} \tag{3.45}$$

Neste caso, as Eqs. (3.44) e (3.45) mostram um procedimento de divisão e expansão em frações parciais bem-sucedido. Ao final do processo recursivo, aplica-se o procedimento de "Verificação

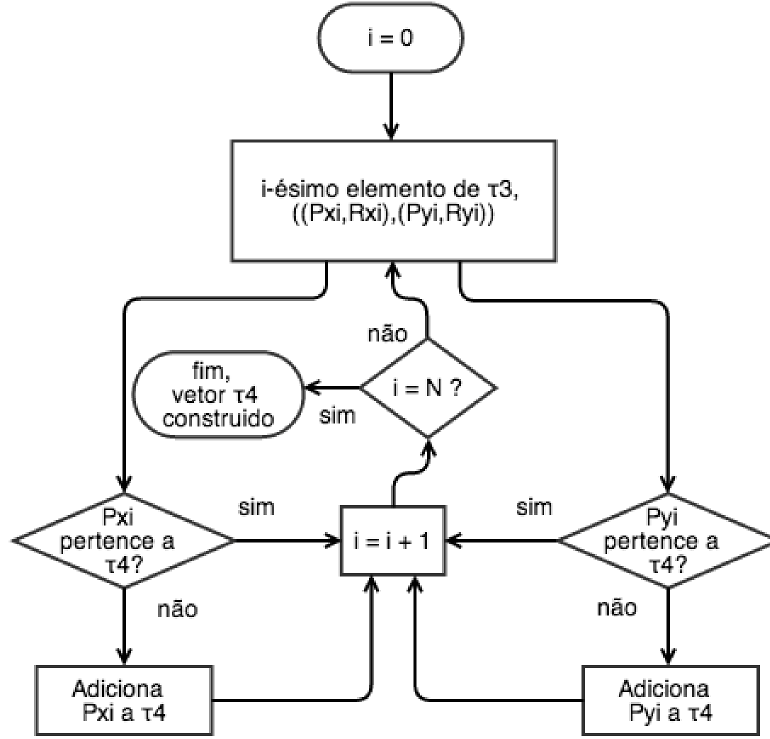


Figura 3.15: Algoritmo de geração do vetor τ_4 de polinômios-base (PB). O índice "i" refere-se a um elemento do vetor τ_3

Final" descrito na seção 3.3.3 aos conjuntos σ_x e σ_y gerados, reduzindo este procedimento a um único teste dado pelo conjunto da Eq. (3.46), considerando que E_r foi o polinômio modo-corrente utilizado para gerar o vetor τ_3:

$$Z = (P_x\sigma_y + P_y\sigma_x) \tag{3.46a}$$

$$K = E_r - Z \tag{3.46b}$$

$$\text{Se } K = 0; \qquad \text{então } P_x\sigma_y + P_y\sigma_x \text{é um polinômio translinear válido de } E_r \tag{3.46c}$$

Como exemplo, este algoritmo é executado nas Eqs.(3.47)-(3.50), com o polinômio de entrada E_r sendo o lado esquerdo da Eq. (2.21), e seu respectivo o vetor reduzido τ_3 é mostrado na Tabela 3.4. Este vetor reduzido foi retirado da Tabela 3.3 para uma melhor legibilidade. Tomando $((P_1, R_{1,2}), (P_2, R_{2,2}))$ como par inicial, os conjuntos σ_1 e σ_2 contém apenas os restos iniciais $R_{1,2}$ e $R_{2,2}$:

$$\sigma_1 = [I_0 I_{out}]$$
$$\sigma_2 = [I_0 I_{out}]$$

Combinando $((P_1, R_{1,2}), (P_2, R_{2,2}))$ com $((P_4, R_{4,2}), (P_3, R_{3,2}))$ obtém-se:

$$\frac{R_{1,2}}{P_1 P_4} = Q + \frac{R_{1,1}}{P_1} + \frac{R_{4,1}}{P_4}$$
$$\frac{I_0 I_{out}}{(I_0 + I_{in}) I_{in}} = \frac{-I_{out}}{(I_0 + I_{in})} + \frac{I_{out}}{I_{in}} \tag{3.47}$$

Tabela 3.4: Vetor reduzido τ_3

elemento de $\tau 3$	P_x P_y	$R_{x,2}$ $R_{y,2}$
P_1	$I_0 + I_{in}$	$I_0 I_{out}$
P_2	$I_0 - I_{in}$	$I_0 I_{out}$
P_3	I_{out}	${I_{in}}^2 - {I_0}^2$
P_4	I_{in}	${I_0}^2$
P_5	I_0	${I_{in}}^2$
P_6	$I_0 + I_{in} + I_{out}$	$-{I_{in}}^2 + 2I_{out}I_0 + {I_0}^2$

e, simultaneamente:

$$\begin{aligned} \frac{R_{2,2}}{P_2 P_3} &= Q + \frac{R_{2,1}}{P_2} + \frac{R_{3,1}}{P_3} \\ \frac{I_0 I_{out}}{(I_0 - I_{in})I_{out}} &= 1 + \frac{I_{in}}{(I_0 - I_{in})} + \frac{0}{I_{out}} \end{aligned} \tag{3.48}$$

Como ambas divisões e expansões em frações parciais dos restos foram bem-sucedidas, adiciona-se P_4 a σ_1 e P_3 a σ_2, e substitui-se $R_{1,2}$ e $R_{2,2}$ por $R_{1,1}$ e $R_{2,1}$, respectivamente:

$$\begin{aligned} \sigma_1 &= [(-I_{out}), (I_{in})] \\ \sigma_2 &= [(I_{in}), (I_{out})] \end{aligned}$$

Combinando $((P_1, R_{1,1}), (P_2, R_{2,1}))$ com $((P_6, R_{6,2}), (P_5, R_{5,2}))$ obtém-se:

$$\begin{aligned} \frac{R_{1,1}}{P_1 P_6} &= Q + \frac{R_{1,0}}{P_1} + \frac{R_{6,1}}{P_6} \\ \frac{-I_{out}}{(I_0 + I_{in})(I_0 + I_{in} + I_{out})} &= \frac{-1}{(I_0 + I_{in})} + \frac{1}{(I_0 + I_{in} + I_{out})} \end{aligned} \tag{3.49}$$

e, simultaneamente:

$$\begin{aligned} \frac{R_{2,1}}{P_2 P_5} &= Q + \frac{R_{2,0}}{P_2} + \frac{R_{5,1}}{P_5} \\ \frac{I_{in}}{(I_0 - I_{in})I_0} &= 1 + \frac{1}{(I_0 - I_{in})} + \frac{-1}{I_0} \end{aligned} \tag{3.50}$$

Como ambas divisões e expansões em frações parciais dos restos foram bem-sucedidas, adiciona-se P_6 a σ_1 e P_5 a σ_2, e substitui-se $R_{1,1}$ e $R_{2,1}$ por $R_{1,0}$ e $R_{2,0}$, respectivamente:

$$\begin{aligned} \sigma_1 &= [(-1), (I_{in}), (I_0 + I_{in} + I_{out})] \\ \sigma_2 &= [(1), (I_{out}), (I_0)] \end{aligned}$$

Como os restos são constantes, $\lambda_1 = -1$, and $\lambda_2 = 1$, e o processo recursivo é interrompido. O

polinômio translinear encontrado é dado pela Eq. (3.51), e o único teste de verificação final fica:

$$\begin{aligned} E_r &= {I_{in}}^3 + {I_{in}}^2 I_{out} - {I_0}^2 I_{in} + {I_0}^2 I_{out} \\ Z &= (P_1\sigma_2 + P_2\sigma_1) \\ Z &= (I_0 + I_{in})I_{out}I_0 - (I_0 - I_{in})I_{in}(I_0 + I_{in} + I_{out}) \\ K &= E_r - Z \\ K &= 0 \end{aligned} \tag{3.51}$$

Se o polinômio translinear é válido, os polinômios do conjunto σ_x e σ_y são ordenados segundo seus índices dentro do vetor τ_4. O polinômio translinear encontrado é testado na forma da seção 3.5, e se não for redundante, é gravado no vetor γ.

O diagrama deste algoritmo pode ser visto na Fig. 3.16

3.5 Eliminação de polinômios translineares redundantes

Como o vetor τ_3 indica qual polinômio pode ser combinado com outro, ao encontrar os polinômios translineares a partir do vetor τ_3 da Tabela 3.3, certamente serão gerados três polinômios translineares equivalentes ao polinômio translinear dado pela Eq. (3.51):

$$\begin{aligned} Z_1 &= P_1P_3P_5 + P_2P_4P_6 = (I_0 + I_{in})I_{out}I_0 - (I_0 - I_{in})I_{in}(I_0 + I_{in} + I_{out}) \\ Z_2 &= P_1P_3P_5 + P_4P_2P_6 = (I_0 + I_{in})I_{out}I_0 - I_{in}(I_0 - I_{in})(I_0 + I_{in} + I_{out}) \\ Z_3 &= P_1P_3P_5 + P_6P_2P_4 = (I_0 + I_{in})I_{out}I_0 - (I_0 + I_{in} + I_{out})(I_0 - I_{in})I_{in} \end{aligned}$$

E não seria possível detectar estas redundâncias de forma preemptiva a não ser comparando os polinômios translineares encontrados com os já existentes no vetor γ. Desta forma, cada polinômio translinear válido encontrado é comparado com todos os outros já gravados, e é descartado caso um equivalente já exista. Assim, o vetor γ conterá apenas polinômios translineares únicos. O diagrama deste procedimento pode ser visto na Fig. 3.17

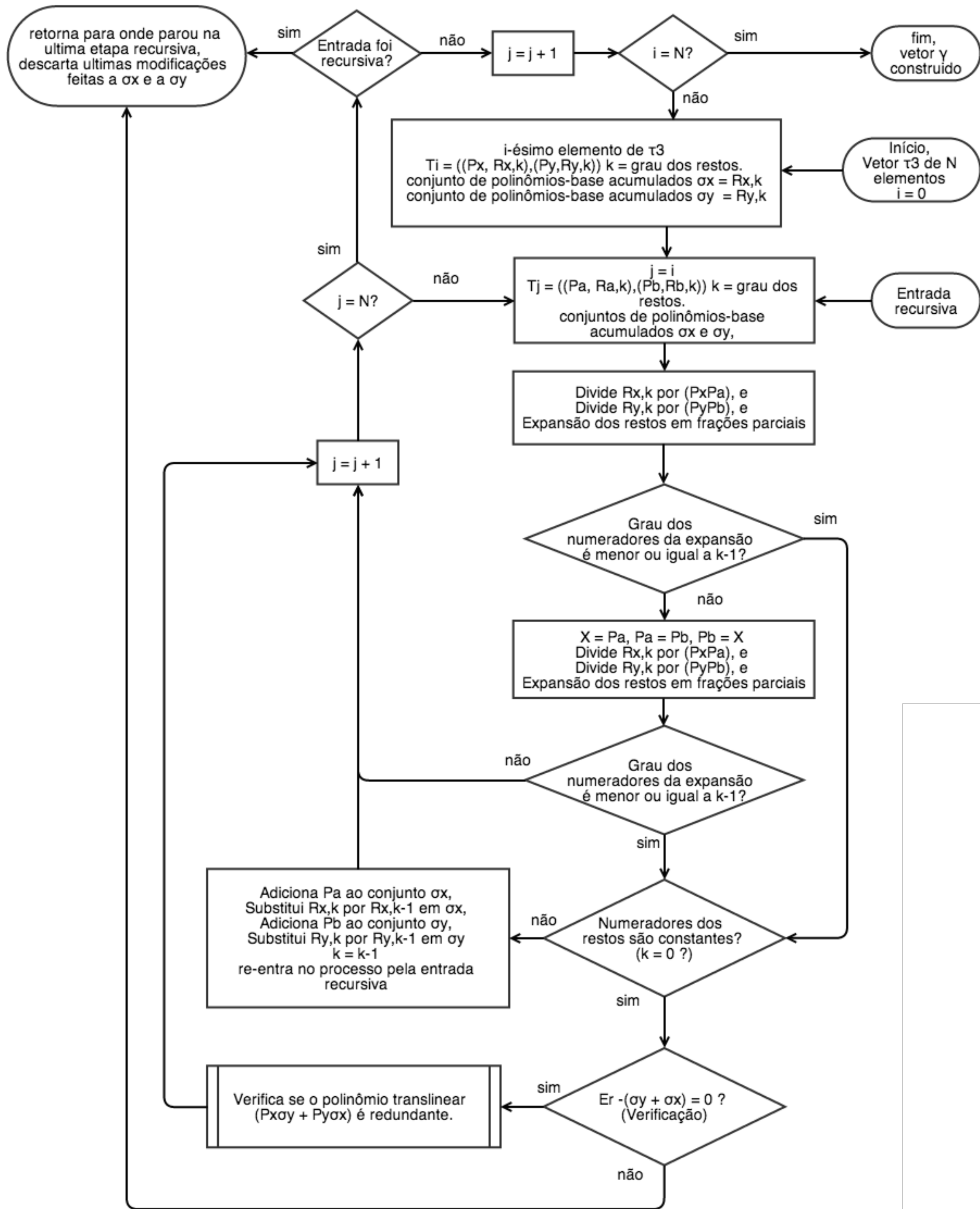


Figura 3.16: Divisão Recursiva por Pares de Polinômios Simultâneos. T_i e T_j referem-se a elementos do vetor τ_3, e "i" e "j" são índices utilizados para percorrer este vetor. Os índices "a" e "b" servem para diferenciar os polinômios-base e restos de T_j dos polinômios-base e restos de T_i, que utilizam os índices "x" e "y".

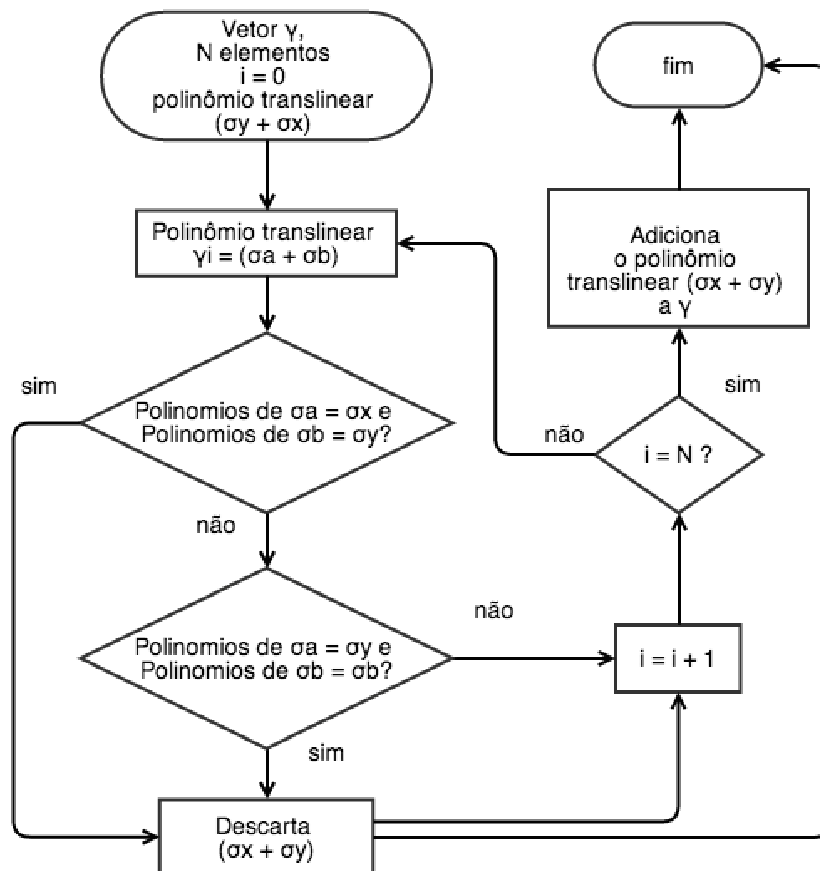


Figura 3.17: Eliminação de polinômios translineares redundantes. O índice "i" refere-se a um polinômio translinear do vetor γ. Subentende-se que σ_x e σ_y já incluem os seus respectivos polinômios de origem.

Capítulo 4

Procedimento de validação e comparação entre algoritmos

4.1 Introdução

Neste capítulo são apresentadas as formas de se avaliar o novo algoritmo quanto a eficácia e eficiência, quando comparado ao algoritmo original, este último em suas versões otimizadas quanto à tempo de execução e uso de memória. Ambos algoritmos foram implementados em linguagem "C", compilados e executados em um computador com processador Intel© core i7, 8GB de memória RAM. Não convém incluir o algoritmo de [25], descrito na seção 2.8.1, nestas análises comparativas, pois não é um algoritmo de decomposição translinear. Os resultados da aplicação da metodologia descrita neste capítulo são apresentados no Capítulo 5.

4.2 Eficácia do algoritmo implementado

A eficácia do algoritmo é a capacidade dele de encontrar todas os polinômios translineares dentro do intervalo de coeficientes $[-N_{inf}, \ldots, N_{sup}]$. Assim, cada polinômio de entrada eleito para este teste é submetido a ambos algoritmos, e se os mesmos polinômios translineares forem encontrados, então o algoritmo está validado até que alguém encontre algum polinômio de entrada no qual o número de polinômios translineares encontrados seja diferente. Como o novo algoritmo utiliza estratégias diferentes para gerar o vetor de polinômios-base em relação ao algoritmo original, utiliza-se a estratégia de geração de polinômios-base do algoritmo original, ou seja o vetor τ_4 da seção 3.4.1.5, junto com a estratégia de decomposição translinear não-paramétrica do algoritmo original descrita na seção 3.3.2 para fazer esta comparação. Assim, ambas estratégias de combinação devem gerar os mesmos polinômios translineares para um mesmo polinômio modo-corrente de entrada. Como não implementou-se a função de fatoração de polinômios multivariáveis, não há como utilizar a estratégia da seção 3.3.1 para comparar os resultados da decomposição de um polinômio qualquer.

4.3 Medida de Eficiência entre algoritmos

Como visto na seção 3.2, um algoritmo ineficiente poderia levar meses, ainda que utilizando um computador poderoso, para encontrar polinômios translineares a partir de polinômios modo-corrente relativamente simples. Assim, os dois algoritmos são comparados em relação ao tempo gasto para encontrar todas os polinômios translineares e em relação ao número de operações de expansão em frações parciais executadas, pois é a rotina mais intensamente executada em ambos algoritmos e de maior consumo de ciclos de *clock*.

Como os algoritmos não são determinísticos, é possível determinar apenas a quantidade de expansões em frações parciais executadas no caso de nenhuma das expansões terem sucesso, ou seja, apenas se nenhuma função é executada de forma recursiva. Para calcular o número N_{novo} de expansões em frações parciais no pior caso para o algoritmo deste trabalho, utiliza-se o número N_{τ_3} de elementos do vetor τ_3. Como cada elemento é combinado com outro de duas formas diferentes, exceto quando um elemento combina com si próprio, o cálculo de $N_{original}$, utilizando a fórmula de arranjo com repetição, é dado pela Eq. (4.1)

$$N_{novo} = {N_{\tau_3}}^2 \tag{4.1}$$

Já no caso do algoritmo original, considerando a sua versão otimizada para menor tempo de execução, como o número de elementos no vetor τ_3 indica quantas vezes a etapa de "divisão por um par de polinômios-base e expansão do resto em frações parciais", mostrada na seção 3.3.2.1 é executada, multiplica-se N_{τ_3} pelo número de elementos do vetor reduzido τ_4, N_{τ_4}, para encontrar o número de expansões em frações parciais executadas no pior caso N_{orig}, dado pela Eq. (4.2).

$$N_{orig} = N_{\tau_3} N_{\tau_4} \tag{4.2}$$

4.3.1 Geração de vetores de polinômios extremamente reduzidos e nova medida de eficiência

Tendo sido gerados todos os polinômios translineares com qualquer um dos algoritmos, percorre-se o vetor de polinômios translineares γ e copia-se em um vetor τ_5 todos os polinômios-base que de fato fizeram parte de algum polinômio translinear. Este procedimento está ilustrado na Fig. 4.1.

Assim, pode-se fazer um novo teste de eficiência dos algoritmos de forma mais significativa com este vetor extremamente reduzido, pois este vetor com certeza é igual ou menor que um vetor que fosse submetido ao processo de "divisão por um polinômio-base e fatoração do resto" visto na seção 3.3.1.2. Um vetor τ_6 de pares de polinômios é gerado a partir de τ_5, conforme o procedimento da seção 3.4.1.4.

O novo algoritmo é re-executado tendo o vetor τ_6 como entrada e o algoritmo original é re-executado tendo o vetor τ_5 como entrada, e os resultados são novamente comparados quanto à eficiência e eficácia.

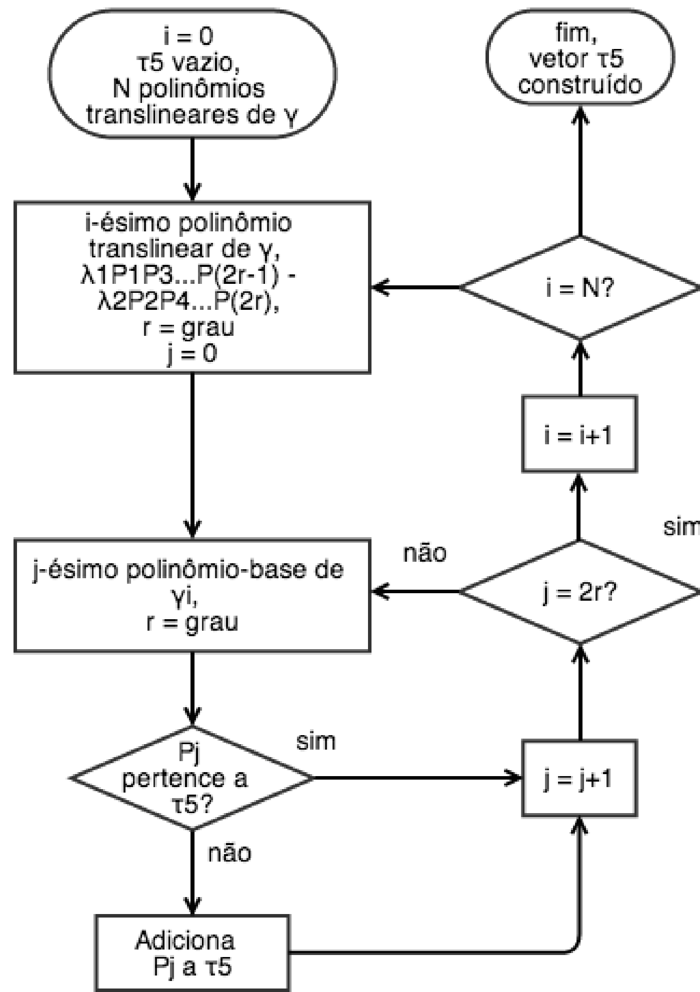


Figura 4.1: Geração do vetor de polinômios-base extremamente reduzido τ_5. O índice "i" refere-se a um polinômio translinear dentro do vetor γ, e o índice "j" refere-se a um polinômio-base pertencente ao polinômio translinear γ_i.

4.4 Polinômios utilizados para testar o algoritmo

4.4.1 Polinômios aleatórios

Para testar tanto a eficiência quanto a eficácia do algoritmo, são construídos polinômios de entrada aleatórios, formados a partir de uma equação na forma da 2.12. Como a ferramenta implementada realiza um pré-processamento do polinômio de entrada, expandindo-o, simplificando e reordenando os monômios resultantes, o algoritmo deve encontrar pelo menos a equação de entrada na forma em que foi passada à ferramenta.

Assim, escolheu-se os polinômios da Tabela 4.1, buscando-se variar o grau r e o número de variáveis v. O valor dos coeficientes variou apenas entre $[-1, \ldots, +1]$ para simplificar a análise.

Tabela 4.1: Vetor de Polinômios de Entrada para avaliação dos algoritmos

índice	grau r	número de variáveis v	Polinômio de entrada
a	2	2	$(x+y)x-(x-y)y$
b	2	3	$(x+y-z)(x+z)-z^2$
c	2	4	$(x-k)(x-y+z)-(k+z)(x+y-z)$
d	3	2	$(x+y)^2(x-y)-xy^2$
e	3	3	$x^2(x+y)-z^2(x-y)$
f	3	4	$(x+k)(x-y)(k+z)-x^2(y+k-z)$
g	4	2	$(-x+y)^2y^2-x^3(y+x)$
h	4	3	$(x+z)^2(-y+z)z-y^2(y-z)^2$
i	4	4	$x^3(-y+k+z)-(x-k)(x-y)(x+y-k)^2$
j	5	2	$x^2(x-y)^3-(x+y)^2y^3$
k	5	3	$x^2y^2(x-y-z)-(y+z)^2(x-z)^2z$
l	5	4	$(x-k)(x+y-z)(x-k-y+z)^2-y^3(x+k-z)^2$

4.4.2 Realizações de circuitos Translineares publicados

Além desta avaliação de funcionalidade, o argumento apresentado na seção 3.4.1.2, de que a estratégia de redução do vetor de polinômios base deste algoritmo pode gerar um numero maior de polinômios translineares, também é testado, para inclusive auxiliar no processo de decomposição translinear paramétrica. Assim, a comparação no sentido de gerar mais polinômios translineares é feita a partir de algumas realizações de circuitos translineares publicadas [26, 27, 48, 49], onde o algoritmo desenvolvido é utilizado para encontrar os mesmos polinômios translineares a partir dos polinômios modo-corrente de entrada utilizados nos referidos trabalhos.

Capítulo 5

Resultados

5.1 Introdução

Neste capítulo são mostrados os resultados da aplicação dos procedimentos mostrados no capítulo 4.

5.2 Comparativo de esforço computacional entre algoritmos

Os polinômios da Tabela 4.1 são utilizados para comparar o tempo em segundos em que os algoritmos encontram todos os polinômios translineares bem como o número de operações de expansões em frações parciais realizadas. Resultados de tempo de execução de zero segundos indicam que o aplicativo levou menos que 1 segundo para concluir sua execução. Para o algoritmo original, utiliza-se o vetor de polinômios-base τ_4, e para o novo algoritmo utiliza-se o vetor de pares de polinômios-base τ_3. O resultados para os coeficientes $[-1, \ldots, +1]$ são mostrados na Tabela 5.1, sendo os melhores resultados destacados em negrito.

Pode-se notar que o algoritmo original, em sua versão otimizada para menor tempo de execução realizaria menos operações no pior caso para todos os polinômios de entrada. Entretanto, este não é um indicador seguro para determinar qual algoritmo realizará a menor quantidade de expansões em frações parciais e nem do tempo necessário para encontrar todos os polinômios translineares. Exceto pelos resultados referentes ao polinômio de entrada "i", o novo algoritmo executou em tempo menor ou igual para os demais polinômios.

Certamente, esta desconexão entre o número de operações de expansão em frações parciais executadas e o tempo de execução refere-se ao fato de o procedimento de "Verificação Final" do algoritmo original otimizado para menor tempo de execução, descrito na seção 3.3.3, efetua muitas verificações ao combinar decomposições parciais da lista $\alpha_{x,n}$ com as decomposições parciais da lista $\alpha_{y,n}$, um comportamento de certa forma parecido com o do algoritmo trivial. Assim, nesta versão do algoritmo original, o número de verificações pode ser tão grande que produz um impacto relevante no tempo de execução. No entanto, demorar apenas alguns segundos de tempo de execução pode

Tabela 5.1: Comparativo de eficiência computacional entre algoritmos com coeficientes $[-1, \ldots, +1]$.

E_r	N_{τ_4}	N_{τ_3}	N_{orig}	N_{novo}	tempo original (exec.)	tempo original (mem.)	tempo novo	ops. original (exec.)	ops. original (mem.)	ops. novo
a	4	6	**24**	36	0s	0s	0s	66	120	64
b	7	12	**84**	144	0s	0s	0s	228	348	**171**
c	11	24	**264**	576	0s	0s	0s	920	1.430	**687**
d	4	6	**24**	36	0s	0s	0s	**174**	381	261
e	11	30	**330**	900	0s	0s	0s	**3.625**	9.884	5.298
f	13	25	**325**	625	0s	0s	0s	2.207	3.970	**1.112**
g	4	6	**24**	36	0s	1s	0s	**354**	906	695
h	13	27	**351**	729	3s	3s	**1s**	**12.594**	100.717	22.589
i	14	27	**378**	729	**0s**	1s	1s	5.185	10.523	**2.042**
j	4	6	**24**	36	0s	0s	0s	**624**	1.776	1510
k	12	30	**360**	900	5s	2s	**1s**	**21.514**	70.939	53.492
l	16	35	**560**	1.225	5s	3s	**0s**	**19.149**	37.898	19.488

ser considerado um ótimo resultado para todos os algoritmos, de forma que a escolha entre um ou outro é irrelevante para o intervalo de coeficientes $[-1, \ldots, +1]$.

Procurando obter uma comparação mais significativa, repetiu-se o procedimento para o intervalo de coeficientes $[-3, \ldots, +3]$. Os resultados para o algoritmo original em suas duas versões utilizando o vetor τ_4, e para o novo algoritmo utilizando τ_3 estão descritos na Tabela 5.2. Os números N_{novo} e N_{orig} não são mostrados, pois este indicador mostrou-se irrelevante de acordo com a análise feita nos parágrafos anteriores.

Os resultados da Tabela 5.2 mostram que não há uma relação direta entre os números N_{τ_3} e N_{τ_4} no tempo necessário para encontrar todas as soluções.

Os resultados do algoritmo original, em sua versão otimizada para minimizar tempo de execução, para os polinômios "h", "k"e "l", não puderam ser obtidos porque o aplicativo, ao atingir cerca de 3 GB de memória, foi finalizado pelo sistema operacional. Isso deve-se ao fato de que não há como prever o quão grande as listas $\alpha_{x,n}$ e $\alpha_{y,n}$ podem ficar. Entretanto, tanto o novo algoritmo quanto o algoritmo original em sua versão otimizada para minimizar memória ocupam memória constante durante a execução das operações recursivas, de cerca de 3,6 MB.

Pode-se verificar que o novo algoritmo supera o a versão otimizada para minimizar memória do algoritmo original, tanto no aspecto do tempo de execução quanto no aspecto de número de operações realizadas. É confirmada a afirmação feita na seção 3.4.1.4, de que polinômios-base que fazem parte de um polinômio translinear sempre estão associados a outro, portanto é mais eficiente e seguro testá-los aos pares.

Tabela 5.2: Comparativo de eficiência computacional entre algoritmos com coeficientes $[-3, \ldots, +3]$

E_r	N_{τ_4}	N_{τ_3}	tempo original (exec.)	tempo original (mem.)	tempo novo	ops. original (exec.)	ops. original (mem.)	ops. novo
a	16	120	1s	1s	1s	**6.990**	54.240	28.104
b	49	353	2s	1s	**1s**	**60.000**	205.968	134.786
c	79	614	**5s**	21s	7s	405.606	605.514	**181.740**
d	16	120	55s	38s	**36s**	**60.990**	2.044.620	2.020.000
e	71	600	**45s**	81s	56s	**498.303**	9.751.122	6.703.563
f	73	520	14s	42s	**6s**	**229.045**	2.128.472	309.485
g	16	120	55m 37s	**13m 20s**	44m 15s	**366.990**	34.451.688	109.561.780
h	145	768	—	3h 9m 3s	**2h 9m 47s**	—	58.156.468	**39.980.584**
i	77	539	53m 26s	14m 5s	**1m 4s**	**163.054**	54.381.588	4.005.024
j	16	120	**1d 10h 54m 48s**	1d 11h 9m 12s	1d 22h 53m 48s	**1.742.102**	355.320.098	464.949.583
k	72	599	—	54m 8s	**37m 12s**	—	322.719.203	**221.858.419**
l	96	683	—	4h 45m	**20m 43s**	—	1.110.270.604	**81.767.756**

Finalmente, para os coeficientes$[-3, \ldots, +3]$, efetua-se novamente a comparação utilizando agora os vetores extremamente reduzidos de polinômios-base obtidos segundo o procedimento da seção 4.3.1, utilizando τ_5 no algoritmo original em sua primeira versão e no trivial, e τ_6 no novo algoritmo. Os resultados são descritos na Tabela 5.3.

O resultado da Tabela 5.3 mostra o impacto que a redução no número de polinômios-base causa na velocidade com que os algoritmos encontram todos os polinômios translineares. Assim, encontrar um algoritmo adequado de fatoração polinomial multivariável e implementá-lo no aplicativo é a principal proposta de trabalhos futuros.

5.3 Aplicação do algoritmo desenvolvido em trabalhos publicados

Nesta seção, o novo algoritmo é aplicado em polinômios utilizados em realizações de circuitos translineares já publicados, nos quais os autores omitem os detalhes do processo utilizado na obtenção dos polinômios translineares a partir de polinômios modo-corrente.

5.3.1 Oscilador de segunda ordem

Em [49], um oscilador de segunda ordem é desenvolvido utilizando síntese translinear, e seu esquemático pode ser visto na Fig. 5.1. Aplicando o método de decomposição paramétrica, utilizando as correntes I_{y1} e I_{y2} como parâmetros, e a relação entre as funções $k(I_i, I_0)$ e $h(I_i, I_0)$ dada

Tabela 5.3: Comparativo de eficiência computacional entre algoritmos com coeficientes $[-3, \ldots, +3]$ e vetores de polinômios-base extremamente reduzidos

E_r	N_{τ_5}	N_{τ_6}	tempo original (exec.)	tempo original (mem.)	tempo novo	ops. original (exec.)	ops. original (mem.)	ops. novo
a	16	120	1s	1s	**0s**	**6.990**	54.240	28.104
b	36	197	1s	1s	1s	**28.280**	88.410	44.367
c	67	464	4s	7s	4s	**125.334**	402.006	235.190
d	13	78	15s	10s	**9s**	**26.076**	581.178	541.817
e	9	24	0s	1s	0s	**2.373**	5.619	3.298
f	5	6	0s	0s	0s	193	171	**88**
g	5	10	0s	0s	0s	**1.108**	4.924	4.464
h	11	41	15s	11s	**8s**	**21.329**	319.204	242.546
i	5	6	0s	0s	0s	245	**161**	165
j	4	6	0s	0s	0s	**624**	1.880	1.510
k	6	13	0s	8s	0s	**2.706**	15.875	9.783
l	6	8	0s	0s	0s	**80**	995	572

pelas Eqs. (5.3) e (5.4), o autor parte dos conjuntos das Eqs. (5.1) e (5.2), e obtém os polinômios translineares dados pelas Eqs. (5.5) e (5.6), utilizando um método manual não-especificado de [27].

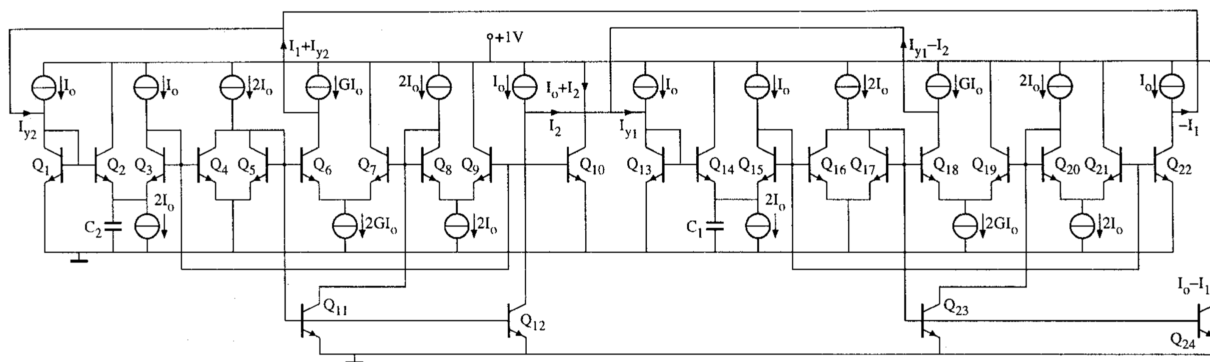


Figura 5.1: Circuito oscilador de segunda ordem [49]

$$(I_1 + I_0)I_{cap1} = I_0^2 h(I_1, I_0) + I_0 I_2 \tag{5.1a}$$
$$I_{y1} = I_2 + I_0 k(I_1, I_0) \tag{5.1b}$$

$$(I_2 + I_0)I_{cap2} = I_0^2 h(I_2, I_0) - I_0 I_1 \tag{5.2a}$$
$$I_{y2} = -I_1 + I_0 k(I_2, I_0) \tag{5.2b}$$

onde:

$$k(I_i, I_0) = \frac{I_i}{I_0} + h(I_i, I_0) \tag{5.3}$$

$$k(I_i, I_0) = \frac{2I_G I_i}{I_0^2 + I_i^2} \tag{5.4}$$

$$I_0(I_0 + I_{y1}) - (I_1 + I_0)(I_{cap1} + I_0) = 0 \tag{5.5a}$$

$$(I_0 + I_1)^2(I_G + I_2 - I_{y1}) - (I_0 - I_1)^2(I_G - I_2 + I_{y1}) = 0 \tag{5.5b}$$

$$I_0(I_0 + I_{y2}) - (I_2 + I_0)(I_{cap2} + I_0) = 0 \tag{5.6a}$$

$$(I_0 + I_2)^2(I_G + I_1 + I_{y2}) - (I_0 - I_2)^2(I_G - I_1 - I_{y2}) = 0 \tag{5.6b}$$

Para mostrar que a estratégia de se eliminar polinômios-base *estritamente negativos*, mostrada na seção 3.4.1.2, torna o algoritmo de decomposição não-paramétrica útil no auxílio da decomposição paramétrica, primeiramente desenvolve-se as Eqs. (5.1) e (5.2) utilizando as relações dadas pelas Eqs. (5.3) e (5.4) de forma a eliminar os fatores $k(I_i, I_0)$ e $h(I_i, I_0)$, e obtém-se o conjunto de polinômios modo-corrente dados pelas Eqs. (5.7) e (5.8)

$$(I_1 + I_0)I_{cap1} - I_0(I_{y1} - I_1) = 0 \tag{5.7a}$$

$$(I_0^2 + I_1^2)(I_{y1} - I_2) - 2I_G I_0 I_1 = 0 \tag{5.7b}$$

$$(I_2 + I_0)I_{cap2} - I_0(I_{y2} - I_2) = 0 \tag{5.8a}$$

$$(I_0^2 + I_2^2)(I_{y2} + I_1) - 2I_G I_0 I_2 = 0 \tag{5.8b}$$

Como as Eqs. (5.7a) e (5.8a) são análogas, os polinômios translineares encontrados de uma delas pode ser atribuída à outra, bastando apenas substituir os índices. Os polinômios translineares obtidos com o novo algoritmo, utilizando o intervalo de coeficientes $[-1, \ldots, +1]$, são dados pelo conjunto das Eqs. (5.9)

$$(I_1 + I_0)(-I_{cap1} + I_{y1}) - I_1(I_0 + I_{y1}) \tag{5.9a}$$

$$(I_1 + I_0)(-I_1 - I_{cap1} + I_{y1}) - I_1(-I_1 + I_{y1}) \tag{5.9b}$$

$$(I_1 + I_0)(I_0 + I_{cap1}) - I_0(I_0 + I_{y1}) = 0 \tag{5.9c}$$

$$(I_1 + I_0)(I_{cap1}) - I_0(-I_1 + I_{y1}) \tag{5.9d}$$

$$I_1(I_0 + I_{cap1}) - I_0(-I_{cap1} + I_{y1}) \tag{5.9e}$$

$$I_1(I_{cap1}) - I_0(-I_1 - I_{cap1} + I_{y1}) \tag{5.9f}$$

$$(I_0 + I_{cap1})(-I_1 + I_{y1}) - (I_0 + I_{y1})(I_{cap1}) \tag{5.9g}$$

$$(I_0 + I_{cap1})(-I_1 - I_{cap1} + I_{y1}) - I_{cap1}(-I_{cap1} + I_{y1}) \tag{5.9h}$$

$$(I_0 + I_{y1})(-I_1 - I_{cap1} + I_{y1}) - (-I_{cap1} + I_{y1})(-I_1 + I_{y1}) \tag{5.9i}$$

Foram encontradas nove polinômios translineares, dentre eles a Eq. (5.9c), que é equivalente ao polinômio translinear da Eq. (5.5a), encontrado manualmente pelo autor. Aplicando o algoritmo no polinômio dado pela Eq. (5.7b), novamente utilizando o intervalo de coeficientes $[-1, \ldots, +1]$, obtém-se os polinômios translineares dados pelo conjunto da Eq. (5.10)

$$(I_0 + I_1)^2(-I_{y1} + I_2) - 2I_0I_1(-I_{y1} + I_2 - I_G) \tag{5.10a}$$

$$(I_0 + I_1)^2(-I_{y1} + I_2 + I_G) - (I_{y1} - I_2 + I_G)(I_0 - I_1)^2 \tag{5.10b}$$

$$2I_0I_1(-I_{y1} + I_2 + I_G) - (I_{y1} - I_2)(I_0 - I_1)^2 \tag{5.10c}$$

Foram encontrados três polinômios translineares, e a Eq.(5.10b) é equivalente à Eq. (5.5b). Aplicando o algoritmo no polinômio dado pela Eq. (5.8b), obtém-se polinômios translineares dados pelo conjunto da Eq. (5.11).

$$(I_0 + I_2)^2(I_1 + I_{y2}) - 2I_0I_2(I_1 + I_{y2} + I_G) \tag{5.11a}$$

$$(I_0 + I_2)^2(-I_1 - I_{y2} + I_G) - (I_1 + I_{y2} + I_G)(I_0 - I_2)^2 \tag{5.11b}$$

$$2I_0I_2(-I_1 - I_{y2} + I_G) - (I_1 + I_{y2})(I_0 - I_2)^2 \tag{5.11c}$$

Novamente, foram encontrados três polinômios translineares, mas nenhum deles é equivalente ao polinômio translinear encontrado pelo autor dado pela Eq. (5.6b). Para investigar a razão de o polinômio translinear do autor não ter sido encontrado pelo algoritmo, expandiu-se e simplificou-se a Eq. (5.6b), e o resultado é dado pela Eq. (5.12)

$$(I_0^2 + I_2^2)(I_{y2} + I_1) + 2I_GI_0I_2 = 0 \tag{5.12}$$

Percebe-se que o sinal do termo $2I_GI_0I_2$ é diferente nas Eqs. (5.12) e (5.8b), enquanto que deveriam ser iguais. Assim, pode-se deduzir que o autor cometeu um erro de sinal ao realizar o procedimento de decomposição paramétrica de forma manual, e fica comprovado que o novo algoritmo é eficaz no auxílio do procedimento de decomposição paramétrica, podendo prevenir este tipo de erro.

5.3.2 Conversor RMS-DC

Em [26], foi desenvolvido um circuito translinear de conversão RMS-DC. A síntese deste circuito foi utilizada no exemplo da seção 2.7. Entretanto, para verificar a eficácia do algoritmo, ele é aplicado à Eq. (5.13a) com coeficientes $[-1, \ldots, +1]$, e os polinômios translineares gerados são mostradas no conjunto da Eq.(5.13)

$$I_{cap}I_{out}^2 + I_0I_{out}^2 - I_0I_{in}^2 \tag{5.13a}$$

$$I_{cap}I_{in}^2 - (I_{cap} + I_0)(I_{out} + I_{in})(I_{in} - I_{out}) \tag{5.13b}$$

$$I_0I_{in}^2 - (I_{cap} + I_0)I_{out}^2 = 0 \tag{5.13c}$$

$$I_{cap}I_{out}^2 - I_0(I_{out} + I_{in})(I_{in} - I_{out}) \tag{5.13d}$$

Assim, o polinômio translinear da Eq. (5.13a) é equivalente ao da Eq.(5.13c), e demonstra-se que mais opções de polinômios translineares são obtidas.

5.3.3 Circuito da função Módulo

Em [27], é desenvolvido um circuito translinear que realiza a função módulo. Este circuito também foi utilizado na seção 2.7.1 como exemplo de síntese de circuitos translineares. Aplicando o algoritmo desenvolvido na função módulo escolhida dada pela Eq. (5.14a), e utilizando o intervalo de coeficientes $[-1, \ldots, +1]$, os polinômios translineares gerados são dados pelo conjunto das Eq. (5.14):

$$I_y^2 - I_x^2 \tag{5.14a}$$
$$2I_x(-I_y + I_x) - (-I_y + I_x)^2 \tag{5.14b}$$
$$2I_x(I_y + I_x) - (I_y + I_x)^2 \tag{5.14c}$$
$$I_x(I_y + I_x) - I_y(I_y + I_x) \tag{5.14d}$$
$$I_y^2 - I_x^2 \tag{5.14e}$$
$$I_y(-I_y + I_x)I_x(I_y - I_x) \tag{5.14f}$$

Como I_x pode assumir valores negativos, e $I_y = |I_x|$, os polinômios-base I_x, $(I_y + I_x)$, e $(-I_y + I_x)$ não são estritamente positivos, podendo ser constantemente zero, logo nenhum dos polinômios translineares da Eq. (5.14) são válidos. Utilizando um artifício de somar e subtrair um termo constante I_0^2 à Eq. (5.14a), o polinômio mantém a sua funcionalidade e a homogeneidade, mas a presença nova variável pode produzir novos polinômios translineares. Aplicando o algoritmo desenvolvido à Eq. (5.15a), novamente urilizando o intervalo de coeficientes $[-1, \ldots, +1]$, os polinômios translineares obtidos são dados pelo conjunto da Eq. (5.15):

$$I_y^2 - I_0^2 - I_x^2 + I_0^2 \tag{5.15a}$$
$$(I_y + I_0 + I_x)(-I_y + I_x) - I_0(-I_y + I_x) \tag{5.15b}$$
$$(I_y + I_0 + I_x)(-I_y + I_x) - (-I_y + I_0 - I_x)(-I_y + I_x) \tag{5.15c}$$
$$(I_y + I_0)(I_y + I_x) - (I_y + I_x)(I_0 + I_x) \tag{5.15d}$$
$$(I_0 + I_y)(I_0 - I_y) - (I_0 + I_x)(I_0 - I_x) \tag{5.15e}$$
$$(I_y + I_0)(-I_y + I_x) - (I_0 - I_x)(-I_y + I_x) \tag{5.15f}$$
$$(I_y + I_x)(-I_y + I_0 + I_x) - I_0(I_y + I_x) \tag{5.15g}$$
$$I_0(I_y + I_x) - (I_y + I_x)(I_y + I_0 - I_x) \tag{5.15h}$$
$$(I_y + I_x)(-I_y + I_0) - (I_y + I_x)(I_0 - I_x) \tag{5.15i}$$
$$(I_0 + I_x)(-I_y + I_x) - (-I_y + I_0)(-I_y + I_x) \tag{5.15j}$$
$$I_0(-I_y + I_x) - (-I_y + I_0 - I_x)(-I_y + I_x) \tag{5.15k}$$
$$2I_x(-I_y + I_x) - (-I_y + I_x)^2 \tag{5.15l}$$
$$2I_x(I_y + I_x) - (I_y + I_x)^2 \tag{5.15m}$$
$$I_x(I_y + I_x) - I_y(I_y + I_x) \tag{5.15n}$$
$$I_y^2 - I_x^2 \tag{5.15o}$$
$$I_y(-I_y + I_x) - I_x(I_y - I_x) \tag{5.15p}$$

O polinômio modo-corrente dado pela Eq. (5.15e) é o mesmo da Eq. (2.65), encontrado durante a síntese do circuito, mostrando assim, a eficácia do algoritmo desenvolvido em automatizar o processo de decomposição translinear.

5.3.4 Circuito de Seno

Em [48], o autor implementa uma função que aproxima o valor do seno do sinal de entrada de forma estática. O polinômio modo-corrente relativo a esta função matemática, dado pela Eq. (2.21), foi desenvolvido na seção 2.4.2, e é repetido aqui na Eq. (5.16). O polinômio translinear encontrado pelo autor é dado pela Eq. (5.17):

$$I_x{}^3 + I_x{}^2 I_y - I_0{}^2 I_x + I_0{}^2 I_y \tag{5.16}$$

$$(I_0 + I_x)^2 (I_0 - I_x - I_y) - (I_0 - I_x)^2 (I_0 + I_x + I_y) \tag{5.17}$$

Aplicando o novo algoritmo à Eq. (5.16) com coeficientes $[-1, \ldots, +1]$, encontrou-se nove polinômios translineares, dados pelo conjunto da Eq. (5.18).

$$I_y{}^2 (I_x + I_y) - (I_0 + I_x + I_y)(I_0 - I_x - I_y)(I_x - I_y) \tag{5.18a}$$

$$(I_0 + I_x)^2 (I_x + I_y) - 2 I_0 I_x (I_0 + I_x + I_y) \tag{5.18b}$$

$$(I_0 + I_x)^2 (I_0 - I_x - I_y) - (I_0 - I_x)^2 (I_0 + I_x + I_y) \tag{5.18c}$$

$$I_0 I_y (I_0 + I_x) - I_x (I_0 - I_x)(I_0 + I_x + I_y) \tag{5.18d}$$

$$(I_0 - I_x)^2 (I_x + I_y) - 2 I_0 I_x (I_0 - I_x - I_y) \tag{5.18e}$$

$$I_x^2 (I_x + I_y) - I_0^2 (I_x - I_y) \tag{5.18f}$$

$$2 I_y I_0^2 - (I_0 + I_x)(I_0 - I_x)(I_0 + I_y) \tag{5.18g}$$

$$I_0 I_y (I_0 - I_x) - I_x (I_0 + I_x)(I_0 - I_x - I_y) \tag{5.18h}$$

$$2 I_x^2 I_y - (I_x - I_y)(I_0 + I_x)(I_0 - I_x) \tag{5.18i}$$

A Eq. (5.18c) é idêntica à Eq. (5.17), o que significa que o algoritmo novamente funcionou corretamente. Se a corrente I_x é restrita ao intervalo $0 < I_x < I_0$, então I_y é sempre positiva e $I_y < I_x$. Assim, qualquer polinômio translinear do conjunto da Eq. (5.18) é implementável em uma malha translinear válida, especialmente a Eq. (5.18f), que usa a menor quantidade de espelhos de corrente, resultando em um *design* mais compacto, e seu esquemático pode ser visto na Fig. 5.2. Este circuito foi simulado com $V_{dd} = 1V$, emo $I_0 = 10nA$, onde M_1, M_3 e M_5 implementam os polinômios $I_x^2(I_x + I_y)$, e M_2, M_4 e M_6 implementam os polinômios $I_0^2(I_x - I_y)$.

Todos os terminais de corpo são conectados com os terminais de fonte para eliminar o efeito de corpo. As fontes de corrente foram implementadas com espelhos de corrente simples em inversão forte, utilizando um total de 18 transistores, ao invés de 31 utilizados na implementação em [48]. O resultado da simulação é mostrado nos gráficos da Figura 5.3, no qual a linha pontilhada representa a saída do circuito e a linha sólida representa o resposta ideal dada pela Eq. (5.16). A diferença mínima entre as curvas dentro do intervalo válido de $0 < I_x < 10nA$ mostra o quão preciso o processamento de sinais por meio de circuitos translineares pode ser, mesmo com tensões e correntes muito baixas.

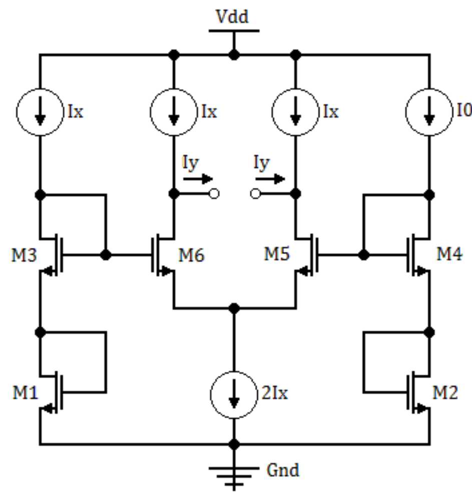


Figura 5.2: Circuito Seno compacto

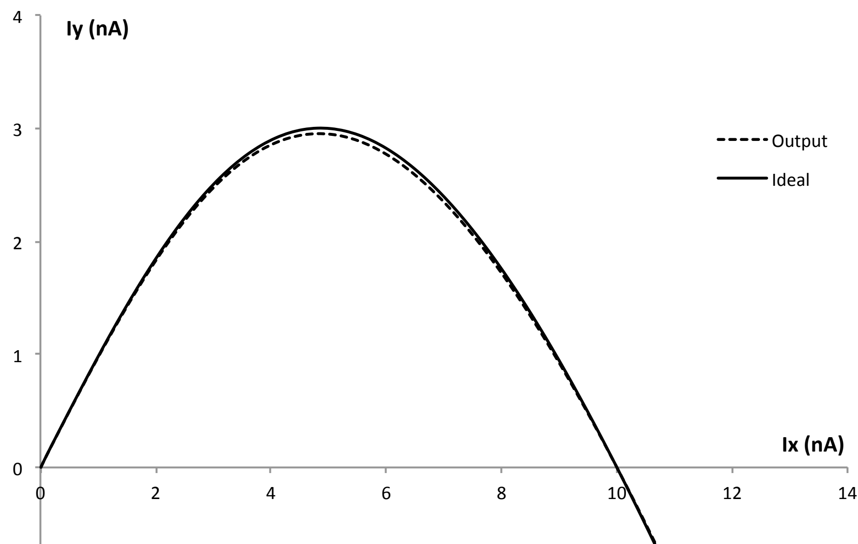


Figura 5.3: Saída do circuito Seno e seu valor ideal

Entretanto, nem toda tecnologia CMOS implementa transistores tipo "N" com substrato isolado. Nestes casos pode-se tentar implementar o circuito utilizando transistores tipo "p", ou então utilizar técnicas de minimização do efeito de corpo [50].

Capítulo 6

Conclusões

Neste trabalho, foi realizado um estudo detalhado a respeito de circuitos translineares e as metodologias de sínteses destes circuitos disponíveis na literatura. Foi realizado uma análise do algoritmo de decomposição translinear não-paramétrica automatizada de [23], e, baseado neste, um novo algoritmo foi desenvolvido. Ambos algoritmos foram implementados em uma ferramenta auto-contida utilizando linguagem de programação "C", possibilitando a verificação da eficácia e eficiência das implementações destes algoritmos. A implementação do novo algoritmo mostrou-se ser mais eficiente em alguns casos e mais segura, por não apresentar riscos de utilização excessiva da memória de aplicativos do computador. Foi implementada uma segunda versão do algoritmo original otimizada para minimizar a memória utilizada, e esta implementação ficou menos eficiente que a implementação do novo algoritmo em todos os casos de teste.

O novo algoritmo também foi aplicado a realizações de circuitos translineares já publicados, o que mostrou que esta ferramenta proporciona realizar a etapa de decomposição translinear de forma automática, vencendo a barreira imposta pela necessidade de "criatividade algébrica" imposta pelos métodos manuais. O novo algoritmo também mostrou-se mais eficaz, pois, para um determinado polinômio modo-corrente, o novo algoritmo pode encontrar mais polinômios translineares que o algoritmo original. O outro importante ganho em eficácia é que o novo algoritmo pode ser utilizado no auxílio do procedimento de decomposição paramétrica, enquanto que o algoritmo original não proporciona esta funcionalidade.

O código fonte da ferramenta desenvolvida que implementa ambos algoritmos está disponível de forma gratuita sob a Licença Pública Geral GNU v.3[1] no anexo deste trabalho e em [51]. Esta ferramenta é auto-contida, ou seja, não depende de pacotes de software proprietários para o seu funcionamento, o que a torna bastante acessível.

[1] GNU General Public License v.3

6.1 Proposta de Trabalhos Futuros

A ferramenta foi implementada como um simples aplicativo de terminal de linha-de-comando. Assim, o desenvolvimento de uma interface gráfica poderia tornar o seu uso mais conveniente. Esta ferramenta também não explora a capacitade de multi-processamento dos processadores de computadores pessoais atuais, que podem chegar a 8 núcleos. Dividir o processamento do vetor-sementes τ_3 entre múltiplos processadores adicionaria um ganho fundamental em eficiência.

A respeito do novo algoritmo, ficou claro que a redução de polinômios-base do vetor τ_2 tem um impacto significativo em sua eficiência, portanto é desejável encontrar e implementar um algoritmo adequado de fatoração polinomial muiltivariável para a ferramenta. A recente implementação gratuita encontrada em [47] parece ser uma boa candidata.

Pode-se investigar também, uma versão do algoritmo em que se possa utilizar coeficientes fracionários de forma eficiente. O desenvolvimento de um algoritmo deste tipo poderia eliminar a necessidade de decomposições paramétricas, pois sempre existe pelo menos um polinômio translinear de um polinômio modo-corrente se todos os coeficientes do conjunto dos números racionais $\mathbb{Q}$ puderem ser utilizados.

Finalmente, a ferramenta pode ser aplicada a sistemas de processamento de sinais conhecidos que tenham restrições severas de nível de tensão de alimentação e de consumo de potência, de modo a tentar encontrar soluções utilizando circuitos translineares que atendam a estas restrições.

REFERÊNCIAS BIBLIOGRÁFICAS

[1] ZHANG, Y. et al. A batteryless 19μw mics/ism-band energy harvesting body sensor node soc for exg applications. *Solid-State Circuits, IEEE Journal of*, v. 48, n. 1, p. 199–213, Jan 2013. ISSN 0018-9200.

[2] CABRERA, F. L.; SOUSA, F. R. de. Optimal design of energy efficient inductive links for powering implanted devices. In: IEEE. *Biomedical Wireless Technologies, Networks, and Sensing Systems (BioWireleSS), 2014 IEEE Topical Conference on*. [S.l.], 2014. p. 37–39.

[3] TANG, S.; JOLESZ, F.; CLEMENT, G. A wireless batteryless deep-seated implantable ultrasonic pulser-receiver powered by magnetic coupling. *Ultrasonics, Ferroelectrics and Frequency Control, IEEE Transactions on*, v. 58, n. 6, p. 1211–1221, June 2011. ISSN 0885-3010.

[4] ALI, M.; ALBASHA, L.; AL-NASHASH, H. A bluetooth low energy implantable glucose monitoring system. In: *Radar Conference (EuRAD), 2011 European*. [S.l.: s.n.], 2011. p. 377–380.

[5] YANG, Z. et al. Wireless energy transmission using ultrasound for implantable devices. In: *Piezoelectricity, Acoustic Waves and Device Applications (SPAWDA), 2013 Symposium on*. [S.l.: s.n.], 2013. p. 1–4.

[6] KILINC, E. et al. A system for wireless power transfer of micro-systems in-vivo implantable in freely moving animals. *Sensors Journal, IEEE*, v. 14, n. 2, p. 522–531, Feb 2014. ISSN 1530-437X.

[7] SALAM, M. et al. An implantable closedloop asynchronous drug delivery system for the treatment of refractory epilepsy. *Neural Systems and Rehabilitation Engineering, IEEE Transactions on*, v. 20, n. 4, p. 432–442, July 2012. ISSN 1534-4320.

[8] FANG, Q. et al. Developing a wireless implantable body sensor network in mics band. *Information Technology in Biomedicine, IEEE Transactions on*, v. 15, n. 4, p. 567–576, July 2011. ISSN 1089-7771.

[9] YAKOVLEV, A.; KIM, S.; POON, A. Implantable biomedical devices: Wireless powering and communication. *Communications Magazine, IEEE*, v. 50, n. 4, p. 152–159, April 2012. ISSN 0163-6804.

[10] MARNAT, L. et al. On-chip implantable antennas for wireless power and data transfer in a glaucoma-monitoring soc. *Antennas and Wireless Propagation Letters, IEEE*, v. 11, p. 1671–1674, 2012. ISSN 1536-1225.

[11] HARB, A.; SAWAN, M. Low-power cmos implantable nerve signal analog processing circuit. In: *Electronics, Circuits and Systems, 2000. ICECS 2000. The 7th IEEE International Conference on.* [S.l.: s.n.], 2000. v. 2, p. 911–914 vol.2.

[12] CROCE, R. et al. Low-power signal processing methodologies for implantable biosensing platforms. In: *Signal Processing in Medicine and Biology Symposium (SPMB), 2013 IEEE.* [S.l.: s.n.], 2013. p. 1–5.

[13] FU, X. et al. A wireless implantable sensor network system for in vivo monitoring of physiological signals. *Information Technology in Biomedicine, IEEE Transactions on*, v. 15, n. 4, p. 577–584, July 2011. ISSN 1089-7771.

[14] NGUYEN, A.-T. et al. Miniaturization of package for an implantable heart monitoring device. In: *Design, Test, Integration and Packaging of MEMS/MOEMS (DTIP), 2013 Symposium on.* [S.l.: s.n.], 2013. p. 1–6.

[15] HADDAD, S. A.; SERDIJN, W. *Ultra low-power biomedical signal processing: an analog wavelet filter approach for pacemakers.* [S.l.]: Springer, 2009. 4–9 p.

[16] PUNZENBERGER, M.; ENZ, C. Low-voltage companding current-mode integrators. In: *Circuits and Systems, 1995. ISCAS '95., 1995 IEEE International Symposium on.* [S.l.: s.n.], 1995. v. 3, p. 2112–2115 vol.3.

[17] PYTHON, D.; ENZ, C. C. A micropower class-ab cmos log-domain filter for dect applications. *Solid-State Circuits, IEEE Journal of*, IEEE, v. 36, n. 7, p. 1067–1075, 2001.

[18] HADDAD, S. et al. An ultra low-power dynamic translinear cardiac sense amplifier for pacemakers. In: *Circuits and Systems, 2003. ISCAS '03. Proceedings of the 2003 International Symposium on.* [S.l.: s.n.], 2003. v. 5, p. V–37–V–40 vol.5.

[19] REDONDO, X.; SERRA-GRAELLS, F. 1 v compact class-ab cmos log filters. In: IEEE. *Circuits and Systems, 2005. ISCAS 2005. IEEE International Symposium on.* [S.l.], 2005. p. 2000–2003.

[20] HADDAD, S. A. P. et al. Ultra low-power analog morlet wavelet filter in 0.18 μm bicmos technology. In: *Solid-State Circuits Conference, 2005. ESSCIRC 2005. Proceedings of the 31st European.* [S.l.: s.n.], 2005. p. 323–326.

[21] HADDAD, S. A.; SERDIJN, W. A. An ultra low-power class-ab sinh integrator. In: ACM. *Proceedings of the 19th annual symposium on Integrated circuits and systems design.* [S.l.], 2006. p. 74–79.

[22] FREY, D.; DRAKAKIS, E. Unifying perspective on log-domain filter synthesis. *Electronics letters*, IET, v. 45, n. 17, p. 861–863, 2009.

[23] MULDER, J. et al. *Dynamic translinear and log-domain circuits: analysis and synthesis.* [S.l.]: Springer, 1998. 74–104 p.

[24] ILSEN, D. *Algebraische Aspekte der Synthese translinearer Netzwerke*. Tese (Doutorado) — Diploma Thesis, Universität Kaiserslautern, 2002.

[25] ILSEN, D.; ROEBBERS, E. J.; GREUEL, G. Algebraic and combinatorial algorithms for translinear network synthesis. *Circuits and Systems I: Regular Papers, IEEE Transactions on*, IEEE, v. 55, n. 10, p. 3131–3144, 2008.

[26] MULDER, J. et al. An rms-dc converter based on the dynamic translinear principle. *Solid-State Circuits, IEEE Journal of*, IEEE, v. 32, n. 7, p. 1146–1150, 1997.

[27] SEEVINCK, E. *Analysis and synthesis of translinear integrated circuits*. [S.l.]: Elsevier Amsterdam, 1988.

[28] MULDER, J. et al. General current-mode analysis method for translinear filters. *Circuits and Systems I: Fundamental Theory and Applications, IEEE Transactions on*, IEEE, v. 44, n. 3, p. 193–197, 1997.

[29] GILBERT, B. Translinear circuits: A proposed classification. *Electronics Letters*, IET, v. 11, n. 1, p. 14–16, 1975.

[30] GILBERT, B. A new wide-band amplifier technique. *Journal of Solid-State Circuits*, IEEE, v. 3, n. 4, p. 353–365, Dec 1968.

[31] GILBERT, B. Translinear circuits: an historical overview. *Analog Integrated Circuits and Signal Processing*, Springer, v. 9, n. 2, p. 95–118, 1996.

[32] MINCH, B. A. Multiple-input translinear element log-domain filters. *Circuits and Systems II: Analog and Digital Signal Processing, IEEE Transactions on*, IEEE, v. 48, n. 1, p. 29–36, 2001.

[33] TSIVIDIS, Y.; MCANDREW, C. *Operation and Modeling of the MOS Transistor*. [S.l.]: Oxford Univ. Press, 2011.

[34] ADAMS, R. W. Filtering in the log domain. In: AUDIO ENGINEERING SOCIETY. *Audio Engineering Society Convention 63*. [S.l.], 1979.

[35] HADDAD, S. A.; SERDIJN, W. A. High-frequency dynamic translinear and log-domain circuits in cmos technology. In: CITESEER. *IEEE INTERNATIONAL SYMPOSIUM ON CIRCUITS AND SYSTEMS*. [S.l.], 2002. p. III–313.

[36] DUERDEN, G. D.; ROBERTS, G. W.; DEEN, M. J. The development of bipolar log domain filters in a standard cmos process. In: IEEE. *Circuits and Systems, 2001. ISCAS 2001. The 2001 IEEE International Symposium on*. [S.l.], 2001. v. 1, p. 145–148.

[37] MINCH, B. A. et al. Translinear circuits using subthreshold floating-gate mos transistors. *Analog Integrated Circuits and Signal Processing*, Springer, v. 9, n. 2, p. 167–179, 1996.

[38] MULDER, J. et al. A generalized class of dynamic translinear circuits. *Circuits and Systems II: Analog and Digital Signal Processing, IEEE Transactions on*, v. 48, n. 5, p. 501–504, May 2001. ISSN 1057-7130.

[39] FREY, D. Log-domain filtering: an approach to current-mode filtering. *IEE Proceedings G (Circuits, Devices and Systems)*, IET, v. 140, n. 6, p. 406–416, 1993.

[40] PERRY, D.; ROBERTS, G. W. The design of log-domain filters based on the operational simulation of lc ladders. *Circuits and Systems II: Analog and Digital Signal Processing, IEEE Transactions on*, IEEE, v. 43, n. 11, p. 763–774, 1996.

[41] DRAKAKIS, E. M.; PAYNE, A. J.; TOUMAZOU, C. ?log-domain state-space?: a systematic transistor-level approach for log-domain filtering. *Circuits and Systems II: Analog and Digital Signal Processing, IEEE Transactions on*, IEEE, v. 46, n. 3, p. 290–305, 1999.

[42] BERNARDIN, L.; MONAGAN, M. B. Efficient multivariate factorization over finite fields. In: *Applied Algebra, Algebraic Algorithms and Error-Correcting Codes*. [S.l.]: Springer, 1997. p. 15–28.

[43] STEEL, A. Conquering inseparability: primary decomposition and multivariate factorization over algebraic function fields of positive characteristic. *Journal of Symbolic Computation*, Elsevier, v. 40, n. 3, p. 1053–1075, 2005.

[44] ALLEM, L. E. *Polinômios Multivariados: fatoraçao e mdc.* Tese (Doutorado) — UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL, 2010.

[45] CHEN, C.; MAZA, M. M. Algorithms for computing triangular decompositions of polynomial systems. In: ACM. *Proceedings of the 36th international symposium on Symbolic and algebraic computation*. [S.l.], 2011. p. 83–90.

[46] LECERF, G. Improved dense multivariate polynomial factorization algorithms. *Journal of Symbolic Computation*, Elsevier, v. 42, n. 4, p. 477–494, 2007.

[47] LEE, M. M.-D. *Factorization of multivariate polynomials*. Tese (Doutorado) — Technische Universität Kaiserslautern, 2013.

[48] MULDER, J. et al. Translinear sin (x)-circuit in mos technology using the back gate. In: IEEE. *Solid-State Circuits Conference, 1995. ESSCIRC'95. Twenty-first European*. [S.l.], 1995. p. 82–85.

[49] SERDIJN, W. A. et al. A low-voltage translinear second-order quadrature oscillator. In: IEEE. *Circuits and Systems, 1999. ISCAS'99. Proceedings of the 1999 IEEE International Symposium on*. [S.l.], 1999. v. 2, p. 701–704.

[50] SERRANO-GOTARREDONA, T.; LINARES-BARRANCO, B.; ANDREOU, A. A general translinear principle for subthreshold mos transistors. *Circuits and Systems I: Fundamental Theory and Applications, IEEE Transactions on*, v. 46, n. 5, p. 607–616, May 1999. ISSN 1057-7122.

[51] ANDRADE, D. NParametricTLDecomp — *2014 Source Code for Non-parametric Translinear Decomposition Application*. 2014. `https://github.com/mudo007/NParametricTLDecomp.git`.

ANEXOS

I. CÓDIGO FONTE DO APLICATIVO

O código fonte deste anexo compreende o aplicativo com a implementação do algoritmo desenvolvido neste trabalho e a implementação do algoritmo de Mulder et. al. [23]. Da forma como está implementado, o aplicativo primeiro executa o algoritmo de Mulder, depois o deste trabalho, e depois re-executa ambos a partir do vetor de polinômios extremamente reduzido τ_4.

Para compilar o código, basta colocar o arquivo "main.c" e o arquivo "main.h" na mesma pasta, e a partir de um terminal executar o comando:

gcc main.c -o NParametricTLDecomp

Este código-fonte foi compilado e executado nos sistemas operacionais Windows© 7, CentOS 5.5, e OS X 10.9.4. A versão mais recente deste código, bem como a contribuição de outros desenvolvedores pode ser encontrada em [51].

I.1 arquivo "main.c'

```
1 /*
2  Decomp_nparametrica: "Reorganizes a multivariable equation so it is
3  suitable to be realized onto a singluar translinear loop analogue
4  current mode circuit."
5
6  Copyright (C) 2014  Diogo Andrade
7
8  This program is free software: you can redistribute it and/or modify
9  it under the terms of the GNU General Public License as published by
10  the Free Software Foundation, either version 3 of the License, or
11  (at your option) any later version.
12
13  This program is distributed in the hope that it will be useful,
14  but WITHOUT ANY WARRANTY; without even the implied warranty of
15  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
16  GNU General Public License for more details.
17
18  You should have received a copy of the GNU General Public License
19  along with this program.  If not, see <http://www.gnu.org/licenses/>.
20
21  email:diogo007@gmail.com
22
23  */
24 #include "main.h"
25
26 int main(int argc, char *argv[])
27 {
28     //Variaveis
29     char    equacao_entrada[302];
30     int i;
31     token   *lista_token = NULL;
32     tabela_literais *lista_literais = NULL;
33     token *expressao_RPN = NULL;
34     arvore_expr *arvore = NULL;
35     lista_expr  *expr_expandida = NULL;
36     lista_expr  *expr_simplificada = NULL;
37     lista_expr *polinomio_base;
38     vetor_polinomios *lista_polinomios = NULL;
39     vetor_polinomios *percorre_polinomios;
40     vetor_sementes  *lista_sementes = NULL;
41     vetor_sementes  *percorre_sementes = NULL;
42     int             contador;
43     vetor_decomp    *decomposicoes_encontradas;
44     int             lim_inferior;
```

```
45     int             lim_superior;
46     time_t    cronometro;
47
48
49
50 //*********
51 // INICIO *
52 //*********
53
54     printf("NParametricTLDecomp  Copyright (C) 2014  Diogo Andrade \nThis program comes with ABSOLUTELY NO WARRANTY.\
       nThis is free software, and you are welcome to redistribute it under certain conditions.\n\nThis application
       performs non-parametric Translinear decomposition onto a homogeneous (all monomials having the same degree)
       multivariate polynomial.\nThe results are suitable for Translinear analog circuit realization with proper
       adjustments.\nIf an error occurs, send a brief description with the polynomial inserted to diogo007@gmail.com.\n\r \
55             Below, type the polynomial to be decomposed (100 characters maximum). Coefficients should be only integer
       numbers. \n\r\t Example:\n\r\t \
56             x^3 + x^2*y - x*z^2 + y*z^2\n\r  (Hit \"enter\" to use it)");
57
58     //leitura da string de entrada
59
60     if ( fgets (equacao_entrada, 100 , stdin) == NULL )
61     {
62         erro(ERRO_002);
63         return(0);
64     }
65
66     //equacao padrao, caso o usuario aperte enter direto. pode ser igual a nova linha ou retorno de carro para tratar
       como o sistema operacional reconhece o ENTER em varias plataformas diferentes
67     if (*equacao_entrada == '\n' || *equacao_entrada == '\r')
68         sprintf(equacao_entrada, "x^3 + x^2*y - x*z^2 + y*z^2");
69
70 //*****************
71 // ANALISE LEXICA *
72 //*****************
73
74     //leitura dos tokens
75     if((lista_token = le_tokens(equacao_entrada)) == NULL)
76     {
77         //limpeza de ponteiros
78         system("PAUSE");
79         return 0;
80     }
81
82     //converte os literais em codigos - apenas de exemplo
83     constroi_tabela_literais(&lista_literais,lista_token);
84
85 #if defined DEBUG_LEXICO
86     //caso a leitura dos tokens tenha sido correta, imprimir os tokens
87     imprime_tokens(lista_token);
88 #endif
89
90 //********************
91 // ANALISE SINTATICA *
92 //********************
93
94     //Criacao das pilhas de avaliacao de expressao
95
96     expressao_RPN = constroi_lista_expr(lista_token);
97     arvore = constroi_arvore_expr(expressao_RPN);
98 #if defined DEBUG_EXPR
99     imprime_arvore_expr(arvore);
100    imprime_lista_expr(expressao_RPN);
101 #endif
102
103    //Expansao da arvore de expressoes
104    expr_expandida = constroi_lista_expressoes_exp(arvore);
105 #if defined DEBUG_EXPAND
106    printf("\n equacao expandida:");
107    imprime_lista_expr_expandida(expr_expandida,lista_literais);
108 #endif
109
110 //********************
111 // ANALISE SEMANTICA *
112 //********************
113
114    //Simplificacao da expressao
115    expr_simplificada = simplifica_expr_expandida(expr_expandida);
116    destroi_lista_expr_expandida(expr_expandida);
117 #if defined DEBUG_SIMPLIFY
118    printf("\n equacao simplificada:");
119    imprime_lista_expr_expandida(expr_simplificada,lista_literais);
```

```
120 #endif
121
122     //ordenacao lexdeg
123    expr_simplificada = lexdegbubblesort(expr_simplificada);
124 #if defined DEBUG_SIMPLIFY
125     printf("\n equacao reordenada lex:");
126     imprime_lista_expr_expandida(expr_simplificada,lista_literais);
127 #endif
128
129
130 /**********************
131  Base-Polynomial generation and reduction
132  *************************/
133
134     //imprimir a lista de literais e construir o polinomio base
135
136     polinomio_base = gera_polinomio_base(lista_literais);
137
138     //leitura dos limites superiores e inferiores dos coeficientes
139     printf("\n Insert the inferior limit of variables coefficients (standard -1)");
140     if ( fgets (equacao_entrada, 100 , stdin) == NULL )
141     {
142         erro(ERRO_002);
143         return(0);
144     }
145
146     //tratamento do numero inserido
147     //se apertar enter direto, e -1
148     if (*equacao_entrada == '\n' || *equacao_entrada == '\r')
149         lim_inferior = -1;
150     else
151     {
152         lim_inferior = atoi(equacao_entrada);
153     }
154
155     //limite superior
156     printf("\n Insert the superior limit of variables coefficients (standard +1)");
157     if ( fgets (equacao_entrada, 100 , stdin) == NULL )
158     {
159         erro(ERRO_002);
160         return(0);
161     }
162
163     //tratamento do numero inserido
164     //se apertar enter direto, e -1
165     if (*equacao_entrada == '\n' || *equacao_entrada == '\r')
166         lim_superior = 1;
167     else
168     {
169         lim_superior = atoi(equacao_entrada);
170     }
171
172     //imprimir os parametros colhidos
173     printf("\n Expanded, simplified and re-ordered Polynomial:");
174     imprime_lista_expr_expandida(expr_simplificada, lista_literais);
175     printf("\n coefficient inferior limit: %d", lim_inferior);
176     printf("\n coefficient superior limit: %d\n", lim_superior);
177
178
179     //construcao dos vetores, aqui com coeficientes especificados pelo usuario
180     lista_polinomios = gera_vetor(lista_polinomios, polinomio_base, polinomio_base,lim_inferior, lim_superior);
181
182     //rebobina a lista
183     while (lista_polinomios->polinomio_anterior != NULL)
184     {
185         lista_polinomios = lista_polinomios->polinomio_anterior;
186     }
187
188     //contagem de polinomios T0
189     percorre_polinomios = lista_polinomios;
190     contador = 0;
191     while (percorre_polinomios != NULL)
192     {
193         contador++;
194         percorre_polinomios = percorre_polinomios->proximo_polinomio;
195     }
196     printf("\n Total number of Base-Polynomials generated (T0 set) is: %d\n",contador);
197
198     //elimina o polinomio nulo
199     lista_polinomios = elimina_zero(lista_polinomios);
200
```

```
201     //elimina os polinomios inteiramente negativos
202     lista_polinomios = remove_polinomios_negativos(lista_polinomios);
203 
204     //contagem de polinomios T1
205     percorre_polinomios = lista_polinomios;
206     contador = 0;
207     while (percorre_polinomios != NULL)
208     {
209         contador++;
210         percorre_polinomios = percorre_polinomios->proximo_polinomio;
211     }
212     printf("\n Number of Base-Polynomials after removing strict negative BP's (T1 set) is: %d\n",contador);
213 
214     //elimina os polinomios redundantes
215     lista_polinomios = remove_polinomios_redundantes(lista_polinomios);
216 
217     //contagem de polinomios T2
218     percorre_polinomios = lista_polinomios;
219     contador = 0;
220     while (percorre_polinomios != NULL)
221     {
222         contador++;
223         percorre_polinomios = percorre_polinomios->proximo_polinomio;
224     }
225     printf("\n Number of Base-Polynomials after redundancy removal (T2 set) is: %d\n",contador);
226 
227     //gerar vetor T3
228     lista_sementes = gera_vetor_semente(lista_polinomios, expr_simplificada);
229 
230     //contagem de pares de polinomios T3
231     percorre_sementes = lista_sementes;
232     contador = 0;
233     while (percorre_sementes != NULL)
234     {
235         contador++;
236         percorre_sementes = percorre_sementes->conjunto_prox;
237     }
238     printf("\n Number of Base-Polynomial pairs (T3 set) is: %d\n",contador);
239 
240     //gerar T4 Ũaps T3 ter sido gerado
241     lista_polinomios = reconta_polinomios(lista_sementes,lista_polinomios);
242 
243     //contagem de T4
244     percorre_polinomios = lista_polinomios;
245     contador = 0;
246     while (percorre_polinomios != NULL)
247     {
248         contador++;
249         percorre_polinomios = percorre_polinomios->proximo_polinomio;
250     }
251     printf("\n Number of Base-Polynomials after seed generation (reduced set T4) is: %d\n",contador);
252 
253 
254 /**********************
255 Recursive Division By BP pairs
256 **********************/
257 
258     //Escolha de qual algoritmo sera utilizado
259     //leitura dos limites superiores e inferiores dos coeficientes
260     printf("\n Select recursive division Algorithm:[A]ndrade, [M]ulder fast, Mulder Memory [S]afe: (Default: A)");
261     if ( fgets (equacao_entrada, 100 , stdin) == NULL )
262     {
263         erro(ERRO_002);
264         return(0);
265     }
266 
267     // Se opcao estranha for utilizada, rea;lizar algoritmo de diogo
268     if (equacao_entrada[0] != 'M' && equacao_entrada[0] != 'm' && equacao_entrada[0] != 'A' && equacao_entrada[0] != 'a'
         &&
269         equacao_entrada[0] != 'S' && equacao_entrada[0] != 's')
270     {
271         equacao_entrada[0] = 'A';
272     }
273 
274     //preparar para a execucao
275     global_num_parfrac = 0;
276     decomposicoes_encontradas = NULL;
277     inicia_cronometro(&cronometro);
278 
279     //selecionar o algoritmo a ser utilizado
280     switch (equacao_entrada[0])
```

```
281     {
282         case 'A':
283         case 'a':
284             printf("\n Running Andrade's algorithm...");
285             decomposicoes_encontradas = encontra_decomp(lista_sementes,deg(expr_simplificada), expr_simplificada,
         lista_literais);
286             break;
287
288         case 'M':
289         case 'm':
290             printf("\n Running Mulder's algorithm (fast version)...");
291             decomposicoes_encontradas = encontra_decomp_mulder(lista_polinomios, deg(expr_simplificada),
         expr_simplificada, lista_literais);
292             break;
293
294         case 'S':
295         case 's':
296             printf("\n Running Mulder's algorithm (memory safe version)...");
297             decomposicoes_encontradas = encontra_decomp_mulder_safe(lista_polinomios, deg(expr_simplificada),
         expr_simplificada, lista_literais);
298             break;
299
300     }
301     para_cronometro(&cronometro);
302
303     //verifica quantos PB's de fato fizeram parte de alguma decomposicao
304     lista_polinomios = remove_polinomios_nao_pertencentes(decomposicoes_encontradas,lista_polinomios);
305
306     //contagem Ûps-encontrar decomposicoes
307     percorre_polinomios = lista_polinomios;
308     contador = 0;
309     while (percorre_polinomios != NULL)
310     {
311         contador++;
312         percorre_polinomios = percorre_polinomios->proximo_polinomio;
313     }
314
315     printf("\n Number of Base-Polynomials effectively used in any translinear polynomial (extremely reduced set T5) is: %
         d",contador);
316
317     //imprime apenas os polinomios utilizados em alguma decomposicao
318     printf("\n The base polynomials used in any of the translinear polynomials found are:\n");
319     percorre_polinomios = lista_polinomios;
320     contador = 0;
321     while (percorre_polinomios != NULL)
322     {
323         contador++;
324         imprime_lista_expr_expandida(percorre_polinomios->polinomio->P, lista_literais);
325         //imprime o identificador do polinomio
326         printf(" \t%d\n",percorre_polinomios->polinomio->id);
327         percorre_polinomios = percorre_polinomios->proximo_polinomio;
328     }
329
330
331     //elimina o vetor de polinomios translineares
332     destroi_vetor_decomp(decomposicoes_encontradas);
333     destroi_lista_sementes(lista_sementes);
334     decomposicoes_encontradas = NULL;
335     lista_sementes = NULL;
336
337     //destroi o vetor de polinomios
338     while (lista_polinomios != NULL)
339     {
340         percorre_polinomios = lista_polinomios->proximo_polinomio;
341         destroi_lista_expr_expandida(lista_polinomios->polinomio->P);
342         free(lista_polinomios->polinomio);
343         free(lista_polinomios);
344         lista_polinomios = percorre_polinomios;
345     }
346
347     //destroi estruturas de dados auxiliares
348     destroi_lista_expr_expandida(polinomio_base);
349     destroi_tabela_literais(lista_literais);
350     destroi_lista(lista_token);
351     destroi_arvore_expr(arvore);
352     destroi_lista_expr(expressao_RPN);
353     destroi_lista_expr_expandida(expr_simplificada);
354     printf("\nPress any key to continue...");
355     getchar();
356     return 0;
357 }
```

I.2 arquivo “main.h”

```
1
2 /*
3  Decomp_nparametrica: "Reorganizes a multivariable equation so it is
4  suitable to be realized onto a singluar translinear loop analogue
5  current mode circuit."
6
7  Copyright (C) 2014  Diogo Andrade
8
9  This program is free software: you can redistribute it and/or modify
10 it under the terms of the GNU General Public License as published by
11 the Free Software Foundation, either version 3 of the License, or
12 (at your option) any later version.
13
14 This program is distributed in the hope that it will be useful,
15 but WITHOUT ANY WARRANTY; without even the implied warranty of
16 MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
17 GNU General Public License for more details.
18
19 You should have received a copy of the GNU General Public License
20 along with this program.  If not, see <http://www.gnu.org/licenses/>.
21
22 email:diogo007@gmail.com
23
24 */
25 #include <stdio.h>
26 #include <stdlib.h>
27 #include <string.h>
28 #include <math.h>
29 #include <ctype.h>
30 #include <time.h>
31
32
33 //definicao de debug
34 //#define DEBUG_LEXICO
35 //#define DEBUG_EXPR
36 //#define DEBUG_EXPAND
37 //#define DEBUG_SIMPLIFY
38
39 //Mensagens de erro
40 #define ERRO_001    "ERRO: Memoria Insuficiente"
41 #define ERRO_002    "ERRO: Equacao muito longa"
42 #define ERRO_003    "ERRO: Equacao errada"
43 #define ERRO_004    "ERRO: Limite de variaveis violado (50)"
44
45 //definicoes boleanas
46 #define TRUE    1
47 #define FALSE   0
48
49 //tipos
50 //estrutura de tokens para parse inicial
51 typedef struct _token
52 {
53     int tipo;           //dependendo do tipo do token, uma das ávariveis abaixo é preenchida
54     char *literal;      //string do literal
55     char codigo;        //caracter do codigo correspondente Ĺ string do literal, ou no caso de operadores e abre_fecha
       pareneteses, o caracter correspondente
56     int parametro;      //valor numerico do parametro
57     int operador;       //operador ámatemtico
58     int abre_fecha;     //abre ou fecha parenteses
59     struct _token *proximo_token;  //ponteiro para o proximo token da fila.
60 }token;
61
62 //estrutura para a lista de literais e seus codigos, limitados a 50 no total
63 typedef struct _tabela_literais
64 {
65     char *literal;
66     char codigo;     //o codigo e so uma letra de A-Z maiuscula ou minuscula
67     struct _tabela_literais *proximo_codigo;
68 } tabela_literais;
69
70 //Estrutura da lista que representara a expressao expandida
71 typedef struct _lista_expr
72 {
73   char  *codigos_numerador; //As variaveis contidas no numerador do termo
74   struct _lista_expr  *codigos_denominador; //As variaveis contidas no numerador do termo
75   //char  sinal;    //se e positivo ou negativo
76   double  parametro;  //Inteiro que multiplica o termo
77   struct _lista_expr  *proximo;
```

```
78    struct _lista_expr  *anterior;
79 } lista_expr;
80
81 //Estrutura da arvore de expressoes:
82 typedef struct _arvore_expr
83 {
84     token  *elemento;               //ponteiro para o token que sera inserido na arvore
85     struct _arvore_expr *esquerda;  //ponteiro para o elemento a direita
86     struct _arvore_expr *direita;   //ponteiro para o elemento a esquerda
87 } arvore_expr;
88
89 //Estrutura de pilha para conversao da expressao em RPN
90 typedef struct _pilha_expr
91 {
92     token  *elemento;               //ponteiro para o token que sera inserido na pilha
93     struct _pilha_expr *proximo;
94 } pilha_expr;
95
96 //Estrutura das pilhas de construcao da arvore
97 typedef struct _pilha_arvore
98 {
99     arvore_expr *node;
100    struct _pilha_arvore *proximo;
101 } pilha_arvore;
102
103 typedef struct _polinomio
104 {
105    lista_expr *P;
106    int         id;
107    int         flag_approved;
108
109 }polinomio;
110
111 //estrutura de vetor de combinacao linear das variaveis de entrada
112 typedef struct _vetor_polinomios
113 {
114    polinomio *polinomio;
115    struct _vetor_polinomios *proximo_polinomio;
116  struct _vetor_polinomios *polinomio_anterior;
117
118 }vetor_polinomios;
119
120 //estrutura que guarda combinacoes de polinomios que podem ser semente de uma decomposicao
121 typedef struct _vetor_sementes
122 {
123    polinomio P1;
124    polinomio P2;
125    lista_expr *quociente;
126    lista_expr *R1;
127    lista_expr *R2;
128    struct _vetor_sementes *conjunto_prox;
129  struct _vetor_sementes *conjunto_ant;
130
131 }vetor_sementes;
132
133 //estrutura que guarda decomposicao parametrica encontrada
134 typedef struct _vetor_decomp
135 {
136    vetor_polinomios *poly_pares;
137    vetor_polinomios *poly_impares;
138    lista_expr *resto_impar;
139    lista_expr *resto_par;
140
141    struct _vetor_decomp *prox_decomp;
142    struct _vetor_decomp *ant_decomp;
143
144
145 } vetor_decomp;
146
147 //estrutura que guarda decomposicoes parciais encontradas ao executar o algoritmo do mulder
148 typedef struct _vetor_decomp_simple
149 {
150    vetor_polinomios *polinomios;
151    lista_expr *resto;
152
153    struct _vetor_decomp_simple *prox_decomp;
154    struct _vetor_decomp_simple *ant_decomp;
155
156 } vetor_decomp_simple;
157
158 //globais
```

```
159
160 //variavel global para debug
161 int global = 0;
162 //contador de fracoes parciais executadas
163 unsigned int global_num_parfrac = 0;
164
165 //definicoes para a struct
166 #define tipo_literal     1
167 #define tipo_parametro  2
168 #define tipo_operador   3
169 #define tipo_abre_fecha 4
170
171 //tipos de operadores
172 #define operador_mais           1
173 #define operador_menos          2
174 #define operador_multiplicacao  3
175 #define operador_potenciacao    4
176 #define operador_negacao        5
177
178
179 //tipos de abre_fecha
180 #define abre_parenteses     0
181 #define fecha_parenteses    1
182
183 //prototipos de funcoes de analise lexica
184 void erro (char* );
185 token *le_tokens(char *);
186 int isoperator (char );
187 int isabre_fecha (char );
188
189 token *cria_token(token **);
190 void destroi_lista( token *);
191 char insere_tabela_literais(tabela_literais **, char*);
192 void destroi_tabela_literais(tabela_literais *);
193 void constroi_tabela_literais(tabela_literais **, token *);
194
195
196 //prototipo das funcoes de construtor de arvore de expressoes
197 token *retira_pilha (pilha_expr **);
198 int insere_pilha(pilha_expr **, token *);
199 void destroi_pilha(pilha_expr *);
200 int insere_lista_expr(token **, token *);
201 token *constroi_lista_expr(token *);
202 int prioridade_operador(token *);
203 arvore_expr *constroi_arvore_expr(token *);
204 arvore_expr *cria_no_arvore(token *);
205 int insere_pilha_arvore(pilha_arvore **, arvore_expr *);
206 arvore_expr *retira_pilha_arvore (pilha_arvore **);
207 void destroi_arvore_expr(arvore_expr *);
208 void destroi_lista_expr( token *);
209
210 //prototipo das funcoes de expansor de expressoes
211 void string_sort(char **);
212 lista_expr *constroi_lista_expressoes_exp(arvore_expr *);
213 lista_expr *multiplica_expr( lista_expr *, lista_expr *);
214 void destroi_lista_expr_expandida(lista_expr *);
215
216 //prototipo das funcoes de simplificador de õexpresses
217 lista_expr *simplifica_expr_expandida(lista_expr *);
218 lista_expr *remove_lista_expr(lista_expr *, lista_expr *);
219 lista_expr *copia_lista_expr(lista_expr *);
220
221 //prototipo de funcoes de divisao polinomial
222 int lexdeg(char *, char *);
223 lista_expr *lexdegbubblesort(lista_expr *);
224 int polydiv(lista_expr *, lista_expr *, lista_expr **, lista_expr **);
225 lista_expr *divide_monomio(lista_expr *,lista_expr *);
226 void subtrai_expr(lista_expr **, lista_expr *);
227 void soma_expr(lista_expr *, lista_expr *);
228 lista_expr *constroi_elemento_zerado(void);
229
230 //prototipo das funcoes de expansao em fracoes parciais
231 int partial_fraction_expansion(lista_expr *, lista_expr *, lista_expr *,lista_expr **, lista_expr **, lista_expr **);
232 lista_expr *substitui_var(lista_expr *, lista_expr *, char );
233
234 //prototipo das funcoes de geracao do vetor de polinomios
235 vetor_polinomios *gera_vetor(vetor_polinomios *, lista_expr *, lista_expr *, int , int );
236 vetor_polinomios *elimina_zero(vetor_polinomios *);
237 vetor_polinomios *remove_polinomio(vetor_polinomios *);
238 vetor_polinomios *remove_polinomio_retorna_anteror(vetor_polinomios *);
239 lista_expr *gera_polinomio_base(tabela_literais *);
```

```
240 vetor_polinomios *remove_polinomios_negativos (vetor_polinomios *);
241 vetor_polinomios *remove_polinomios_redundantes (vetor_polinomios *);
242 vetor_polinomios *reconta_polinomios(vetor_sementes *,vetor_polinomios *);
243 vetor_polinomios *remove_polinomios_nao_pertencentes(vetor_decomp *,vetor_polinomios *);
244
245 //prototipo das funcoes que implementam o algoritmo propriamente dito
246 int deg(lista_expr *);
247 vetor_sementes *gera_vetor_semente(vetor_polinomios *, lista_expr *);
248 vetor_sementes *novo_vetor_semente(void);
249 void destroi_lista_sementes(vetor_sementes *);
250 vetor_polinomios *novo_vetor_polinomios(void);
251 vetor_decomp *novo_vetor_decomp(void);
252 void insere_polinomio(vetor_polinomios **,polinomio *);
253 void destroi_decomp(vetor_decomp *);
254 void destroi_vetor_polinomios(vetor_polinomios *);
255 void destroi_vetor_decomp(vetor_decomp *);
256 vetor_decomp *encontra_decomp(vetor_sementes *, int, lista_expr *, tabela_literais *);
257 int prova_real(vetor_decomp *, lista_expr *);
258 vetor_decomp *insere_lista_decomp(vetor_decomp *, vetor_decomp *);
259 vetor_decomp *copia_vetor_semente(vetor_sementes *);
260 void encontra_decomp_recursiva(vetor_decomp *, vetor_decomp *, vetor_decomp **, int, lista_expr *, tabela_literais *, int
      *);
261 vetor_decomp *copia_semente(vetor_sementes *);
262 vetor_decomp *copia_decomp(vetor_decomp *);
263 void elimina_decomp_redundantes(vetor_decomp *);
264 vetor_polinomios *ordena_polinomios(vetor_polinomios *);
265 int compara_decomp(vetor_decomp *, vetor_decomp *);
266
267 //funcoes que implementam o algoritmo do mulder
268 void combina_decomp_mulder(vetor_decomp_simple *, vetor_decomp_simple *,polinomio *, polinomio *, lista_expr *,
      vetor_decomp **, int *, tabela_literais *);
269 vetor_decomp *encontra_decomp_mulder(vetor_polinomios *, int , lista_expr *, tabela_literais *);
270 vetor_decomp *encontra_decomp_mulder_safe(vetor_polinomios *, int , lista_expr *, tabela_literais *);
271
272 void encontra_decomp_parcial(vetor_decomp_simple *, polinomio *, lista_expr *, vetor_polinomios *, int grau,
      vetor_decomp_simple **);
273 vetor_decomp_simple *copia_decomp_simples(vetor_decomp_simple *);
274 vetor_decomp_simple *novo_vetor_decomp_simples(void);
275 vetor_decomp_simple *insere_decomp_simples(vetor_decomp_simple *, vetor_decomp_simple *);
276 void destroi_decomp_simples(vetor_decomp_simple *);
277 vetor_decomp_simple *novo_vetor_decomp_simples(void);
278 void destroi_vetor_decomp_simples(vetor_decomp_simple *);
279
280 //funcoes experimentais do algoritmo do mulder memory safe
281 void encontra_decomp_parcial_primaria(vetor_decomp_simple *, polinomio *, lista_expr *, polinomio *, lista_expr *,
      vetor_polinomios *, int , vetor_decomp **, int *,tabela_literais *, lista_expr *);
282 void encontra_decomp_parcial_secundaria(vetor_decomp_simple *, polinomio *, lista_expr *, vetor_polinomios *, int ,
      vetor_decomp **, vetor_decomp_simple *, polinomio *base_primario, int *,tabela_literais *, lista_expr *);
283
284 //funcoes de tamanho de memoria
285 int decomp_size(vetor_decomp *);
286 int poly_size(vetor_polinomios *);
287 int expr_size(lista_expr *);
288
289 //prototipo das funcoes de interface
290 void imprime_lista_expr_expandida(lista_expr *, tabela_literais *);
291 void print_monomio(char *, tabela_literais *);
292 void imprime_arvore_expr(arvore_expr *);
293 void imprime_tokens(token* );
294 void imprime_decomposicao(vetor_decomp *, tabela_literais *);
295
296 //erro - imprime uma mensagem de erro na tela
297 void erro (char* n_erro)
298 {
299     printf("\n\r\t%s\n",n_erro);
300 }
301
302 //le_tokens - realiza o parse da ãequaao de entrada para construir uma lista de tokens
303 token *le_tokens(char *equacao_entrada)
304 {
305     //o parser e uma maquina de estados que percorre a string da ãequaao caracter a caracter
306     int estado = 0;                              //variavel de estado
307     char *inicio_literal = NULL;                 //ponteiro temporario para manipulacao de literais
308     token *lista_token = NULL;                   //ponteiro para a lista de tokens a ser retornada
309     token *token_atual = NULL;                   //proximo token a ser preenchido;
310
311     //variaveis para calculos temporarios
312     int n_chars;    //calculo de numero de caracteres de um literal
313     int cont = 0;   //contador de caracteres
314     //repetir éat encontrar o caracter nulo, e continuar o laco se estiver no meio de um reconhecimento de literal.
315     while (*(equacao_entrada + cont) != '\0' || estado == 2)
```

```
316     {
317 #if defined DEBUG_LEXICO
318     printf("\n%c\n",*(equacao_entrada + cont));
319 #endif
320         switch(estado)
321         {
322             case 0: //estado inicial
323                 if (isspace(*(equacao_entrada + cont)))
324                 {
325                     //nada é feito
326                     break;
327                 }
328                 //operadores matematicos
329                 if (isoperator(*(equacao_entrada + cont)))
330                 {
331                     //cria um token na lista
332                     token_atual = cria_token(&lista_token);
333 
334                     //preenche o tipo
335                     token_atual->tipo = tipo_operador;
336 
337                     //descobre qual o operador e preenche na estrutura
338                     switch(*(equacao_entrada + cont))
339                     {
340                         case '+':
341                             token_atual->operador = operador_mais;
342                             token_atual->codigo = '+';
343                             break;
344                         case '-':
345                             //a diferenciacao entre subtracao e a operacao de negacao sera tratada mais tarde.
346                             token_atual->operador = operador_menos;
347                             token_atual->codigo = '-';
348                             break;
349                         case '*':
350                             token_atual->operador = operador_multiplicacao;
351                             token_atual->codigo = '*';
352                             break;
353                         case '^':
354                             token_atual->operador = operador_potenciacao;
355                             token_atual->codigo = '^';
356                             break;
357                     }
358                     break;
359                 }
360                 //parametros
361                 if (isdigit(*(equacao_entrada + cont)))
362                 {
363                     //cria um token na lista
364                     token_atual = cria_token(&lista_token);
365 
366                     //preenche o tipo
367                     token_atual->tipo = tipo_parametro;
368 
369                     //comeaa a preencher o numero
370                     token_atual->parametro = (int)(*(equacao_entrada + cont) - '0');
371 
372 
373                     //muda o estado
374                     estado = 1;
375                     break;
376                 }
377                 //literais
378                 if (isalpha(*(equacao_entrada + cont)))
379                 {
380 
381                     //cria um token na lista
382                     token_atual = cria_token(&lista_token);
383 
384                     //preenche o tipo
385                     token_atual->tipo = tipo_literal;
386 
387                     //marca o inicio do literal
388                     token_atual->literal = (equacao_entrada + cont);
389 
390                     //muda o estado
391                     estado = 2;
392 
393                     break;
394                 }
395                 //abre e fecha parenteses
396                 if (isabre_fecha(*(equacao_entrada + cont)))
```

```
397                 {
398                     //cria um token na lista
399                     token_atual = cria_token(&lista_token);
400 
401                     //preenche o tipo
402                     token_atual->tipo = tipo_abre_fecha;
403                      //descobre qual o operador e preenche na estrutura
404                     switch(*(equacao_entrada + cont))
405                     {
406                         case '(':
407                             token_atual->abre_fecha = abre_parenteses;
408                             token_atual->codigo = '(';
409                             break;
410                         case ')':
411                             token_atual->abre_fecha = fecha_parenteses;
412                             token_atual->codigo = ')';
413                             break;
414                     }
415                     break;
416                 }
417 
418                 //se ãno encontrar nenhum dos caracteres esprados, a ãequaao áest errada
419                 destroi_lista(lista_token);
420                 erro(ERRO_003);
421                 //imprimir êtamm a equacao a partir do erro
422                 printf("-> %s\n",(equacao_entrada + cont));
423                 return(NULL);
424                 break;
425 
426             case 1: //numero
427                 //se o proximo caracter ê um digito, multiplicar o parametro atual por 10 e adicionar o digito
428 
429                 if (isdigit(*(equacao_entrada + cont)))
430                 {
431                     token_atual->parametro *= 10;
432                     token_atual->parametro += (int)(*(equacao_entrada + cont) - '0');
433                 }
434                 else
435                 {
436                      //qualquer outra coisa significa que o numero ja foi lido
437                     //volta o caracter, para que seja lido no proximo loop
438                     cont--;
439                     estado = 0;
440                 }
441                 break;
442 
443             case 2: //literal
444                 //se o proximo for digito ou letra, ou _, continuar lendo
445                 if(isdigit(*(equacao_entrada + cont)) || isalpha(*(equacao_entrada + cont)) || (*(equacao_entrada + cont)
          == '_'))
446                     break;
447                 else
448                 {
449                     //salva a âposiao do inico do literal
450                     inicio_literal = token_atual->literal;
451 
452                     //calcula o numero de caracteres necessarios
453                     n_chars = (int)((equacao_entrada + cont) - inicio_literal);
454 
455                     //aloca memoria para o literal encontrado com espaao para o terminador nulo
456                     if((token_atual->literal = (char *)malloc((n_chars + 1)*sizeof(char))) == NULL)
457                     {
458                         erro(ERRO_001);
459                         destroi_lista(lista_token);
460                         return (NULL);
461                     }
462                     //copia a string do literal;
463                     if((strncpy ( token_atual->literal, inicio_literal, n_chars )) == NULL)
464                     {
465                         erro(ERRO_001);
466                         destroi_lista(lista_token);
467                         return (NULL);
468                     }
469 
470                     //coloca o terminador nulo
471                     *(token_atual->literal + n_chars) = '\0';
472 
473                     //volta o caracter, para que seja lido no proximo loop
474                     cont--;
475                     estado = 0;
476                     break;
```

```
477                 }
478                 break;
479
480         }
481         //proximo caracter a ser lido
482         cont++;
483     }
484     //retorna a lista de tokens
485     return (lista_token);
486 }
487
488 //isoperator - verifica se o caracter é um operador ámatemtico
489 int isoperator (char digito)
490 {
491     if ((digito == '+') || (digito == '-') || (digito == '*') || (digito == '^'))
492             return TRUE;
493     else
494         return FALSE;
495 }
496
497 //isoabre_fecha - verifica se o caracter é um abre ou fecha parenteses
498 int isabre_fecha (char digito)
499 {
500     if ((digito == '(') || (digito == ')'))
501             return TRUE;
502     else
503         return FALSE;
504 }
505
506 //cria_token - aloca memoria para um novo token no fim da fila e retorna o ponteiro para ele
507 token *cria_token(token **lista_token)
508 {
509     token *token_atual;     //variavel para percorrer a lista
510
511     //verifica se a lista esta vazia e a cria se ánecessrio
512     if ((*lista_token) == NULL)
513     {
514         if (((*lista_token) = (token *)malloc(sizeof(token))) != NULL)
515         {
516             (*lista_token)->proximo_token = NULL;
517             (*lista_token)->tipo = 0;
518             (*lista_token)->literal = NULL;
519             (*lista_token)->parametro = 0;
520             (*lista_token)->operador = 0;
521             (*lista_token)->abre_fecha = 0;
522
523             return(*lista_token);
524         }
525     }
526     else
527     {
528         token_atual = (*lista_token);
529         while(token_atual->proximo_token != NULL)
530         {
531             token_atual = token_atual->proximo_token;
532         }
533         if ((token_atual->proximo_token = (token *)malloc(sizeof(token))) != NULL)
534         {
535             token_atual->proximo_token->proximo_token = NULL;
536             token_atual->proximo_token->tipo = 0;
537             token_atual->proximo_token->literal = NULL;
538             token_atual->proximo_token->parametro = 0;
539             token_atual->proximo_token->operador = 0;
540             token_atual->proximo_token->abre_fecha = 0;
541             return(token_atual->proximo_token);
542         }
543     }
544     //se ocorreu um erro apenas retorna NULL
545     return (NULL);
546 }
547
548 //destroi lista - ãfunao para destruir todos os ponteiros da lista em caso de erro ou fim do programa
549 void destroi_lista( token *lista)
550 {
551     //percorre a lista recursivamente éat encontrar o ultimo elemento
552     if(lista->proximo_token != NULL)
553     {
554         destroi_lista(lista->proximo_token);
555     }
556     //destroi os ponteiros dos literais e dos proprios tokens nos retornos das chamadas recursivas.
557     if (lista->literal != NULL)
```

```
558     {
559         free(lista->literal);
560     }
561     lista->literal = NULL;
562     lista->proximo_token = NULL;
563     free(lista);
564 }
565 
566 //percorre a lista imprimindo os tokens encontrados
567 void imprime_tokens(token *lista_token)
568 {
569     int contador = 0;
570     token   *p_lista = lista_token;
571 
572     while(p_lista != NULL)
573     {
574         switch(p_lista->tipo)
575         {
576             case tipo_literal:
577                 printf("#%d literal \"%s\"\n\r ",contador, p_lista->literal);
578                 contador++;
579                 break;
580 
581             case tipo_parametro:
582                 printf("#%d parametro \"%d\"\n\r ",contador, p_lista->parametro);
583                 contador++;
584                 break;
585             case tipo_operador:
586                 printf("#%d operador \"%c\"\n\r ",contador, p_lista->codigo);
587                 contador++;
588                 break;
589             case tipo_abre_fecha:
590                 printf("#%d abre_fecha \"%c\"\n\r ",contador, p_lista->codigo);
591                 contador++;
592                 break;
593 
594         }
595         p_lista = p_lista->proximo_token;
596     }
597 }
598 
599 //cria_tablea_literais - aloca memoria para um novo literal e retorna o seu codigo
600 char insere_tabela_literais(tabela_literais **lista_literais, char *literal)
601 {
602     tabela_literais *codigo_atual;     //variavel para percorrer a lista
603 
604     //verifica se a lista áest vazia e a cria se ánecessrio
605     if ((*lista_literais) == NULL)
606     {
607         if (((*lista_literais) = (tabela_literais *)malloc(sizeof(tabela_literais))) != NULL)
608         {
609             //alocar memoria e copiar a string
610             (*lista_literais)->literal = (char *)malloc(strlen(literal)+1); //TODO : erro alocacao memoria
611             strcpy((*lista_literais)->literal, literal);
612             (*lista_literais)->codigo = 'A'; //o primeiro
613             (*lista_literais)->proximo_codigo = NULL;
614 
615             return((*lista_literais)->codigo);
616         }
617     }
618     else
619     {
620         codigo_atual = (*lista_literais);
621 
622         while(1)
623         {
624             //procura se o ócdigo áj ãno existe
625             if ( !strcmp(codigo_atual->literal, literal))
626                 return(codigo_atual->codigo);
627 
628             if(codigo_atual->proximo_codigo == NULL)
629                 break;
630             else
631                codigo_atual = codigo_atual->proximo_codigo;
632         }
633 
634 
635         if ((codigo_atual->proximo_codigo = (tabela_literais *)malloc(sizeof(tabela_literais))) != NULL)
636         {
637             codigo_atual->proximo_codigo->proximo_codigo = NULL;
638             //alocar memoria e fazer uma stringcopy
```

```
639             codigo_atual->proximo_codigo->literal = (char *)malloc(strlen(literal)+1); //TODO : erro alocacao memoria
640             strcpy(codigo_atual->proximo_codigo->literal, literal);
641             if (codigo_atual->codigo == 'Z') //passar para as letras minusculas
642                 codigo_atual->proximo_codigo->codigo = 'a';
643             else if (codigo_atual->codigo == 'z') //estouro de codigos
644             {
645                 erro(ERRO_004);
646                 return(0);
647             }
648             else
649                 codigo_atual->proximo_codigo->codigo = codigo_atual->codigo + 1;
650             return(codigo_atual->proximo_codigo->codigo);
651         }
652     }
653     //se ocorreu um erro apenas retorna NULL
654     return ('\0');
655 }
656 //destroi lista - ãfunao para destruir todos os ponteiros da lista em caso de erro ou fim do programa
657 void destroi_tabela_literais(tabela_literais *lista_literais)
658 {
659     //percorre a lista recursivamente éat encontrar o ultimo elemento
660     if(lista_literais->proximo_codigo != NULL)
661     {
662         destroi_tabela_literais(lista_literais->proximo_codigo);
663     }
664     //destroi os elementos de forma recursiva.
665     free(lista_literais->literal);
666     free(lista_literais);
667 }
668
669 //percorre a lista imprimindo os tokens encontrados
670 void constroi_tabela_literais(tabela_literais **lista_literais, token *lista_token)
671 {
672     token   *p_lista = lista_token;
673     char codigo;
674
675     while(p_lista != NULL)
676     {
677         if(p_lista->tipo == tipo_literal)
678         {
679             codigo = insere_tabela_literais(lista_literais, p_lista->literal);
680             //tem que trocar a string do token apenas pelo codigo na lista de tokens, senao isso nao faz sentido.
681             p_lista->codigo = codigo;
682 #if defined DEBUG_LEXICO
683             printf("#codigo:%c literal \"%s\"\n\r ",codigo,  p_lista->literal);
684 #endif
685         }
686         p_lista = p_lista->proximo_token;
687     }
688 }
689
690 //Funcoes de manipulacao da pilha
691
692 //destruicao de pilhas
693 void destroi_pilha(pilha_expr *pilha_destruida)
694 {
695     if (pilha_destruida->proximo != NULL)
696         destroi_pilha(pilha_destruida->proximo);
697     free(pilha_destruida);
698 }
699
700 //inserir um elemento na pilha
701 int insere_pilha(pilha_expr **pilha, token *elemento)
702 {
703     pilha_expr *topo_pilha;
704
705     //aloca memoria para o proximo elemento
706     topo_pilha = (pilha_expr*)malloc(sizeof(pilha_expr));
707     if(topo_pilha == NULL)
708         return(0); //houve erro de falta de memoria
709
710     //aponta o elemento da pilha para o token adequado
711     topo_pilha->elemento = elemento;
712
713     //posiciona o novo elemento no topo da pilha
714     topo_pilha->proximo = *pilha;
715     *pilha = topo_pilha;
716
717     return (1); //sucesso
718
719 }
```

```
720
721 //retirar um elemento da pilha
722 token *retira_pilha (pilha_expr **pilha)
723 {
724     token *elemento;
725     pilha_expr *topo_pilha;
726
727     //guarda o token que sera retornado
728     elemento = (*pilha)->elemento;
729
730     //guarda o novo topo da pilha
731     topo_pilha = (*pilha)->proximo;
732
733     //destroi o ponteiro a ser removido
734     free(*pilha);
735
736     //atualiza o topo da pilha
737     *pilha = topo_pilha;
738
739     //retorna o elemento
740     return(elemento);
741 }
742
743 //construir pilha de operandos e operadores
744 token *constroi_lista_expr(token *lista_token)
745 {
746     //crio uma variavel para percorrer a lista de tokens
747   token *p_lista = lista_token;
748     //crio a pilha de operadores
749     pilha_expr *pilha_operadores = NULL;
750     //crio a lista de nos da arvore de expressoes
751     token *lista_expressao = NULL;
752     //flag para deteccao de oeprador negativo unario no inicio da expressao
753     int flag_unario = 1;
754     //enquanto nao chegar ao final da lista, iterar:
755
756     while(p_lista != NULL)
757     {
758         switch(p_lista->tipo)
759         {
760             case tipo_parametro:
761             case tipo_literal: //pilha de operandos
762                 insere_lista_expr(&lista_expressao,p_lista);
763                 break;
764
765             case tipo_operador:
766                 //unario: caso especial. Se encontrou um operador qualquer, e o proximo token e o operador '-', significa
        que e uma operacao unaria,
767                 // e devo modificar o token seguinte
768                 if (p_lista->proximo_token != NULL)
769                     if (p_lista->proximo_token->tipo == tipo_operador)
770                         if (p_lista->proximo_token->operador == operador_menos)
771                             p_lista->proximo_token->operador = operador_negacao;
772                 //unario: devo verificar a flag_unario, caso seja 1, e for um operador '-', significa que e um operador
       unario no inicio da expressao
773                 if (flag_unario && (p_lista->operador == operador_menos))
774                     p_lista->operador = operador_negacao;
775                 //devo verificar a precedencia do operador (quanto menor o valor, maior a precedencia) e inserir na pilha
        diretamente se o novo estiver prioridade maior em relacao ao topo da pilha.
776                 if ((pilha_operadores != NULL) && (prioridade_operador(p_lista) < prioridade_operador(pilha_operadores->
       elemento)))
777                 {
778                     insere_pilha(&pilha_operadores, p_lista);
779                 }
780                 else
781                 //caso contrario, devo remover todos os operadores da pilha de maior prioridade e inserir o novo operador
        na pilha
782                 {
783                     while ((pilha_operadores != NULL) && (prioridade_operador(p_lista) >= prioridade_operador(
       pilha_operadores->elemento)))
784                         insere_lista_expr(&lista_expressao,retira_pilha(&pilha_operadores));
785                     //inserir o novo operador na pilha
786                     insere_pilha(&pilha_operadores, p_lista);
787
788                 }
789                 break;
790
791             case tipo_abre_fecha: //precedencia de parenteses
792                 //unario: caso especial. Se o proximo token e o operador '-', apos abre parenteses, significa que e uma
       operacao unaria,
793                 // e devo modificar o token seguinte
```

```
794                 //abre parenteses e inserido na pilha, mas jamais sera colocado na lista de expressao, serve apenas como
        controle
795                 if (p_lista->abre_fecha == abre_parenteses)
796                 {
797                     if (p_lista->proximo_token != NULL)
798                         if (p_lista->proximo_token->tipo == tipo_operador)
799                             if (p_lista->proximo_token->operador == operador_menos)
800                                 p_lista->proximo_token->operador = operador_negacao;
801                     insere_pilha(&pilha_operadores, p_lista);
802                 }
803                 else if (p_lista->abre_fecha == fecha_parenteses)
804                 //se for fecha parenteses, tenho que retirar todos os operadores ate encontrar o abre-parenteses na pilha
        , e coloca-los na lista
805                 {
806                     while(pilha_operadores->elemento->tipo != tipo_abre_fecha)
807                     {
808                         //retiro um operador da pilha e coloco no fim da lista de expressao
809                         insere_lista_expr(&lista_expressao,retira_pilha(&pilha_operadores));
810 
811                         //se retirou o ultimo elemento da pilha sem encontrar abre-fecha, erro de parentees
812                         if (pilha_operadores == NULL)
813                         {
814                             //erro parenteses mal colocados
815                             return NULL;
816                         }
817                     }
818                     //retiro o abre parenteses que sobrou na pilha
819                     if(pilha_operadores->elemento->abre_fecha == abre_parenteses)
820                         retira_pilha(&pilha_operadores);
821                 }
822                 break;
823 
824         }
825         //limpa o flag_unario, pois ja nao estamos mais no primeiro token.
826         flag_unario = 0;
827         p_lista = p_lista->proximo_token;
828 
829     }
830     //desempilhar todos os operadores restantes
831     while(pilha_operadores != NULL)
832     {
833         insere_lista_expr(&lista_expressao,retira_pilha(&pilha_operadores));
834     }
835     return(lista_expressao);
836 }
837 
838 void imprime_lista_expr(token *lista)
839 {
840     token   *p_lista = lista;
841 
842     //titulo
843     printf("\n A expressao em formato RNP e: ");
844     while(p_lista != NULL)
845     {
846         switch(p_lista->tipo)
847         {
848             case tipo_literal:
849                 printf("%s ", p_lista->literal);
850                 break;
851 
852             case tipo_parametro:
853                 printf("%d ", p_lista->parametro);
854                 break;
855 
856             case tipo_operador:
857             case tipo_abre_fecha:
858 
859                 printf("%c ", p_lista->codigo);
860                 break;
861 
862         }
863         p_lista = p_lista->proximo_token;
864     }
865 }
866 
867 //inserir um elemento na lista encadeada dupla d expressoes
868 int insere_lista_expr(token **lista, token *elemento)
869 {
870     token *p_lista = *lista;
871     token *fim_lista = NULL;
872 
```

```
873     //aloca memoria para o novo elemento
874     fim_lista = (token *)malloc(sizeof(token));
875     if(fim_lista == NULL)
876         return(0); //houve erro de falta de memoria
877 
878     //preenche os campos do elemento
879     memcpy(fim_lista,elemento,sizeof(token));
880     fim_lista->proximo_token = NULL;
881 
882     //se a lista estiver vazia, retornar o elemento criado
883     if (*lista == NULL)
884     {
885         *lista = fim_lista;
886         return(1); //sucesso
887     }
888     else
889     {
890         //percorre a lista ate o final
891         while(p_lista->proximo_token != NULL)
892         {
893             p_lista = p_lista->proximo_token;
894         }
895 
896         //aponta o ponteiro anterior do novo elemento para o final da lista, e vice-versa
897         p_lista->proximo_token = fim_lista;
898 
899         return (1); //sucesso
900     }
901 }
902 
903 //prioridade_operador -> funcao que retorna a precedencia do oeprador, sendo que um numero menor signigfica maior
        prioridade
904 int prioridade_operador(token *elemento)
905 {
906     //se tentar comparar com um token inexistente
907     if(elemento == NULL)
908         return(99);
909     //parenteses sempre tem prioridade maior
910     if (elemento->tipo == tipo_abre_fecha)
911         return (99);
912     else
913         //caso sejam operadores
914         switch(elemento->operador)
915         {
916             case operador_mais:
917             case operador_menos:
918                 return (4);
919             case operador_multiplicacao:
920                 return (3);
921             case operador_negacao:
922                 return(2);
923             case operador_potenciacao:
924                 return(1);
925         }
926     return(0);
927 }
928 
929 //destroi lista - funcao para destruir todos os ponteiros da lista em caso de erro ou fim do programa
930 void destroi_lista_expr( token *lista)
931 {
932     //percorre a lista recursivamente êat encontrar o ultimo elemento
933     if(lista->proximo_token != NULL)
934     {
935         destroi_lista_expr(lista->proximo_token);
936     }
937     lista->literal = NULL;
938     lista->proximo_token = NULL;
939     free(lista);
940 }
941 
942 
943 //funcao que constroi a arvore binaria de expressoes a partir da expressao RPN gerada
944 arvore_expr *constroi_arvore_expr(token *lista)
945 {
946     token *p_lista = lista;
947     pilha_arvore *pilha = NULL;
948     arvore_expr *temp_node = NULL;
949     //percorre a lista da expressao RPN construindo a pilha de arvore
950     while(p_lista != NULL)
951     {
952         switch(p_lista->tipo)
```

```
953         {
954             case tipo_literal:
955             case tipo_parametro:
956                 //para literais ou parametros, os tokens sao inseridos diretamente na pilha
957                 insere_pilha_arvore(&pilha, cria_no_arvore(p_lista));
958                 break;
959             case tipo_operador:
960                 //para operadores, retirar os ultimos 2 nos de arvore, criar uma arvore a partir deles e re-inserir na
        pilha
961 
962                 //construir o node do operador
963                 temp_node = cria_no_arvore(p_lista);
964 
965                 //unario: Para operacoes negacao unaria, preencho apenas o no da direita
966                 if (p_lista->operador == operador_negacao)
967                 {
968                     temp_node->direita = retira_pilha_arvore(&pilha);
969                     temp_node->esquerda = NULL;
970                 }
971                 else
972                 {
973                     //retira os ultimos 2 elementos da pilha para construir a expressao do no
974                     temp_node->direita = retira_pilha_arvore(&pilha);
975                     temp_node->esquerda = retira_pilha_arvore(&pilha);
976                 }
977 
978                 //re-insere o no na pilha
979                 insere_pilha_arvore(&pilha, temp_node);
980 
981                 //reseta o ponteiro de temp_node
982                 temp_node = NULL;
983 
984                 break;
985         }
986         p_lista = p_lista->proximo_token;
987     }
988 
989     //ao final do processo, devera sobrar apenas um elemento na pilha, contnedo a arvore completa da expressao
990     return(retira_pilha_arvore(&pilha));
991 }
992 
993 arvore_expr *cria_no_arvore(token *elemento)
994 {
995     arvore_expr *novo_no;
996     //aloca memoria para o proximo elemento
997     novo_no = (arvore_expr*)malloc(sizeof(arvore_expr));
998     if(novo_no == NULL)
999         return(0); //houve erro de falta de memoria
1000     //aponta o no da pilha para o elemento e inicializa o resto da estrutura node
1001     novo_no->elemento = elemento;
1002     novo_no->direita = NULL;
1003     novo_no->esquerda = NULL;
1004 
1005     return(novo_no);
1006 }
1007 
1008 //inserir um elemento na pilha
1009 int insere_pilha_arvore(pilha_arvore **pilha, arvore_expr *elemento)
1010 {
1011     pilha_arvore *topo_pilha = NULL;
1012 
1013     topo_pilha = (pilha_arvore *)malloc(sizeof(pilha_arvore));
1014     if (topo_pilha == NULL)
1015         return (0); //erro
1016     //cria um novo elemento na pilha
1017 
1018     //posiciona o novo elemento no topo da pilha
1019     topo_pilha->node = elemento;
1020     topo_pilha->proximo = *pilha;
1021     *pilha = topo_pilha;
1022     return (1); //sucesso
1023 
1024 }
1025 
1026 //retirar um elemento da pilha
1027 arvore_expr *retira_pilha_arvore (pilha_arvore **pilha)
1028 {
1029     arvore_expr *elemento;
1030     pilha_arvore *topo_pilha;
1031 
1032     if (*pilha != NULL)
```

```
1033     {
1034         //guarda o token que sera retornado
1035         elemento = (*pilha)->node;
1036
1037         //guarda o novo topo da pilha
1038         topo_pilha = (*pilha)->proximo;
1039
1040         //desaloca memoria da estrutura da pilha
1041         free(*pilha);
1042
1043         //atualiza o topo da pilha
1044         *pilha = topo_pilha;
1045
1046         //retorna o elemento
1047         return(elemento);
1048     }
1049     else
1050         return (NULL);
1051 }
1052
1053 //funcao que desaloca a memoria da arvore
1054 void destroi_arvore_expr(arvore_expr *arvore)
1055 {
1056
1057     if (arvore!=NULL)
1058     {
1059         //percorre sub-arvore esquerda
1060         destroi_arvore_expr(arvore->esquerda);
1061         //percorre sub-arvore direita
1062         destroi_arvore_expr(arvore->direita);
1063
1064         //apos ter desalocado todas as sub-arvores, desaloca o proprio no
1065         free((void*)arvore);
1066     }
1067 }
1068
1069 //funcao que imprime a expressao da arvore
1070 void imprime_arvore_expr(arvore_expr *arvore)
1071 {
1072     if (arvore!=NULL)
1073     {
1074         if(arvore->elemento->tipo == tipo_operador)
1075         {
1076             //cada sub-expressao estara contida em parenteses
1077             printf("( ");
1078         }
1079         //percorre a sub-arvore da esquerda
1080         imprime_arvore_expr(arvore->esquerda);
1081         //imprime o caracter do token
1082         switch(arvore->elemento->tipo)
1083         {
1084             case tipo_literal:
1085                 printf("%s", arvore->elemento->literal);
1086                 break;
1087
1088             case tipo_parametro:
1089                 printf("%d", arvore->elemento->parametro);
1090                 break;
1091
1092             case tipo_operador:
1093                 printf(" %c ", arvore->elemento->codigo);
1094                 break;
1095         }
1096         //percorre a sub-arvore da direita
1097         imprime_arvore_expr(arvore->direita);
1098          if(arvore->elemento->tipo == tipo_operador)
1099         {
1100             //cada sub-expressao estara contida em parenteses
1101             printf(" )");
1102         }
1103     }
1104 }
1105
1106 //funcao que reordena um astring em ordem alfabetica
1107 void string_sort(char **input)
1108 {
1109   char *p_input;
1110   char swap;
1111   int flag = 0;
1112   //o loop so para quando nao houve reordenacao alguma
1113   do
```

```
1114   {
1115     //aponta para o inicio da string
1116     p_input = *input;
1117
1118     //reset na flag de controle do loop
1119     flag = 0;
1120
1121     //percorre a string ate chegar ao final
1122     while(*(p_input + 1) != '\0')
1123     {
1124
1125       //compara termos adjacentes
1126       if ((*p_input) > *(p_input+1))
1127       {
1128         //realiza a troca entre os termos adjacentes, caso o anterior seja maior
1129         swap = *p_input;
1130         *p_input = *(p_input+1);
1131         *(p_input + 1) = swap;
1132
1133         //sinaliza que houve troca
1134         flag = 1;
1135       }
1136             //incrementa o ponteiro para proxima comparacao
1137             p_input++;
1138     }
1139   } while(flag);
1140 }
1141
1142 //funcao que imprime a expressao da arvore
1143 lista_expr *constroi_lista_expressoes_exp(arvore_expr *arvore)
1144 {
1145     lista_expr *no_esq; //toda a expressao representada pelo no esquerdo
1146     lista_expr *no_dir; //toda a expressao representada pelo no direito
1147     lista_expr *elemento_lista = NULL; //elemento novo a ser alocado
1148     lista_expr *elemento_ant = NULL; //referencia para o ultimo elemento criado
1149     lista_expr *p_elemento_dir; //ponteiro para percorrer a lista da sub-arvore direita
1150     int i;  //indice de proposito geral
1151
1152     if (arvore!=NULL)
1153     {
1154         //percorre a sub-arvore da esquerda
1155         no_esq = constroi_lista_expressoes_exp(arvore->esquerda);
1156         //percorre a sub-arvore da direita
1157         no_dir = constroi_lista_expressoes_exp(arvore->direita);
1158         //Realiza operacoes dependendo do tipo do no
1159         switch(arvore->elemento->tipo)
1160         {
1161             //se chegou ate um literal ou um parametro, e uma folha da arvore, portanto devemos alocar o elemento da
       lista de expressoes expandidas
1162             case tipo_literal:
1163                 if((elemento_lista = (lista_expr *)malloc(sizeof(lista_expr))) == NULL)
1164                 {
1165                     //erro memoria insuficiente
1166                     return(NULL);
1167                 }
1168                 //aloca apenas 2 posicoes, uma para a variavel e outra para o \0
1169                 if((elemento_lista->codigos_numerador = (char *)malloc(2*sizeof(char))) == NULL)
1170                 {
1171                     //erro memoria insuficiente
1172                     return(NULL);
1173                 }
1174                 //copia o codigo e o \0 na string
1175                 elemento_lista->codigos_numerador[0] = arvore->elemento->codigo;
1176                 elemento_lista->codigos_numerador[1] = '\0';
1177                 elemento_lista->codigos_denominador = NULL;
1178                 //por padrao, todos os termos sao positivos
1179                 //elemento_lista->sinal = operador_mais;
1180                 //a variavel inicia-se multiplicada por 1
1181                 elemento_lista->parametro = 1.0;
1182                 elemento_lista->proximo = NULL;
1183                 elemento_lista->anterior = NULL;
1184                 break;
1185
1186             case tipo_parametro:
1187                 if((elemento_lista = (lista_expr *)malloc(sizeof(lista_expr))) == NULL)
1188                 {
1189                     //erro memoria insuficiente
1190                     return(NULL);
1191                 }
1192                 //escalares nao tem nenhuma lista de codigos associado
1193                 elemento_lista->codigos_numerador = NULL;
```

```
1194                elemento_lista->codigos_denominador = NULL;
1195                //por padrao, todos os termos sao positivos
1196               // elemento_lista->sinal = operador_mais;
1197                //insere-se o parametro ja transformado para inteiro
1198                elemento_lista->parametro = (double)(arvore->elemento->parametro);
1199                elemento_lista->proximo = NULL;
1200                elemento_lista->anterior = NULL;
1201                break;
1202
1203            //aqui estara a maior complexidade do codigo, onde as expansoes serao de fato realizadas
1204            case tipo_operador:
1205                switch(arvore->elemento->operador)
1206                {
1207                    case operador_mais:
1208                        //o operador positivo apenas concatena a lista do no esquerdo com a lista do no direito
1209                        //se o no esquerdo for nulo, significa que ha apenas uma operacao unaria, entao nao deve haver
        concatenacao
1210                        //aponta para o final da lista do no esquerdo
1211                        elemento_lista = no_esq;
1212                        while (elemento_lista->proximo != NULL)
1213                            elemento_lista = elemento_lista->proximo;
1214
1215                        //concatenacao das expressoes
1216                        elemento_lista->proximo = no_dir;
1217                        no_dir->anterior = elemento_lista;
1218
1219                        //recupera o inicio da lista
1220                        while (elemento_lista->anterior != NULL)
1221                            elemento_lista = elemento_lista->anterior;
1222                        break;
1223
1224                    case operador_menos:
1225                        //o operador negativo troca todos os sinais dos elementos da sub-arvore direita e depois
        concatena como na adicao
1226                        p_elemento_dir = no_dir;
1227                        while(p_elemento_dir != NULL)
1228                        {
1229                            /*if(p_elemento_dir->sinal == operador_mais)
1230                                p_elemento_dir->sinal = operador_menos;
1231                            else
1232                                p_elemento_dir->sinal = operador_mais;*/
1233                            p_elemento_dir->parametro *= -1.0;
1234
1235                            p_elemento_dir = p_elemento_dir->proximo;
1236                        }
1237
1238                        //aponta para o final da lista do no esquerdo
1239                        elemento_lista = no_esq;
1240                        while (elemento_lista->proximo != NULL)
1241                            elemento_lista = elemento_lista->proximo;
1242
1243                        //concatenacao das expressoes
1244                        elemento_lista->proximo = no_dir;
1245                        no_dir->anterior = elemento_lista;
1246
1247                        //recupera o inicio da lista
1248                        while (elemento_lista->anterior != NULL)
1249                            elemento_lista = elemento_lista->anterior;
1250                        break;
1251
1252                    case operador_negacao:
1253                        //o operador negacao apenas troca todos os sinais dos elementos da sub-arvore direita
1254                        p_elemento_dir = no_dir;
1255                        while(p_elemento_dir != NULL)
1256                        {
1257                           /* if(p_elemento_dir->sinal == operador_mais)
1258                                p_elemento_dir->sinal = operador_menos;
1259                            else
1260                                p_elemento_dir->sinal = operador_mais;*/
1261                            p_elemento_dir->parametro *= -1.0;
1262
1263                            p_elemento_dir = p_elemento_dir->proximo;
1264                        }
1265
1266                        //como nao ha operando na no esquerdo, nao ha o que concatenar
1267                        elemento_lista = no_dir;
1268                        break;
1269
1270                    case operador_multiplicacao:
1271
1272                        // o procedimento foi encapsulado para ser utilizado na potenciacao
```

```
1273                        elemento_lista = multiplica_expr(no_esq,no_dir);
1274
1275                        //desalocar as listas originais
1276                        destroi_lista_expr_expandida(no_dir);
1277                        destroi_lista_expr_expandida(no_esq);
1278
1279                        //erro
1280                        if (elemento_lista == NULL)
1281                        {
1282                            //erro
1283                            return(NULL);
1284                        }
1285                        break;
1286
1287                    case operador_potenciacao:
1288                        //primeiramente verificar se o expoente e um numero inteiro
1289                        if (no_dir->codigos_numerador != NULL)
1290                        {
1291                            //erro - expressao no expoente
1292                            return(NULL);
1293                        }
1294
1295                        //TODO divisao: tambem retornar erro se o expoente for negativo
1296                       // if (no_dir->sinal == operador_menos)
1297                        if (no_dir->parametro < 0.0)
1298                        {
1299                            //erro - expoente negativo
1300                            return(NULL);
1301                        }
1302
1303                        // se o expoente for 0, retornar 1
1304                        if (no_dir->parametro == 0.0)
1305                        {
1306                            if((elemento_lista = (lista_expr *)malloc(sizeof(lista_expr))) == NULL)
1307                            {
1308                                //erro memoria insuficiente
1309                                return(NULL);
1310                            }
1311                            //escalares nao tem nenhuma lista de codigos associado
1312                            elemento_lista->codigos_numerador = NULL;
1313                            elemento_lista->codigos_denominador = NULL;
1314                            //por padrao, todos os termos sao positivos
1315                            //elemento_lista->sinal = operador_mais;
1316                            //insere-se o numero 1 no campo do parametro
1317                            elemento_lista->parametro = 1.0;
1318                            elemento_lista->proximo = NULL;
1319                            elemento_lista->anterior = NULL;
1320
1321                            //desalocar as listas do no direito e esquerdo
1322                            destroi_lista_expr_expandida(no_dir);
1323                            destroi_lista_expr_expandida(no_esq);
1324
1325                        } //se o expoente for 1, retornar o proprio no esquerdo
1326                        else if (no_dir->parametro == 1.0)
1327                        {
1328                            elemento_lista = no_esq;
1329
1330                            //desalocar o no direito
1331                            destroi_lista_expr_expandida(no_dir);
1332
1333                        }
1334                        else //multiplicar o no esquerdo por si mesmo quantas vezes for o parametro do no direito
1335                        {
1336                            elemento_ant = no_esq;
1337                            for (i=0;i < (int)(no_dir->parametro - 1.0);i++)
1338                            {
1339                                elemento_lista = multiplica_expr(elemento_ant, no_esq);
1340
1341                                //desaloca o elemento anterior, mas apenas se nao for o no_Esq original
1342                                if (elemento_ant != no_esq)
1343                                    destroi_lista_expr_expandida(elemento_ant);
1344
1345                                //aponta o elemento anterior para o atual
1346                                elemento_ant = elemento_lista;
1347                            }
1348                        }
1349
1350                        //desaloca as listas originais
1351                        destroi_lista_expr_expandida(no_dir);
1352                        destroi_lista_expr_expandida(no_esq);
1353                        break;
```

```
1354                }
1355                break;
1356        }
1357    }
1358    return(elemento_lista);
1359 }
1360
1361 //realiza multiplicacao entre duas listas de expressao
1362 lista_expr *multiplica_expr( lista_expr *no_esq, lista_expr *no_dir)
1363 {
1364
1365    lista_expr *elemento = NULL; //elemento novo a ser alocado
1366    lista_expr *elemento_ant = NULL; //referencia para o ultimo elemento criado
1367    lista_expr *p_elemento_esq; //ponteiro para percorrer a lista da sub-arvore esquerda
1368    lista_expr *p_elemento_dir; //ponteiro para percorrer a lista da sub-arvore direita
1369    int temp;
1370
1371    //aqui deve-se combinar todos os elementos da lista esquerda com os da lista direita, gerando uma nova lista
1372    //no processo que devera ser desalocada ao final da operacao.
1373    p_elemento_esq = no_esq;
1374    p_elemento_dir = no_dir;
1375
1376    //caso algum dos termos seja um ponteiro nulo
1377    if (no_esq == NULL || no_dir == NULL)
1378        return NULL;
1379
1380    while (p_elemento_esq != NULL)
1381    {
1382        while (p_elemento_dir != NULL)
1383        {
1384            //reseta o ponteiro elemento para a etapa
1385            elemento = NULL;
1386            //aloco cada elemento novo
1387            if((elemento = (lista_expr *)malloc(sizeof(lista_expr))) == NULL)
1388            {
1389                //erro memoria insuficiente
1390                return(NULL);
1391            }
1392
1393            //inicializar ponteiros
1394            elemento->proximo = NULL;
1395            elemento->anterior = NULL;
1396
1397            //TODO: DIVISAO implementar as operacoes cabiveis com os denominadores tambem
1398            //vejo a quantidade de variaveis em cada termo da multiplicacao e aloco uma string adequada
1399            if ((p_elemento_esq->codigos_numerador != NULL) || (p_elemento_dir->codigos_numerador != NULL))
1400            {
1401                temp = 0; //acumulador de tamanho de string
1402                if (p_elemento_esq->codigos_numerador != NULL)
1403                {
1404                    temp += strlen(p_elemento_esq->codigos_numerador);
1405                }
1406
1407                if (p_elemento_dir->codigos_numerador != NULL)
1408                {
1409                    temp += strlen(p_elemento_dir->codigos_numerador);
1410                }
1411
1412                elemento->codigos_numerador = (char *)malloc(sizeof(char)*(1 + temp));
1413                *(elemento->codigos_numerador) = '\0';
1414                elemento->codigos_denominador = NULL;
1415                //copio a string das variaveis do primeiro elemento
1416                if (p_elemento_esq->codigos_numerador != NULL)
1417                    strcpy(elemento->codigos_numerador,p_elemento_esq->codigos_numerador);
1418                //e concateno com os do segundo
1419                if (p_elemento_dir->codigos_numerador != NULL)
1420                    strcat(elemento->codigos_numerador,p_elemento_dir->codigos_numerador);
1421                //finalmente, reordeno a nova string
1422                string_sort(&(elemento->codigos_numerador));
1423            }
1424            else //multiplicacao entre 2 parametros
1425            {
1426                elemento->codigos_numerador = NULL;
1427            }
1428
1429            //inicializar o ponteiro do denominador
1430            elemento->codigos_denominador = NULL;
1431
1432            //multiplicar os parametros
1433            elemento->parametro = p_elemento_esq->parametro * p_elemento_dir->parametro;
1434
```

```
1435             //atualizar os ponteiros
1436             if(elemento_ant == NULL)
1437                 elemento_ant = elemento;
1438             else
1439             {
1440                 elemento_ant->proximo = elemento;
1441                 elemento->anterior = elemento_ant;
1442                 elemento_ant = elemento;
1443             }
1444 
1445             //incrementa o ponteiro da lista do no da direita
1446             p_elemento_dir = p_elemento_dir->proximo;
1447         }
1448         //avanco o ponteiro da lista do no esquerdo
1449         p_elemento_esq = p_elemento_esq->proximo;
1450 
1451         //reseto o ponteiro da sub-arvore esquerda
1452         p_elemento_dir = no_dir;
1453     }
1454     //aproveitando que a lista e duplamente encadeada, eu posso recuperar o inicio atraves dos ponteiros para anterior
1455     while (elemento->anterior != NULL)
1456         elemento = elemento->anterior;
1457     return (elemento);
1458 }
1459 
1460 //desalocar um lista de expressoes
1461 void destroi_lista_expr_expandida(lista_expr *lista)
1462 {
1463     //teste de consistencia
1464     if (lista == NULL)
1465         return;
1466 
1467     //chamadas recursivas
1468     if (lista->proximo != NULL)
1469        destroi_lista_expr_expandida(lista->proximo);
1470 
1471     //desalocar cada estrutura da lista
1472     if (lista->codigos_denominador != NULL)
1473         destroi_lista_expr_expandida(lista->codigos_denominador);
1474     if (lista->codigos_numerador != NULL)
1475         free(lista->codigos_numerador);
1476     //desalocar o no
1477     free(lista);
1478 
1479 }
1480 
1481 void imprime_lista_expr_expandida(lista_expr *lista, tabela_literais *tabela)
1482 {
1483     lista_expr   *p_lista = lista;
1484     double teste;
1485 
1486     //imprime todos os campos do elemento
1487     while(p_lista != NULL)
1488     {
1489         //imprime o sinal
1490         if (p_lista->parametro > 0.0)
1491             printf("+");
1492         else if (p_lista->parametro == -1.0)
1493             printf("-");
1494 
1495         //imprime o parametro
1496         if (p_lista->parametro != 1.0 && p_lista->parametro != -1.0)
1497             //checar se o coeficiente e inteiro
1498             if ((teste = (int)p_lista->parametro - p_lista->parametro) == 0.0)
1499                 printf("%d",(int)(p_lista->parametro));
1500             else
1501             {
1502                 printf("%2.2f",p_lista->parametro);
1503             }
1504 
1505         //imprime o numerador
1506         if (p_lista->codigos_numerador != NULL)
1507             //printf("%s", p_lista->codigos_numerador);
1508             print_monomio(p_lista->codigos_numerador, tabela);
1509         else  if (p_lista->parametro == 1.0 || p_lista->parametro == -1.0)
1510             printf("1.00");
1511 
1512         //imprime o denominador
1513         if (p_lista->codigos_denominador != NULL)
1514             //printf("/(%s)",p_lista->codigos_denominador);
1515             imprime_lista_expr_expandida(p_lista->codigos_denominador, tabela);
```

```
1516
1517        //imprime um espaco para o proximo elemento
1518        printf(" ");
1519
1520        //imprime o proximo elemento
1521        p_lista = p_lista->proximo;
1522    }
1523 }
1524
1525 //funcao que imprime o monomio segundo suas variaveis a partir dos codigos internos
1526 void print_monomio(char *monomio, tabela_literais *tabela)
1527 {
1528    int i;
1529    tabela_literais *p_tabela;
1530
1531    //percorrer os codigos dos monomios, contando quantos tem igual, e imprimindo com expoente quando for o caso
1532    while (*monomio != '\0')
1533    {
1534        i = 1; //primeira variavel
1535        //testar se os proximos codigos sao iguais
1536        while (*monomio == *(monomio + 1))
1537        {
1538            i++; //incrementa o numero de variaveis iguais
1539            monomio++;
1540        }
1541        //inicializa a tabela de literias
1542        p_tabela = tabela;
1543        //Pesquisa o codigo na tabela de literais
1544        while (p_tabela->codigo != *monomio)
1545            p_tabela = p_tabela->proximo_codigo;
1546
1547        //imprime a variavel correspondente ao codigo
1548        printf("%s",p_tabela->literal);
1549
1550        //imprime o expoente
1551        if (i > 1)
1552            printf("^%d",i);
1553        //proximo codigo
1554        monomio++;
1555
1556        //adicionar um sinal de multiplicacao caso o proximo nao seja o \0
1557        if (*monomio != '\0')
1558            printf("*");
1559    }
1560
1561 }
1562
1563 // funcao que agrupa elementos iguais da expressao expandida
1564 lista_expr *simplifica_expr_expandida(lista_expr *p_lista)
1565 {
1566    lista_expr *p_1, *p_2, *p_remove, *p_inicio;  //ponteiros para percorrer a lista
1567    double resultado;          //avaliacao da soma ou subtracao dos termos
1568
1569    //realiza uma copia de p_lista
1570    //TODO: controle de erro
1571    if (p_lista == NULL)
1572        return NULL;
1573
1574    p_inicio = copia_lista_expr(p_lista);
1575
1576    //inicializa os ponteiros
1577    p_1 = p_inicio;
1578
1579    if (p_1->proximo == NULL)
1580    {
1581        //a expressao so tem um elemento
1582        return(p_1);
1583    }
1584
1585    //loop de comparacao
1586    while (p_1 != NULL)
1587    {
1588        //inicializar p_2 como o segundo elemento
1589        p_2 = p_1->proximo;
1590        while (p_2 != NULL)
1591        {
1592            //verificar se o conjunto de variaveis dos dois elementos sao iguais
1593            // TODO divisao: levar em conta o denominador tambem
1594            if (p_1->codigos_numerador != NULL && p_2->codigos_numerador != NULL)
1595            {
1596                if (!strcmp(p_1->codigos_numerador, p_2->codigos_numerador))
```

```
1597                {
1598                    //inicializar a variavel de resultado
1599                    resultado = 0.0;
1600                    //somar ou subtrair o primeiro termo
1601                    resultado+= p_1->parametro;
1602
1603                    //somar ou subtrair o segundo termo
1604                    resultado+= p_2->parametro;
1605
1606                    //BUG: arredondar quando o resultado e residual.Por exemplo, 0.9 - 0.9 = 11.1E-15
1607
1608                    if ((resultado <= 0.0000000000001 && resultado >= 0.0) || (resultado >= -0.0000000000001 && resultado
        <= 0.0))
1609                        resultado = 0.0;
1610                    //guardar o resultado em p_1 sem sinal
1611                    p_1->parametro = resultado;
1612
1613                    //ja atualiza p_2
1614                    p_remove = p_2;
1615                    p_2 = p_2->proximo;
1616                    //remover p_2 da lista
1617                    //TODO: implementar a funcao
1618                    p_inicio = remove_lista_expr(p_inicio, p_remove);
1619                }
1620                else //atualiza p_2
1621                    p_2 = p_2->proximo;
1622            }
1623            else //atualiza p_2
1624                p_2 = p_2->proximo;
1625
1626        }
1627        //atualiza p_1
1628        p_1 = p_1->proximo;
1629    }
1630    //percorre a lista novamente elimienando todos os resultados 0, deixando somente um caso nao sobre elementos na lista
1631    p_1 = p_inicio;
1632    while (p_1 != NULL)
1633    {
1634        //se sobrou apenas um elemento (p_1 == p_inicio) e  (p_1->proximo == NULL) entao nao descarta-lo
1635        if ((p_1->parametro == 0.0) && (p_1 == p_inicio) && (p_1->proximo == NULL))
1636        {
1637            //desalocar os codigos se houver algum
1638            if(p_1->codigos_numerador != NULL)
1639            {
1640                free(p_1->codigos_numerador);
1641                p_1->codigos_numerador = NULL;
1642            }
1643            if(p_1->codigos_denominador != NULL)
1644            {
1645                //free(p_1->codigos_denominador);
1646                destroi_lista_expr_expandida(p_1->codigos_denominador);
1647                p_1->codigos_denominador = NULL;
1648            }
1649
1650            p_1 = p_1->proximo;
1651
1652        }
1653        //caso o parametro seja 0 em outras circunstancias
1654        else if (p_1->parametro == 0.0)
1655        {
1656            //atualiza p_1
1657            p_remove = p_1;
1658            p_1 = p_1->proximo;
1659            //remove p_1 da lista
1660            p_inicio = remove_lista_expr(p_inicio, p_remove);
1661        }
1662        else
1663            p_1 = p_1->proximo; //nao e zero
1664    }
1665    return p_inicio;
1666 }
1667
1668 //funcao que remove um ponteiro da lista de expressoes expandida
1669 lista_expr *remove_lista_expr(lista_expr *p_inicio, lista_expr *p_remove)
1670 {
1671    lista_expr *p_lista; //ponteiro para percorrer a lista
1672
1673    //inicializa p_lista
1674    p_lista = p_inicio;
1675
1676    //percorre a lista a procura de p_remove
```

```
1677    while (p_lista != p_remove && p_lista != NULL)
1678        p_lista = p_lista->proximo;
1679
1680    //se nao encontrou o elemento, retorna simplesmente o inicio da lista denovo
1681    if (p_lista == NULL)
1682    {
1683        return p_inicio;
1684    }
1685
1686    //se o ponteiro a ser removido esta no inicio da lista atualizar p_inicio antes da remocao
1687    if (p_lista == p_inicio)
1688    {
1689        p_inicio = p_inicio->proximo;
1690    }
1691
1692    //atualizar o pontiero proximo do elemento anterior a p_remove
1693    if (p_remove->anterior != NULL)
1694        (p_remove->anterior)->proximo = p_remove->proximo;
1695
1696    //atualizar o ponteiro anterior do elemento proximo a p_remove
1697    if (p_remove->proximo != NULL)
1698        (p_remove->proximo)->anterior = p_remove->anterior;
1699
1700    //liberar memoria dos elementos internos de p_remove
1701    if (p_remove->codigos_numerador != NULL)
1702        free(p_remove->codigos_numerador);
1703
1704    if (p_remove->codigos_denominador != NULL)
1705       // free(p_remove->codigos_denominador);
1706        destroi_lista_expr_expandida(p_remove->codigos_denominador);
1707
1708    //desalocar p_remove
1709    free(p_remove);
1710
1711    //retornar o inicio da lista
1712    return p_inicio;
1713 }
1714
1715 lista_expr *copia_lista_expr(lista_expr *lista)
1716 {
1717    lista_expr *p_lista_prox = NULL, *p_lista;
1718
1719    //chamadas recursivas
1720    if (lista->proximo != NULL)
1721        p_lista_prox = copia_lista_expr(lista->proximo);
1722
1723    //alocar memoria para o no:
1724    //TODO controle de erro
1725    p_lista = (lista_expr *)malloc(sizeof(lista_expr));
1726
1727    //inicializa os elementos
1728    p_lista->anterior = NULL;
1729    p_lista->proximo = NULL;
1730    p_lista->codigos_denominador = NULL;
1731
1732    //copia os elementos
1733    p_lista->parametro = lista->parametro;
1734    //p_lista->sinal = lista->sinal;
1735    if (lista->codigos_numerador != NULL)
1736    {
1737        //TODO controle de erro
1738        p_lista->codigos_numerador = (char *)malloc((strlen(lista->codigos_numerador)+1)*sizeof(char));
1739        strcpy(p_lista->codigos_numerador, lista->codigos_numerador);
1740    }
1741    else
1742        p_lista->codigos_numerador = NULL;
1743
1744    if (lista->codigos_denominador != NULL)
1745    {
1746        //TODO controle de erro
1747        p_lista->codigos_denominador = copia_lista_expr(lista->codigos_denominador);
1748    }
1749    else
1750        p_lista->codigos_denominador = NULL;
1751
1752    //atualiza os ponteiros de anterior e proximo
1753    p_lista->proximo = p_lista_prox;
1754    if (p_lista_prox != NULL)
1755        p_lista_prox->anterior = p_lista;
1756
1757    return p_lista;
```

```
1758
1759 }
1760
1761 //reordena um polinomio na forma lexdeg
1762 lista_expr *lexdegbubblesort(lista_expr *poly)
1763 {
1764     lista_expr *p_atual;    //ponteiro para elemento atual do polinomio
1765     lista_expr *p_troca;    //ponteiro auxiliar para troca
1766     int flag_troca = 0;     //flag que indica se houve alguma troca de elementos
1767
1768     if (poly == NULL)
1769         return NULL;
1770     //percorre a lista de polinomios e troca os elementos 1 a 1 quando for necessario. So para quando a percorrer o
         polinomio inteiro e nao fizer troca alguma
1771     do
1772     {
1773         //reseta o flag de flag_troca
1774         flag_troca = 0;
1775         //aponta para o inicio do polinomio
1776         p_atual = poly;
1777         //rebobina para o inicio do polinomio
1778         while (p_atual->anterior != NULL)
1779             p_atual = p_atual->anterior;
1780         //percorre a lista
1781         while (p_atual->proximo != NULL)
1782         {
1783             p_troca = p_atual->proximo;
1784
1785             //compara dois monomios adjacentes. Se o proximo tem precedencia lexdeg, e feita uma troca entre os monomios
         na lista do polinomio
1786             if (lexdeg(p_atual->codigos_numerador,p_troca->codigos_numerador))
1787             {
1788                 //sinaliza que houve troca
1789                 flag_troca = 1;
1790                 //realiza a troca
1791                 //ajusta os ponteiros de borda
1792                 p_atual->proximo = p_troca->proximo;
1793                 if (p_atual->proximo != NULL)
1794                     p_atual->proximo->anterior = p_atual;
1795
1796                 p_troca->anterior = p_atual->anterior;
1797                 if(p_troca->anterior != NULL)
1798                     p_troca->anterior->proximo = p_troca;
1799
1800                 //ajusta os ponteiros entre p_atual e p_troca
1801                 p_troca->proximo = p_atual;
1802                 p_atual->anterior = p_troca;
1803             }
1804             else
1805                 p_atual = p_atual->proximo;
1806         }
1807     }while (flag_troca == 1);
1808
1809     //recuperar o inicio do polinomio
1810     while (poly->anterior != NULL)
1811     {
1812         poly = poly->anterior;
1813     }
1814     return poly;
1815 }
1816
1817 //lexdeg funcao que compara dois monomios e retorna 1 caso o segundo argumento tenha precedencia lexdeg em relacao ao
         primeiro
1818 int lexdeg(char *primeiro, char *segundo)
1819 {
1820     char parametro = 'A'; //parametro de comparacao lexica
1821     char *p_deg;    //ponteiros para comparacao de grau monomial
1822     int deg_1, deg_2;           //grau dos monomios
1823
1824     //se o segundo monomio for uma constante, retornar 0
1825     if (segundo == NULL)
1826         return 0;
1827     //se o primeiro for uma constante, retornar 1
1828     else if (primeiro == NULL)
1829         return 1;
1830
1831     //loop infinito de procura, o retorno sera efetuado dentro das condicionais
1832     while (1)
1833     {
1834         //procura o parametro no segundo monomio
1835         if ((p_deg = strchr(segundo, parametro)) != NULL)
```

```
1836        {
1837            //procuro o parametro no primeiro
1838            if ((p_deg = strchr(primeiro, parametro)) != NULL)
1839            {
1840                //se o argumento prioritario foi encontrado em ambos monomios, comparar o grau dos monomios para o mesmo
     parametro
1841                //encontrar o grau dos monomios
1842                p_deg = primeiro;   //inicializa o ponteiro
1843                //inicializa o grau
1844                deg_1 = 0;
1845                while (p_deg!= NULL)
1846                {
1847                    //a cada vez que encontra o parametro, refazer a busca a partir do proximo caracter
1848                    p_deg = strchr(p_deg, parametro);
1849                    if (p_deg != NULL)
1850                    {
1851                        deg_1 += 1;
1852                        //atualiza o ponteiro da string
1853                        p_deg+=1;
1854                    }
1855                }
1856
1857                //repetir o procedimento para encontrar o grau do segundo monomio para o presente parametro
1858                p_deg = segundo;   //inicializa o ponteiro
1859                //inicializa o grau
1860                deg_2 = 0;
1861                while (p_deg!= NULL)
1862                {
1863                    //a cada vez que encontra o parametro, refazer a busca a partir do proximo caracter
1864                    p_deg = strchr(p_deg, parametro);
1865                    if (p_deg != NULL)
1866                    {
1867                        deg_2 += 1;
1868                        //atualiza o ponteiro da string
1869                        p_deg+=1;
1870                    }
1871                }
1872
1873                //de posse dos graus dos monomios, comparar
1874                if (deg_2 > deg_1)
1875                {
1876                    //no caso do grau do segundo monomio ser maior, retornar 1
1877                    return 1;
1878                }
1879                else if (deg_2 < deg_1)
1880                {
1881                    //no caso do grau do segundo monomio ser menor que o primeiro, retornar 0
1882                    return 0;
1883                }
1884
1885                //caso contrario, se o grau for identico para a mesma variavel em ambos monomios, nao faco mais nada e
     atualizo o parametro
1886                //para o proximo loop
1887
1888            }
1889            //se o argumento prioritario foi encontrado apenas no segundo monomio, retornar 1
1890            else
1891                return 1;
1892
1893        }
1894        else if ((p_deg = strchr(primeiro, parametro)) != NULL)
1895        {
1896            //se encontrou o parametro prioritario apenas no primeiro monomio, retornar 0.
1897            return 0;
1898        }
1899
1900        //caso nao tenha encontrado o parametro prioriatario em nenhum monomio, ou caso o grau da variavel relatica ao
     parametro atual for igual
1901        //em ambos monomios, refazer a comparacao com o proximo parametro.
1902
1903        //se ja esta na letra Z, passar a busca para "a" minusculo
1904        if (parametro == 'Z')
1905        {
1906            parametro = 'a';
1907        }
1908        else
1909            parametro += 1;
1910
1911        //caso nao tenha encontrado variavel alguma valida, retornar 0
1912        if (parametro > 'z')
1913        {
```

```
1914             return(0);
1915             //erro
1916         }
1917     }
1918 }
1919
1920 //funcao que realiza a divisao polinomial.
1921 int polydiv(lista_expr *dividendo, lista_expr *divisor, lista_expr **quociente, lista_expr **resto)
1922 {
1923     lista_expr *dividendo_cpy = NULL;
1924     lista_expr *resultado_monomio = NULL;
1925     lista_expr *poly_ptr;
1926     lista_expr *resto_temp;
1927     lista_expr *quociente_temp;
1928
1929     //inicializa variaveis de retorno
1930     *quociente = NULL;
1931     *resto = NULL;
1932
1933     //se passar um dividendo 0 ou inexistente, devo retornar imediatamente
1934     if (dividendo == NULL)
1935     {
1936         *quociente = constroi_elemento_zerado();
1937         *resto = constroi_elemento_zerado();
1938
1939         return 1;
1940     } //se o dividendo possui apenas um monomio de valor 0
1941     else if (dividendo->proximo == NULL && dividendo->parametro == 0)
1942     {
1943         *quociente = constroi_elemento_zerado();
1944         *resto = constroi_elemento_zerado();
1945
1946         return 1;
1947     }
1948
1949     //inicializar resto e quociente
1950     quociente_temp = NULL;
1951     resto_temp = NULL;
1952     //devo copiar o dividendo a cada operacao, visto que ela pode falhar e eu ter que recomecar
1953     dividendo_cpy = copia_lista_expr(dividendo); //este passo pode deixar o algoritmo bem pesado. Posso otimizar para que
           a lista vire um bloco de memoria, e
1954                                                //realizar a copia via memcpy.
1955
1956     //1) Compara-se o Lt(dividendo) com o Lt(divisor) -> Leading term.
1957     //2) Se a string do LT(divisor) esta contida na Lt(dividendo), -> cuidado: AAABBBCCC = A^3*B^3*C^3, e ABC. A segunda
          divide a primeira, mas nao ha string ABC na primeira! encontro a string que falta e crio um termo novo.
1958     //o resultado_monomio e a string que falta no divisor para chegar ao dividendo. Se o divisor nao estiver totalmente
          contido no dividendo, ela retorna null, ou seja, o LT do dividendo ja comeca a integrar o resto.
1959
1960     //repetir a procedimento ate o dividendo ficar vazio
1961     while (dividendo_cpy != NULL)
1962     {
1963         //divide os LT's do dividendo e divisor, guardando o resultado em resultado_monomio
1964         resultado_monomio = divide_monomio(dividendo_cpy,divisor);
1965
1966         //condicao de erro
1967         if (resultado_monomio == NULL)
1968         {
1969             //erro, divisao por 0
1970             //limpar variaveis
1971             destroi_lista_expr_expandida(dividendo_cpy);
1972             //se der merda aqui, tenho que zerar as variaveis de retorno
1973             *quociente = constroi_elemento_zerado();
1974             *resto = constroi_elemento_zerado();
1975             return 0;
1976         }
1977         else if (resultado_monomio->parametro == 0.0) // nao houve divisao
1978         {
1979             //O LT do dividendo passa a integrar o resto
1980             if (resto_temp == NULL)
1981             {
1982                 resto_temp = dividendo_cpy;
1983                 poly_ptr = resto_temp; //manter a referencia para por NULL no final
1984             }
1985             else
1986             {
1987                 //inserir o resto no final da lista do resto
1988                 poly_ptr = resto_temp;
1989                 while (poly_ptr->proximo != NULL)
1990                     poly_ptr = poly_ptr->proximo;
1991                 poly_ptr->proximo = dividendo_cpy;
```

```
1992                 dividendo_cpy->anterior = poly_ptr;
1993                 poly_ptr = poly_ptr->proximo; //manter a referencia para por NULL no final
1994             }
1995             //O LT passa a ser o proximo monomio
1996             dividendo_cpy = dividendo_cpy->proximo;
1997             if (dividendo_cpy != NULL)
1998                 dividendo_cpy->anterior = NULL;
1999             //colocar NULL no final do resto
2000             poly_ptr->proximo = NULL;
2001 
2002             //desalocar o resultado_monomio
2003             destroi_lista_expr_expandida(resultado_monomio);
2004 
2005         }
2006         else //caso houve divisao
2007         {
2008 
2009             //eliminar o LT do dividendo, pois sabemos que ele sera zerado na operacao
2010             poly_ptr = dividendo_cpy;
2011             dividendo_cpy = dividendo_cpy->proximo;
2012             if (dividendo_cpy != NULL)
2013                 dividendo_cpy->anterior = NULL;
2014             poly_ptr->proximo = NULL;
2015             destroi_lista_expr_expandida(poly_ptr);
2016 
2017             //multiplicar o monomio resultante pelo divisor sem o LT (pois este produto ira cancelar com o LT do
       dividendo) e subtrair do dividendo
2018             if (divisor->proximo != NULL) //se o divisor tiver apenas um monomio, pula-se esta etapa.
2019             {
2020                 poly_ptr = multiplica_expr(resultado_monomio, divisor->proximo);
2021 
2022                 //subtrai-se poly_ptr do dividendo
2023                 subtrai_expr(&dividendo_cpy, poly_ptr);
2024             }
2025 
2026             //o monomio resultante integra o quociente
2027             if (quociente_temp == NULL)
2028                 quociente_temp = resultado_monomio;
2029             else
2030             {
2031                 //inserir o monomio resultante ao final do quociente
2032                 poly_ptr = quociente_temp;
2033                 while (poly_ptr->proximo != NULL)
2034                     poly_ptr = poly_ptr->proximo;
2035                 poly_ptr->proximo = resultado_monomio;
2036                 resultado_monomio->anterior = poly_ptr;
2037                 resultado_monomio->proximo = NULL;
2038             }
2039         }
2040     }
2041 
2042     //ao final da divisao, reordenar e simplificar o quociente e o resto
2043     if (quociente_temp != NULL)
2044     {
2045         poly_ptr = simplifica_expr_expandida(quociente_temp);
2046         destroi_lista_expr_expandida(quociente_temp);
2047         quociente_temp = lexdegbubblesort(poly_ptr);
2048 
2049     }
2050     else
2051     {
2052         //construir um elemento zerado
2053         quociente_temp = constroi_elemento_zerado();
2054 
2055     }
2056     if (resto_temp != NULL)
2057     {
2058         poly_ptr = simplifica_expr_expandida(resto_temp);
2059         destroi_lista_expr_expandida(resto_temp);
2060         resto_temp = lexdegbubblesort(poly_ptr);
2061 
2062     }
2063     else
2064     {
2065         //construir um elemento zerado
2066         resto_temp = constroi_elemento_zerado();
2067     }
2068 
2069     *quociente = quociente_temp;
2070     *resto = resto_temp;
2071 
```

```
2072     return 1;
2073
2074 }
2075
2076
2077 //funcao que subtrai duas expressoes
2078 void subtrai_expr(lista_expr **no_esq, lista_expr *no_dir)
2079 {
2080     lista_expr *poly_ptr;
2081
2082     //troca o sinal dos elementos do no direito
2083     poly_ptr = no_dir;
2084     while(poly_ptr != NULL)
2085     {
2086        /* if(poly_ptr->sinal == operador_mais)
2087             poly_ptr->sinal = operador_menos;
2088         else
2089             poly_ptr->sinal = operador_mais;*/
2090
2091         poly_ptr->parametro *= -1.0;
2092
2093         poly_ptr = poly_ptr->proximo;
2094     }
2095     //aponta para o final da lista do no esquerdo
2096     if (*no_esq != NULL)
2097     {
2098         poly_ptr = *no_esq;
2099         while (poly_ptr->proximo != NULL)
2100             poly_ptr = poly_ptr->proximo;
2101
2102         //concatenacao das expressoes
2103         poly_ptr->proximo = no_dir;
2104         if (no_dir != NULL)
2105         {
2106             no_dir->anterior = poly_ptr;
2107         }
2108
2109     }
2110     else //o elemento do no esquerdo e vazio
2111     {
2112         *no_esq = no_dir;
2113     }
2114 }
2115
2116 //funcao que soma duas expressoes
2117 void soma_expr(lista_expr *no_esq, lista_expr *no_dir)
2118 {
2119     lista_expr *poly_ptr;
2120
2121     //aponta para o final da lista do no esquerdo
2122     poly_ptr = no_esq;
2123     while (poly_ptr->proximo != NULL)
2124         poly_ptr = poly_ptr->proximo;
2125
2126     //concatenacao das expressoes
2127     poly_ptr->proximo = no_dir;
2128     no_dir->anterior = poly_ptr;
2129 }
2130
2131 //funcao que realiza a divisao entre LT's -> OTIMIZACAO
2132 lista_expr *divide_monomio(lista_expr *dividendo, lista_expr *divisor)
2133 {
2134     char *dividendo_ptr;
2135     char *divisor_ptr;
2136     char *div_string = NULL;        //guardar a string do monomio resultante
2137     lista_expr *div_monomio = NULL; //guardar a estrutur do monomio resultante
2138     double div_parametros = 1.0;    //guardar o parametro resultante
2139     int tamanho_string =0 ;             //calcula o tamanho da string do monomio a ser alocada
2140
2141     if (divisor == NULL)
2142         return NULL;
2143     //teste de erro
2144     if (divisor->parametro == 0.0)
2145     {
2146         //erro divisao por 0
2147         return NULL;
2148     }
2149     //teste de apenas parametro
2150     if (dividendo->codigos_numerador == NULL)
2151     {
2152         //erro o monomio e apenas um parametro
```

```
2153            return NULL;
2154        }
2155
2156        dividendo_ptr = dividendo->codigos_numerador;
2157        divisor_ptr = divisor->codigos_numerador;
2158
2159        //divide os parametros
2160        div_parametros = dividendo->parametro / divisor->parametro;
2161
2162        //testar se e uma simples divisao de pariametros
2163        if (dividendo->codigos_numerador == NULL && divisor->codigos_numerador == NULL)
2164        {
2165            //nao havera string de parametros
2166            div_string = NULL;
2167        }
2168        else //ha uma string no divisor ou no dividendo ou em ambos TODO: aqui pode dar pau com string nula
2169        {
2170            //calcula-se o tamanho da string a ser alocado
2171            if (dividendo->codigos_numerador != NULL)
2172                tamanho_string = (int)strlen(dividendo->codigos_numerador);
2173            if (divisor->codigos_numerador != NULL)
2174                tamanho_string -= (int)strlen(divisor->codigos_numerador);
2175
2176            //se o divisor tiver grau maior que o dividendo, deve-se retornar 0
2177            if (tamanho_string <0)
2178            {
2179                div_parametros = 0.0;
2180                div_string = NULL;
2181            }
2182            //caso o dividendo e divisor tenham grau semelhante
2183            else if (tamanho_string == 0)
2184            {
2185                //se as strings forem diferentes, muda o resultado de div_parametros para 0
2186                if (strcmp(dividendo->codigos_numerador, divisor->codigos_numerador))
2187                    div_parametros = 0.0;
2188                //caso contrario, mantem-se o div_parametros calculado anteriormente.
2189                //em qualquer hipotese, nao ha string de variaveis aqui
2190                //div_string = NULL; -> ja esta inicializado assim
2191            }
2192            //string no dividendo maior que no divisor
2193            else
2194            {
2195                //caso o divisor seja apenas um parametro
2196                if (divisor->codigos_numerador == NULL)
2197                {
2198                    //copia a string do dividendo na string do monomio
2199                    div_string = (char *)malloc((tamanho_string+1)*sizeof(char)); //TODO: tratar erro de alocacao de memoria
2200                    *div_string = '\0';
2201                    strcpy(div_string, dividendo->codigos_numerador);
2202                }
2203                //ha string no divisor, encontrar a string que a completa para que fique igual ao dividendo.
2204                else
2205                {
2206                    //aloca a string do resultado
2207                    div_string = (char *)malloc((tamanho_string+1)*sizeof(char)); //TODO: tratar erro de alocacao de memoria
2208                    *div_string = '\0';
2209
2210                    //busca os caracteres da string do divisor na string do dividendo
2211                    while ((*dividendo_ptr) != '\0')
2212                    {
2213                        //caso nao encontre o caracter no divisor, adiciona-lo ao resultado
2214                        if (*dividendo_ptr != *divisor_ptr)
2215                        {
2216                            strncat(div_string,dividendo_ptr,1);
2217                            //incrementa ponteiro apenas do dividendo
2218                            dividendo_ptr++;
2219                        }
2220                        //caso os caracteres sejam iguais
2221                        else
2222                        {
2223                            //incrementa ambos ponteiros
2224                            dividendo_ptr++;
2225                            divisor_ptr++;
2226                        }
2227
2228                    }
2229
2230                    //ao final do processo, se o divisor nao tiver sido esgotado, entao os monomios nao dividem
2231                    if (*divisor_ptr != '\0')
2232                    {
2233                        //liberar a memoria do resultado
```

```
2234                     free(div_string);
2235                     div_string = NULL;
2236                     //o parametro resultante devera ser 0
2237                     div_parametros = 0.0;
2238                 }
2239             }
2240         }
2241     }
2242 
2243     //com todas as possibilidades cobertas, basta alocar a memoria para o monomio resultante e preencher a estrutura
2244     div_monomio = (lista_expr *)malloc(sizeof(lista_expr)); // TODO: tratar erro
2245 
2246     div_monomio->proximo = NULL;
2247     div_monomio->anterior = NULL;
2248     div_monomio->codigos_denominador = NULL;
2249     div_monomio->codigos_numerador = div_string;
2250     div_monomio->parametro = div_parametros;
2251 
2252     return div_monomio;
2253 }
2254 
2255 lista_expr *constroi_elemento_zerado(void)
2256 {
2257     lista_expr *elemento;
2258     //construir um elemento zerado
2259     elemento = (lista_expr *)malloc(sizeof(lista_expr)); //TODO: controle de erro
2260     elemento->codigos_numerador = NULL;
2261     elemento->codigos_denominador = NULL;
2262     elemento->proximo = NULL;
2263     elemento->anterior = NULL;
2264     //(*quociente)->sinal = operador_mais;
2265     elemento->parametro = 0.0;
2266 
2267     return elemento;
2268 }
2269 
2270 
2271 /*
2272 
2273 Fracoes parciais->  P3/(P1 * P2) =  Q + a/P1 + b/P2
2274 
2275 - O denominador de cada elemento da lista_expr devera ser um ponteiro para outra lista_expr.
2276 
2277 etapas da expansao em fracao parcial:
2278 
2279 */
2280 //retorna 0 se nao encontrar uma expansao certinha, ou 1 se encontrar
2281 int partial_fraction_expansion(lista_expr *P3, lista_expr *P1, lista_expr *P2, lista_expr **quociente,
2282                                 lista_expr **numerador_a, lista_expr **numerador_b)
2283 {
2284     lista_expr *lista_ptr1, *lista_ptr2, *lista_ptr3, *P3_resto, *resto;
2285     int achou = 0;  //controle de loop de busca
2286     char common_var = 0;    //variavel em comum encontrada para a expansao
2287     double coeficiente = 0.0; //coeficiente da variavel comum
2288 
2289     //inicializacao de retornos
2290     *numerador_a = NULL;
2291     *numerador_b = NULL;
2292     *quociente = NULL;
2293     resto = NULL;
2294 
2295     //incrementa o numero de fracoes parciais executadas, para comparaca de performance
2296     global_num_parfrac++;
2297 
2298     //primeiro: dividir P3 por P1*P2 e guardar o Quociente
2299 
2300     //multiplica-se P1 e P2
2301     lista_ptr1 = multiplica_expr(P1, P2);
2302 
2303     //condicao de erro
2304     if (lista_ptr1 == NULL)
2305     {
2306         return 0;
2307     }
2308 
2309     //simplifica, reordena e elimina a lista desordenada
2310     lista_ptr2 = simplifica_expr_expandida(lista_ptr1);
2311     destroi_lista_expr_expandida(lista_ptr1);
2312     lista_ptr1 = lexdegbubblesort(lista_ptr2);
2313 
2314     //divide-se P3 por P1 *P2, guardando em lista_ptr3, o resto que vai gerar as fracoes parciais.
```

```
2315    lista_ptr3 = NULL;
2316    polydiv(P3, lista_ptr1, quociente, &lista_ptr3); //TODO: controle de erro
2317
2318    //salva o resto para depois
2319    P3_resto = copia_lista_expr(lista_ptr3);
2320
2321    //elimina-se P1*P2 expandido
2322    destroi_lista_expr_expandida(lista_ptr1);
2323
2324    //1) escolhe-se uma variavel comum a P1 e P2
2325    //comparo cada variavel de cada monomio de p1 com todas as variaveis de P2, ate encontrar.
2326    lista_ptr1 = P1;
2327
2328    //compara cada elemento de P1 com todos de P2
2329    while (lista_ptr1 != NULL && !achou)
2330    {
2331        //reinicializa lista_ptr2
2332        lista_ptr2 = P2;
2333        while (lista_ptr2 != NULL && !achou)
2334        {
2335            //partindo do principio que os polinomios geradores so possuem monomios de primeiro grau, devido
2336            // a serem formado por combinacao linear entre as variaveis, a comparacao sera simplificada.
2337            if (*(lista_ptr1->codigos_numerador) == *(lista_ptr2->codigos_numerador))
2338            {
2339                achou = 1; //fim do loop
2340                common_var = *(lista_ptr1->codigos_numerador);
2341            }
2342            lista_ptr2 = lista_ptr2->proximo;
2343
2344        }
2345        lista_ptr1 = lista_ptr1->proximo;
2346    }
2347    if (!achou)
2348    {
2349        //eleger uma variavel qualquer caso nao haja variaveis em comum
2350        //neste caso, a primeira variavel de P1
2351        common_var = *(P1->codigos_numerador);
2352        //TODO: implementar algum tipo de contador para verificar estes casos e refinar mais tarde
2353    }
2354
2355    // 2) encontra-se o polinomio que faca P1 ser zero ao substituir na variavel comum
2356    // -gero um polinomio igual a P1, mas sem a variavel e com sinal trocado. Divido o resultado
2357    // pelo coeficiente do monomio que contem a variavel.
2358
2359    lista_ptr1 = copia_lista_expr(P1);
2360    //salvo o inicio da lista
2361    lista_ptr2 = lista_ptr1;
2362
2363    //novamente, P1 e P2 sao apenas combinacoes lineares das variaveis, entao este passo esta optimizado, comparando
2364    // apenas o caractere do numerador, e procurando apenas uma unica vez.
2365    while (lista_ptr2 != NULL)
2366    {
2367        if (*(lista_ptr2->codigos_numerador) == common_var)
2368        {
2369            //guardo o coeficiente da variavel comumn no monomio a ser removido
2370            coeficiente = lista_ptr2->parametro;
2371            //removo o ponteiro
2372            lista_ptr1 = remove_lista_expr(lista_ptr1, lista_ptr2);
2373            //interrompo o laco
2374            break;
2375        }
2376        else //do contrario avanco ao proximo.
2377            lista_ptr2 = lista_ptr2->proximo;
2378    }
2379
2380    //TODO: caso lista_ptr2 seja NULL, procurar a variavel em P2, condicao em que nao ha variavel comum
2381    //entretatno, se eu sempre eleger uma variavel de P1, este problema jamais ocorrera.
2382
2383    //se lista_ptr1 ficar null, criar um elemento zerado
2384    if (lista_ptr1 == NULL)
2385        lista_ptr1 = constroi_elemento_zerado();
2386    else
2387    {
2388        lista_ptr2 = lista_ptr1;
2389        while (lista_ptr2 != NULL)
2390        {
2391            //agora que removi common_var dos monomios, devo inverter o sinal dos que sobraram e dividir pelo coeficiente
2392            //salvo o inicio da lista
2393            lista_ptr2->parametro *= -1.0;
2394            lista_ptr2->parametro /= coeficiente;
2395            lista_ptr2 = lista_ptr2->proximo;
```

```
2396        }
2397
2398    }
2399
2400    //3) substitui o polinomio encontrado na variavel comum em lista_expr3
2401    //agora e a parte mais complicada, devo scanear cada variavel de cada monomio de lista_expr3,
2402    //e se encontrar a variavel comum, eu a excluo, e multiplico o monomio resultante por cada um dos monomios de
2403    //lista_ptr1 resultante do passo anterior. continuo escaneando a partir do primeiro monomio modificado,
2404    //ja que a variavel pode estar em um grau maior.
2405
2406
2407    lista_ptr2 = substitui_var(lista_ptr3, lista_ptr1, common_var);
2408
2409    //destroi-se opolinomio de substituicao
2410    destroi_lista_expr_expandida(lista_ptr3);
2411
2412    //atualiza-se o ponteiro
2413    lista_ptr3 = lista_ptr2;
2414
2415    //4) simplifica e reordena P3_resto
2416    //BUG: 0,9 - 0,9 = 1.11E-15, ou seja tenho que forcar um zero no braco.
2417    lista_ptr2 = simplifica_expr_expandida(lista_ptr3);
2418
2419
2420    //destroi-se P3_resto sem simplificacao
2421    destroi_lista_expr_expandida(lista_ptr3);
2422
2423    //ordena-se o polinomio
2424    lista_ptr2 = lexdegbubblesort(lista_ptr2);
2425
2426    //substituo o polinomio de substituicao em P2
2427    lista_ptr3 = substitui_var(P2, lista_ptr1, common_var);
2428
2429    //destruo o polinomio de substituicao na variavel comum
2430    destroi_lista_expr_expandida(lista_ptr1);
2431
2432    //simplifico e reordeno
2433    lista_ptr1 = simplifica_expr_expandida(lista_ptr3);
2434    lista_ptr1 = lexdegbubblesort(lista_ptr1);
2435    destroi_lista_expr_expandida(lista_ptr3);
2436    lista_ptr3 = NULL;
2437
2438   // 5) divide-se P3_resto por P2_subst e guarda o resultado em numerador_a
2439    resto = NULL;
2440
2441    if(!polydiv(lista_ptr2, lista_ptr1, numerador_a, &resto))
2442    {
2443        //divisao por 0
2444        destroi_lista_expr_expandida(*numerador_a);
2445        *numerador_a = NULL;
2446        destroi_lista_expr_expandida(lista_ptr1);
2447        destroi_lista_expr_expandida(lista_ptr2);
2448        destroi_lista_expr_expandida(*quociente);
2449        destroi_lista_expr_expandida(P3_resto);
2450        destroi_lista_expr_expandida(resto);
2451        *quociente = NULL;
2452        return 0;
2453
2454
2455    }
2456    destroi_lista_expr_expandida(lista_ptr1);
2457    destroi_lista_expr_expandida(lista_ptr2);
2458
2459    //HIPOTESE  - se o resto for diferente de 0, entao obrigatoriamente teremos um numerador em "a" de grau igual a p_3,
         resto.
2460    //Eu poderia ter feito toda a conta e encontrado os numeradores para depois testar o grau, mas quero que esta etapa
         seja eficiente.
2461
2462    if (resto->parametro != 0.0)
2463    {
2464        //limpar tudo e retornar 0
2465        destroi_lista_expr_expandida(*numerador_a);
2466        destroi_lista_expr_expandida(*quociente);
2467        destroi_lista_expr_expandida(P3_resto);
2468        destroi_lista_expr_expandida(resto);
2469        *numerador_a = NULL;
2470        *quociente = NULL;
2471        return 0;
2472    }
2473
2474    //liberar o resto para a proxima etapa
```

```
2475     destroi_lista_expr_expandida(resto);
2476     resto = NULL;
2477
2478     //foi encontrado um numerador_a decente. Encontrar agora o numerador_b
2479     // Para achar o segundo numerador da fracao parcial:
2480     //1) multiplico "a" por P2
2481     //copio P3
2482     lista_ptr2 = copia_lista_expr(P3_resto);
2483     lista_ptr1 = multiplica_expr(*numerador_a, P2);
2484
2485     //2) subtraio lista_ptr1 de P3_resto, e guardo o resultado em lista_ptr2
2486     subtrai_expr(&lista_ptr2, lista_ptr1);
2487     lista_ptr3 = simplifica_expr_expandida(lista_ptr2);
2488     destroi_lista_expr_expandida(lista_ptr2);
2489     lista_ptr2 = lexdegbubblesort(lista_ptr3);
2490
2491     //3) divido o resultado por P1
2492     polydiv(lista_ptr2, P1, numerador_b, &resto);
2493
2494     //HIPOTESE - igual acima.
2495     if (resto->parametro != 0.0)
2496     {
2497         //limpar tudo e retornar 0
2498         destroi_lista_expr_expandida(lista_ptr2);
2499         destroi_lista_expr_expandida(*numerador_b);
2500         destroi_lista_expr_expandida(*numerador_a);
2501         destroi_lista_expr_expandida(*quociente);
2502         *numerador_a = NULL;
2503         *numerador_b = NULL;
2504         *quociente = NULL;
2505         destroi_lista_expr_expandida(resto);
2506         destroi_lista_expr_expandida(P3_resto);
2507
2508         return 0;
2509     }
2510
2511     //se chegou ate aqui, a operacao foi um sucesso
2512     destroi_lista_expr_expandida(lista_ptr2);
2513     destroi_lista_expr_expandida(resto);
2514     destroi_lista_expr_expandida(P3_resto);
2515     return 1;
2516
2517 }
2518
2519
2520 lista_expr *substitui_var(lista_expr *p_destino, lista_expr *p_fonte, char common_var)
2521 {
2522     char *str_ptr;
2523     lista_expr *lista_ptr1, *lista_ptr2, *lista_ptr3, *p_dest;
2524     p_dest = copia_lista_expr(p_destino);
2525     while (TRUE)
2526     {
2527         //procuro pela variavel comum no monomio
2528         if(p_dest->codigos_numerador != NULL)
2529         {
2530             if ((str_ptr = strchr(p_dest->codigos_numerador, common_var)) != NULL)
2531             {
2532                 //removo o caracter da variavel da string, deslocando o restante dela para a esquerda
2533                 while (*str_ptr != '\0')
2534                 {
2535                     *str_ptr = *(str_ptr + 1);
2536                     str_ptr++;
2537                 }
2538
2539                 //inicializacao de ponteiros
2540                 lista_ptr1 = NULL;
2541                 lista_ptr2 = NULL;
2542
2543                 //agora isolo o monomio, caso P3 tenha mais de 1
2544                 if (p_dest->proximo != NULL)
2545                 {
2546                     //salvo o proximo monomio
2547                     lista_ptr1 = p_dest->proximo;
2548
2549                     //isolo o monomio
2550                     p_dest->proximo = NULL;
2551                 }
2552
2553                 if (p_dest->anterior != NULL)
2554                 {
2555                     //salvo o monomio anterior
```

```
2556                     lista_ptr2 = p_dest->anterior;
2557 
2558                     //pode ser que isso seja desnecessario se eu for optimizar
2559                     p_dest->anterior = NULL;
2560                 }
2561 
2562                 //a principio, a string conter apenas um \0 nao impede a multiplicacao de funcionar corretamente
2563                 // mas se der bug, devo comecar por aqui.
2564                 //realizo a multiplicacao
2565                 lista_ptr3 = multiplica_expr(p_fonte, p_dest);
2566 
2567                 //desaloco o monomio original
2568                 destroi_lista_expr_expandida(p_dest);
2569 
2570                 //reaproveito o ponteiro para apontar para o proximo monomio a ser avaliado, que e o primeiro
2571                 //do resultado da multiplicacao, para que esgote-se todos os graus da variavel comum
2572                 p_dest = lista_ptr3;
2573 
2574                 //conecto o polinomio resultante da multiplicacao
2575                 lista_ptr3->anterior = lista_ptr2;
2576                 if (lista_ptr2 != NULL)
2577                     lista_ptr2->proximo = lista_ptr3;
2578 
2579                 //encontro o ultimo monomio reultante da multiplicacao
2580                 while (lista_ptr3->proximo != NULL)
2581                     lista_ptr3 = lista_ptr3->proximo;
2582 
2583                 //e o conecto com o resto de P3
2584                 lista_ptr3->proximo = lista_ptr1;
2585                 if (lista_ptr1!= NULL)
2586                     lista_ptr1->anterior = lista_ptr3;
2587 
2588                 //como ja determinei qual monomio sera avaliado no proximo passo, posso pular para o inicio do laco
2589                 continue;
2590             }
2591         }// else: TODO eu simplesmente pulo o monomio se for so um parametro.
2592 
2593         //continuo a busca ou termino o laco, "rebobinando" o ponteiro para o inicio da lista
2594         if (p_dest->proximo == NULL)
2595         {
2596             while (p_dest->anterior != NULL)
2597                 p_dest = p_dest->anterior;
2598 
2599             break;
2600         }
2601         else
2602             p_dest = p_dest->proximo;
2603 
2604     }
2605     return p_dest;
2606 }
2607 
2608 
2609 
2610 
2611 //funcao que gera um conjunto de vetores de entrada
2612 vetor_polinomios *gera_vetor(vetor_polinomios *ultimo_gerado, lista_expr *polinomio_base, lista_expr *variavel_atual, int
        lim_inferior, int lim_superior)
2613 {
2614     //variavel que vai dar um identificador aos polinomios
2615     static int poly_id = 0;
2616     vetor_polinomios *elemento_atual = NULL;
2617     int i;
2618     for (i=lim_superior; i>=lim_inferior; i--)
2619     {
2620         //atribuo a variavel atual o valor de i;
2621         variavel_atual->parametro = i;
2622         //percorre o polinomio base ate chegar na ultima variavel
2623         if (variavel_atual->proximo != NULL)
2624             ultimo_gerado = gera_vetor(ultimo_gerado, polinomio_base, variavel_atual->proximo, lim_inferior, lim_superior
       );
2625         else //chegou na ultima variavel
2626         {
2627             //ao chegar na ultima variavel, gerar vetores variando o parametro da ultima variavel de lim_inferior a
       lim_superior
2628             for (i=lim_superior; i>=lim_inferior; i--)
2629             {
2630                 variavel_atual->parametro = i;
2631                 elemento_atual = (vetor_polinomios *)malloc(sizeof(vetor_polinomios)); //TODO: controle de erro
2632                 elemento_atual->polinomio = (polinomio *)malloc(sizeof(polinomio)); //TODO: controle de erro
2633                 //copio o polinomio base com os parametros do jeito que estao
```

```
2634                //elimino os elementos zero
2635                elemento_atual->polinomio->P = simplifica_expr_expandida(polinomio_base);
2636
2637                //atribuo um identificador e incremento
2638                elemento_atual->polinomio->id = poly_id;
2639                poly_id++;
2640
2641                //initialize the approval flag for further BP set reduction purposes
2642                elemento_atual->polinomio->flag_approved = 0;
2643
2644                //concateno na lista de polinomios
2645                if (ultimo_gerado != NULL)
2646                {
2647                    ultimo_gerado->proximo_polinomio = elemento_atual;
2648                    elemento_atual->polinomio_anterior = ultimo_gerado;
2649                    ultimo_gerado = elemento_atual;
2650                    ultimo_gerado->proximo_polinomio = NULL;
2651                }
2652                else
2653                {
2654                    ultimo_gerado = elemento_atual;
2655                    elemento_atual->proximo_polinomio = NULL;
2656                    elemento_atual->polinomio_anterior = NULL;
2657                }
2658            }
2659            return ultimo_gerado;
2660        }
2661    }
2662    return ultimo_gerado;
2663 }
2664
2665 vetor_polinomios *elimina_zero(vetor_polinomios *lista)
2666 {
2667    vetor_polinomios *p_lista = lista;
2668    while (p_lista != NULL)
2669    {
2670        //apenas no polinomio nulo, o primeiro monomio e zero
2671        if (p_lista->polinomio->P->parametro == 0.0)
2672        {
2673            p_lista = remove_polinomio(p_lista);
2674
2675            //rebobinar a lista
2676            while (p_lista->polinomio_anterior != NULL)
2677                p_lista = p_lista->polinomio_anterior;
2678
2679            return p_lista;
2680        }
2681
2682        //incrementa a busca
2683        p_lista = p_lista->proximo_polinomio;
2684    }
2685    //se nada encontrou, retorna o inicio da lista
2686    return lista;
2687 }
2688 //retorna o elemento anterior ao removido, ou o inico da lista
2689 vetor_polinomios *remove_polinomio(vetor_polinomios *elemento)
2690 {
2691    vetor_polinomios *esq, *dir;
2692
2693    //gravo as bordas
2694    esq = elemento->polinomio_anterior;
2695    dir = elemento->proximo_polinomio;
2696
2697    //ajusto os ponteiros Ĺ esquerda
2698    if (esq != NULL)
2699        esq->proximo_polinomio = dir;
2700
2701    //ajusto os ponteiros Ĺ direita
2702    if (dir != NULL)
2703        dir->polinomio_anterior = esq;
2704
2705    //eliminar o elemento
2706    destroi_lista_expr_expandida(elemento->polinomio->P);
2707    free(elemento->polinomio);
2708    free(elemento);
2709
2710    //caso tenha esvaziado a lista
2711    if (dir == NULL && esq == NULL)
2712        return NULL;
2713
2714    //se apeans o esquerdo for nulo, ja sabemos que o inicio da lista e o da direita
```

```
2715     if (esq == NULL)
2716     {
2717         return dir;
2718     }
2719     else //retornar o esquerdo
2720     {
2721         return esq;
2722     }
2723 }
2724
2725 //retorna o anterior, pois pode ser que remova o primeiro elemento
2726 vetor_polinomios *remove_polinomio_retorna_anterior(vetor_polinomios *elemento)
2727 {
2728     vetor_polinomios *esq, *dir;
2729
2730     //gravo as bordas
2731     esq = elemento->polinomio_anterior;
2732     dir = elemento->proximo_polinomio;
2733
2734     //ajusto os ponteiros Ĺ esquerda
2735     if (esq != NULL)
2736         esq->proximo_polinomio = dir;
2737
2738     //ajusto os ponteiros Ĺ direita
2739     if (dir != NULL)
2740         dir->polinomio_anterior = esq;
2741
2742     //eliminar o elemento
2743     destroi_lista_expr_expandida(elemento->polinomio->P);
2744     free(elemento->polinomio);
2745     free(elemento);
2746
2747     //caso tenha esvaziado a lista
2748     if (dir == NULL && esq == NULL)
2749         return NULL;
2750
2751     //retornar o proximo polinomio
2752     return esq;
2753 }
2754
2755
2756 //gera o polinomio base
2757 lista_expr *gera_polinomio_base(tabela_literais *lista_literais)
2758 {
2759     tabela_literais *percorre_tabela_literais = lista_literais;
2760     lista_expr *polinomio_base, *monomio_atual;
2761
2762     polinomio_base = NULL;
2763
2764     // percorre as variaveis gravadas na tabela de literais, gerando um polinomio que e a combinacao linear delas.
2765     while (percorre_tabela_literais != NULL)
2766     {
2767         monomio_atual = constroi_elemento_zerado();
2768         monomio_atual->codigos_numerador = (char *)malloc(2*sizeof(char));
2769         monomio_atual->codigos_numerador[0] = percorre_tabela_literais->codigo;
2770         monomio_atual->codigos_numerador[1] = '\0';
2771         monomio_atual->parametro = 1.0;
2772         monomio_atual->codigos_denominador = NULL;
2773         if (polinomio_base == NULL)
2774         {
2775             polinomio_base = monomio_atual;
2776             polinomio_base->anterior = NULL;
2777             polinomio_base->proximo = NULL;
2778         }
2779         else
2780         {
2781             polinomio_base->proximo = monomio_atual;
2782             monomio_atual->anterior = polinomio_base;
2783             polinomio_base = polinomio_base->proximo;
2784         }
2785         percorre_tabela_literais = percorre_tabela_literais->proximo_codigo;
2786
2787     }
2788     //rebobina o polinomio base
2789     while (polinomio_base->anterior != NULL)
2790     {
2791         polinomio_base = polinomio_base->anterior;
2792     }
2793
2794     return polinomio_base;
2795
```

```
2796 }
2797
2798 vetor_polinomios *remove_polinomios_negativos (vetor_polinomios *vetor_entrada)
2799 {
2800     vetor_polinomios *percorre_vetor;
2801     lista_expr *percorre_expr;
2802     int teste;
2803
2804     //inicializo o ponteiro de varredura
2805     percorre_vetor = vetor_entrada;
2806
2807     //condicao de erro
2808     if (percorre_vetor == NULL)
2809         return NULL;
2810     //varredura de todas as entradas geradas removendo tododos os polinomios onde todos os coeficientes sao negativos
2811     while (percorre_vetor != NULL)
2812     {
2813         //aponta para o polinomio
2814         percorre_expr = percorre_vetor->polinomio->P;
2815
2816         //inicializo a variavel de teste
2817         teste = 0;
2818
2819         //percorro o polinomio procurando coeficientes positivos. Se achar algum mudo o estado da variavel de teste
2820         while (percorre_expr != NULL)
2821         {
2822             if (percorre_expr->parametro > 0.0)
2823             {
2824                 teste = 1;
2825                 break;
2826             }
2827
2828             //incremento o ponteiro
2829             percorre_expr = percorre_expr->proximo;
2830         }
2831
2832         //se teste for 0, o polinomio e todo negativo, e deve ser removido do vetor
2833         if (!teste)
2834         {
2835             //a variavel de retorno guarda o polinomio anterior, para que o final da lista seja mantido
2836             percorre_vetor = remove_polinomio_retorna_anterior(percorre_vetor);
2837
2838         }
2839         //se nao for o ultimo elemento incrementa, caso contrario sai do loop
2840         else if(percorre_vetor->proximo_polinomio != NULL)
2841             percorre_vetor = percorre_vetor->proximo_polinomio;
2842         else
2843             break;
2844
2845     }
2846
2847     //rebobina o vetor reduizdo
2848     while (percorre_vetor->polinomio_anterior != NULL)
2849     {
2850         percorre_vetor = percorre_vetor->polinomio_anterior;
2851     }
2852
2853     return percorre_vetor;
2854 }
2855
2856 vetor_polinomios *remove_polinomios_redundantes (vetor_polinomios *vetor_entrada)
2857 {
2858     vetor_polinomios *p_vetor1, *p_vetor2;
2859     lista_expr *expr1, *expr2, *Q, *R;
2860
2861     int flag = 0;
2862
2863     //inicializo o ponteiro de varredura
2864     p_vetor1 = vetor_entrada;
2865
2866     //condicao de erro
2867     if (p_vetor1 == NULL)
2868         return NULL;
2869     //para cada vetor, eu divido ele por todos os outros polinomoios. se o resto da zero, e porque um e combinacao linear
           do outro
2870     while (p_vetor1->proximo_polinomio != NULL)
2871     {
2872         flag = 0;
2873
2874         p_vetor2 = p_vetor1->proximo_polinomio;
2875         while (p_vetor2 != NULL)
```

```
2876        {
2877            // pula caso os polinomios forem os mesmos
2878            if (p_vetor2 == p_vetor1)
2879            {
2880                p_vetor2 = p_vetor2->proximo_polinomio;
2881                continue;
2882            }
2883            //copio a expressao de vetor1
2884            expr1 = copia_lista_expr(p_vetor1->polinomio->P);
2885            //copio a expressao de vetor2
2886            expr2 = copia_lista_expr(p_vetor2->polinomio->P);
2887
2888             //divido os dois
2889            polydiv(expr1, expr2, &Q, &R);
2890
2891            //seo resto for 0, e o quociente for uma constante, devo remover p_vetor_1
2892            //bogus = (int)(R->parametro);
2893            if ((R->parametro == 0.0) && Q->codigos_numerador == NULL)
2894            {
2895                //devide se remove vetor1 ou vetor2
2896                if (fabs(Q->parametro) >= 1.0)
2897                {
2898                    //p_vetor1 apontara para o proximo da lista apos ser removido
2899                    p_vetor1 = remove_polinomio(p_vetor1);
2900                    //retoma a busca a partir do proximo polinomio principal
2901                    p_vetor2 = NULL;
2902
2903                    //seta flag de reinicio
2904                    flag = 1;
2905                }
2906                else
2907                {
2908                    //p_vetor1 apontara para o proximo da lista apos ser removido
2909                    p_vetor2 = remove_polinomio(p_vetor2);
2910                }
2911            }
2912
2913            //destruo expr
2914            destroi_lista_expr_expandida(expr1);
2915            destroi_lista_expr_expandida(expr2);
2916            destroi_lista_expr_expandida(Q);
2917            destroi_lista_expr_expandida(R);
2918            Q = NULL;
2919            R = NULL;
2920
2921            //incremento o ponteiro
2922            if (p_vetor2 != NULL)
2923                p_vetor2 = p_vetor2->proximo_polinomio;
2924        }
2925
2926        //testa flag de reinicio
2927        if (!flag)
2928        {
2929            //incremento a busca principal
2930            if (p_vetor1->proximo_polinomio!= NULL)
2931                p_vetor1 = p_vetor1->proximo_polinomio;
2932            else
2933                break;
2934        }
2935    }
2936
2937    //rebobina o vetor reduizdo
2938    while (p_vetor1->polinomio_anterior != NULL)
2939    {
2940        p_vetor1 = p_vetor1->polinomio_anterior;
2941    }
2942
2943    return p_vetor1;
2944 }
2945
2946 //funcao que retorna o grau de um polinomio
2947 int deg(lista_expr *poly_in)
2948 {
2949    int grau = 0;
2950    int grau_max = 0;
2951    lista_expr *ptr_poly = poly_in;
2952
2953    //percorre monomio a monomio verificando o grau, que nada mais e senao o tamanho da string de codigos do numerador
2954    while (ptr_poly != NULL)
2955    {
2956        if (ptr_poly->codigos_numerador == NULL)
```

```
2957            grau = 0;
2958        else
2959            grau = (int)strlen(ptr_poly->codigos_numerador);
2960
2961        if (grau > grau_max)
2962            grau_max = grau;
2963
2964        ptr_poly = ptr_poly->proximo;
2965
2966    }
2967
2968    return grau_max;
2969 }
2970
2971 vetor_sementes *gera_vetor_semente(vetor_polinomios *vetor_in, lista_expr *eq_entrada)
2972 {
2973    vetor_polinomios *ptr_vetor_in;
2974    vetor_polinomios *ptr_vetor_base = vetor_in;
2975    vetor_sementes *vetor_gerado = NULL;
2976
2977    //resultados
2978    lista_expr *R1;
2979    lista_expr *R2;
2980    lista_expr *Q;
2981
2982    //se por um acaso passar apenas um elemento, retornar nulo
2983    if (vetor_in == NULL)
2984    {
2985        return NULL;
2986    }
2987    if (vetor_in->proximo_polinomio == NULL)
2988    {
2989        return NULL;
2990    }
2991
2992    ptr_vetor_in = vetor_in->proximo_polinomio;
2993
2994    //percorre o vetor de entrada, tentando todas as combinacoes possiveis. Como a ordem nao importa, vou sempre
     incrementando vetor_in tambem
2995    while (ptr_vetor_base->proximo_polinomio != NULL)
2996    {
2997        //ptr_vetor_in = vetor_in->proximo_polinomio;
2998        //nao restricao nenhuma em poder combinar consigo mesmo
2999        ptr_vetor_in = ptr_vetor_base;
3000
3001        while (ptr_vetor_in != NULL)
3002        {
3003            //tentar realizar a expansao em fracoes parciais
3004            if(partial_fraction_expansion(eq_entrada, ptr_vetor_base->polinomio->P, ptr_vetor_in->polinomio->P, &Q,&R1, &
     R2))
3005            {
3006                //consegui uma expansao em fracoes parciais adequada, preparar vetor de semente, ao final ele sera
     rebobinado
3007                if (vetor_gerado == NULL)
3008                    vetor_gerado = novo_vetor_semente();
3009                else
3010                {
3011                    //crio um novo elemento ja ligado ao anterior
3012                    vetor_gerado->conjunto_prox = novo_vetor_semente();
3013                    //realizo o duplo encadeamento
3014                    vetor_gerado->conjunto_prox->conjunto_ant = vetor_gerado;
3015                    //atualizo o ponteiro
3016                    vetor_gerado = vetor_gerado->conjunto_prox;
3017                }
3018
3019                //preencho os campos da semente, P1 e P2 devem ser copiados, pois ainda serao utilizados
3020                vetor_gerado->P1.P = copia_lista_expr(ptr_vetor_base->polinomio->P);
3021                vetor_gerado->P1.id = ptr_vetor_base->polinomio->id;
3022                vetor_gerado->P2.P = copia_lista_expr(ptr_vetor_in->polinomio->P);
3023                vetor_gerado->P2.id = ptr_vetor_in->polinomio->id;
3024                //estes outros 3 nao precisam de copia, pois serao utilizados apenas dentro do vetorde sementes
3025                vetor_gerado->quociente = Q;
3026                vetor_gerado->R1 = R1;
3027                vetor_gerado->R2 = R2;
3028
3029            }
3030
3031            //reseta as variaveis
3032            R1 = NULL;
3033            R2 = NULL;
3034            Q = NULL;
```

```
3035
3036            //incremento ponteiro para proximo teste
3037            ptr_vetor_in = ptr_vetor_in->proximo_polinomio;
3038        }
3039
3040        //incremento o ponteiro para a proxima bateria de testes
3041        ptr_vetor_base = ptr_vetor_base->proximo_polinomio;
3042    }
3043    //nao encontrou nenhuma semente
3044    if (vetor_gerado == NULL)
3045        return NULL;
3046    //rebobina o vetor de sementes
3047    while (vetor_gerado->conjunto_ant != NULL)
3048        vetor_gerado = vetor_gerado->conjunto_ant;
3049
3050    //retorna o vetor gerado
3051    return vetor_gerado;
3052 }
3053
3054 //cria elemento de vetor de sementes
3055 vetor_sementes *novo_vetor_semente(void)
3056 {
3057    vetor_sementes *novo;
3058
3059    novo = (vetor_sementes *)malloc(sizeof(vetor_sementes)); //TODO: conferir erro de alocacao de memoria
3060
3061    novo->P1.P = NULL;
3062    novo->P1.id = 0;
3063    novo->P2.P = NULL;
3064    novo->P2.id = 0;
3065    novo->quociente = NULL;
3066    novo->R1 = NULL;
3067    novo->R2 = NULL;
3068    novo->conjunto_ant = NULL;
3069    novo->conjunto_prox = NULL;
3070
3071    return novo;
3072 }
3073
3074 //destroi toda a lista de sementes
3075 void destroi_lista_sementes(vetor_sementes *entrada)
3076 {
3077    if (entrada != NULL)
3078        destroi_lista_sementes(entrada->conjunto_prox);
3079    else
3080        return;
3081
3082    destroi_lista_expr_expandida(entrada->P1.P);
3083    destroi_lista_expr_expandida(entrada->P2.P);
3084    destroi_lista_expr_expandida(entrada->quociente);
3085    destroi_lista_expr_expandida(entrada->R1);
3086    destroi_lista_expr_expandida(entrada->R2);
3087    free(entrada);
3088 }
3089
3090
3091 //cria um novo elemento de vetor de polinomios
3092 vetor_polinomios *novo_vetor_polinomios(void)
3093 {
3094    vetor_polinomios *novo;
3095
3096    novo = (vetor_polinomios *)malloc(sizeof(vetor_polinomios));
3097
3098    novo->polinomio = NULL;
3099    novo->polinomio_anterior = NULL;
3100    novo->proximo_polinomio = NULL;
3101
3102    return (novo);
3103 }
3104
3105 //cria elemento de vetor de decomposicao
3106 vetor_decomp *novo_vetor_decomp(void)
3107 {
3108    vetor_decomp *novo;
3109    novo = (vetor_decomp *)malloc(sizeof(vetor_decomp));
3110    novo->poly_pares = NULL;
3111    novo->poly_impares = NULL;
3112    novo->resto_impar = NULL;
3113    novo->resto_par = NULL;
3114
3115    novo->prox_decomp = NULL;
```

```
3116     novo->ant_decomp = NULL;
3117
3118     return novo;
3119 }
3120
3121 //insere um polinomio na lista de polinomios
3122 void insere_polinomio(vetor_polinomios **vetor,polinomio *elemento)
3123 {
3124     vetor_polinomios *p_vetor, *novo_poly;
3125
3126     //cria-se o novo elemento
3127     novo_poly = novo_vetor_polinomios();
3128     novo_poly->polinomio = elemento;
3129
3130     //inicializa a lista
3131     p_vetor = *vetor;
3132
3133     //caso a lista esteja vazia, tornar o novo elemento o primeiro elemento da lista
3134     if (p_vetor == NULL)
3135     {
3136         *vetor = novo_poly;
3137     }
3138     else
3139     {
3140         //procura o fim da fila
3141         while (p_vetor->proximo_polinomio != NULL)
3142             p_vetor = p_vetor->proximo_polinomio;
3143
3144         //insere o elemento
3145         p_vetor->proximo_polinomio = novo_poly;
3146         novo_poly->polinomio_anterior = p_vetor;
3147     }
3148 }
3149
3150
3151 void destroi_decomp(vetor_decomp *entrada)
3152 {
3153     if (entrada->resto_impar != NULL)
3154         destroi_lista_expr_expandida(entrada->resto_impar);
3155     if (entrada->resto_par != NULL)
3156         destroi_lista_expr_expandida(entrada->resto_par);
3157 //destroi os vetores de polinomios
3158     if (entrada->poly_pares != NULL)
3159         destroi_vetor_polinomios(entrada->poly_pares);
3160     if (entrada->poly_impares != NULL)
3161         destroi_vetor_polinomios(entrada->poly_impares);
3162
3163     free(entrada);
3164 }
3165
3166 void destroi_vetor_polinomios(vetor_polinomios *entrada)
3167 {
3168     if (entrada->proximo_polinomio != NULL)
3169     {
3170         destroi_vetor_polinomios(entrada->proximo_polinomio);
3171     }
3172
3173     //o vetor de polinomios apenas pode referenciar polinomios e nao copia-los
3174     free(entrada);
3175 }
3176
3177 void destroi_vetor_decomp(vetor_decomp *entrada)
3178 {
3179     if (entrada == NULL)
3180         return;
3181     if (entrada->prox_decomp != NULL)
3182     {
3183         destroi_vetor_decomp(entrada->prox_decomp);
3184     }
3185
3186     destroi_decomp(entrada);
3187 }
3188 //tirar a prova real entre a decomposicao encontrada e a equacao de entrada
3189 int prova_real(vetor_decomp *decomp, lista_expr *eq_entrada)
3190 {
3191     lista_expr *acum_par, *acum_impar, *acum, *acum2, *Q, *R;
3192     vetor_polinomios *poly_par_ptr = decomp->poly_pares;
3193     vetor_polinomios *poly_impar_ptr = decomp->poly_impares;
3194     int flag_teste;
3195
3196     acum_par = copia_lista_expr(poly_par_ptr->polinomio->P);
```

```
3197     acum_impar = copia_lista_expr(poly_impar_ptr->polinomio->P);
3198
3199
3200     while (poly_par_ptr->proximo_polinomio!= NULL)
3201     {
3202         acum = multiplica_expr(acum_par, poly_par_ptr->proximo_polinomio->polinomio->P);
3203         destroi_lista_expr_expandida(acum_par);
3204         acum_par = acum;
3205
3206         poly_par_ptr = poly_par_ptr->proximo_polinomio;
3207     }
3208
3209     //repito o mesmo para a parte impar
3210     while (poly_impar_ptr->proximo_polinomio!= NULL)
3211     {
3212         acum = multiplica_expr(acum_impar, poly_impar_ptr->proximo_polinomio->polinomio->P);
3213         destroi_lista_expr_expandida(acum_impar);
3214         acum_impar = acum;
3215
3216         poly_impar_ptr = poly_impar_ptr->proximo_polinomio;
3217     }
3218
3219     //simplifica a parte par
3220     acum = simplifica_expr_expandida(acum_par);
3221     destroi_lista_expr_expandida(acum_par);
3222     acum = lexdegbubblesort(acum);
3223     acum_par = acum;
3224     acum = NULL;
3225
3226     //simplifica a parte impar
3227     acum = simplifica_expr_expandida(acum_impar);
3228     destroi_lista_expr_expandida(acum_impar);
3229     acum = lexdegbubblesort(acum);
3230     acum_impar = acum;
3231     acum = NULL;
3232
3233
3234     //multiplico o resultado pelo lambda, se for diferente de zero. Caso contrario, temos que dividir a eq_entrada pelo
          acumulado
3235     if (decomp->resto_impar->parametro == 0.0 || decomp->resto_par->parametro == 0.0 ||
3236         decomp->resto_impar->parametro == NAN || decomp->resto_par->parametro == NAN ||
3237         decomp->resto_impar->parametro == INFINITY || decomp->resto_par->parametro == INFINITY)
3238     {
3239         //Divido Er por polys impares, e Acum = R e acum2 = Q
3240         polydiv(eq_entrada, acum_impar, &Q, &R);
3241         acum = Q;
3242         acum2 = R;
3243
3244         //setar um flag para saber se o primeiro teste falhou
3245         flag_teste = 0;
3246         //verificar se Q e uma constante
3247         if (Q->parametro != 0.0 && !isnan(Q->parametro) && !isinf(Q->parametro) && Q->codigos_numerador == NULL && Q->
          proximo == NULL)
3248         {
3249             Q = NULL;
3250             R = NULL;
3251             //Dividir o resto(acum2) por acum_par, se Q for uma constante, e R = 0, entao decomp->resto_impar = Q, decomp
          ->resto_par = acum
3252             polydiv(acum2, acum_par, &Q, &R);
3253             //verificar se Q e uma constante
3254             if (Q->parametro != 0.0 && !isnan(Q->parametro) && !isinf(Q->parametro) && Q->codigos_numerador == NULL && Q
          ->proximo == NULL && R->parametro == 0.0)
3255             {
3256                 destroi_lista_expr_expandida(decomp->resto_impar);
3257                 destroi_lista_expr_expandida(decomp->resto_par);
3258                 decomp->resto_par = acum;
3259                 decomp->resto_impar = Q;
3260
3261             }
3262             else
3263             {
3264                 flag_teste = 1;
3265                 destroi_lista_expr_expandida(Q);
3266                 destroi_lista_expr_expandida(R);
3267             }
3268         }
3269         else
3270         {
3271             flag_teste = 1;
3272             destroi_lista_expr_expandida(Q);
3273             destroi_lista_expr_expandida(R);
```

```
3274
3275        }
3276        Q = NULL;
3277        R = NULL;
3278        acum = NULL;
3279
3280
3281        //se a primeira tentativa nao funcionar, fazer ao contrario
3282        if (flag_teste)
3283        {
3284            //Divido Er por polys pares, e Acum = R e acum2 = Q
3285            polydiv(eq_entrada, acum_par, &Q, &R);
3286            acum = Q;
3287            acum2 = R;
3288
3289            //verificar se Q e uma constante
3290            if (Q->parametro != 0.0 && !isnan(Q->parametro) && !isinf(Q->parametro) && Q->codigos_numerador == NULL && Q
   ->proximo == NULL)
3291            {
3292                Q = NULL;
3293                R = NULL;
3294                //Dividir o resto(acum2) por acum_impar, se Q for uma constante, e R = 0, entao decomp->resto_par = Q,
   decomp->resto_impar = acum
3295                polydiv(acum2, acum_impar, &Q, &R);
3296                //verificar se Q e uma constante
3297                if (Q->parametro != 0.0 && !isnan(Q->parametro) && !isinf(Q->parametro) && Q->codigos_numerador == NULL
   && Q->proximo == NULL && R->parametro == 0.0)
3298                {
3299                    destroi_lista_expr_expandida(decomp->resto_impar);
3300                    destroi_lista_expr_expandida(decomp->resto_par);
3301                    decomp->resto_impar = acum;
3302                    decomp->resto_par = Q;
3303                }
3304                else
3305                {
3306                    destroi_lista_expr_expandida(Q);
3307                    destroi_lista_expr_expandida(R);
3308                }
3309
3310            }
3311            else
3312            {
3313                destroi_lista_expr_expandida(Q);
3314                destroi_lista_expr_expandida(R);
3315            }
3316        }
3317        Q = NULL;
3318        R = NULL;
3319        acum = NULL;
3320
3321
3322    }
3323    //se mesmo apos esta tentativa de encontrar coeficientes reais ainda tiver alguma inconsistencia, retornar 0
3324    if (decomp->resto_impar->parametro == 0.0 || decomp->resto_par->parametro == 0.0 ||
3325        decomp->resto_impar->parametro == NAN || decomp->resto_par->parametro == NAN ||
3326        decomp->resto_impar->parametro == INFINITY || decomp->resto_par->parametro == INFINITY)
3327    {
3328        destroi_lista_expr_expandida(acum_impar);
3329        destroi_lista_expr_expandida(acum_par);
3330        return FALSE;
3331    }
3332
3333    //multiplico a parte par pelo lambda impar
3334    acum = multiplica_expr(acum_par,decomp->resto_impar);
3335    destroi_lista_expr_expandida(acum_par);
3336    acum_par = simplifica_expr_expandida(acum);
3337    destroi_lista_expr_expandida(acum);
3338    acum_par = lexdegbubblesort(acum_par);
3339    acum = NULL;
3340
3341    //multiplico a parte impar pelo lambda par
3342    acum = multiplica_expr(acum_impar,decomp->resto_par);
3343    destroi_lista_expr_expandida(acum_impar);
3344    acum_impar = simplifica_expr_expandida(acum);
3345    destroi_lista_expr_expandida(acum);
3346    acum_impar = lexdegbubblesort(acum_impar);
3347    acum = NULL;
3348
3349    //somo a parte impar com a parte par
3350    acum = copia_lista_expr(acum_impar);
3351    acum2 = copia_lista_expr(acum_par);
```

```
3352     soma_expr(acum, acum2);
3353
3354     //simplifica
3355     acum2 = simplifica_expr_expandida(acum);
3356     destroi_lista_expr_expandida(acum);
3357     acum = lexdegbubblesort(acum2);
3358     acum2 = NULL;
3359
3360     //divido o resultado pela equacao de entrada, e o resto deve dar 0 e o quociente ser uma constante
3361     polydiv(eq_entrada, acum, &Q, &R);
3362     if (Q->codigos_numerador == NULL && R->codigos_numerador == NULL && R->parametro == 0.0 && !isnan(Q->parametro) && !
          isinf(Q->parametro) && Q->parametro != 0.0)
3363     {
3364         destroi_lista_expr_expandida(Q);
3365         destroi_lista_expr_expandida(R);
3366         destroi_lista_expr_expandida(acum_impar);
3367         destroi_lista_expr_expandida(acum_par);
3368         destroi_lista_expr_expandida(acum);
3369         return TRUE;
3370     }
3371     else
3372     {
3373         destroi_lista_expr_expandida(Q);
3374         destroi_lista_expr_expandida(R);
3375         destroi_lista_expr_expandida(acum_impar);
3376         destroi_lista_expr_expandida(acum_par);
3377         destroi_lista_expr_expandida(acum);
3378         return FALSE;
3379     }
3380 }
3381
3382 //insere uma lista de decomposicoes dentro de outra, retornando um ponteiro para o final dela
3383 vetor_decomp *insere_lista_decomp(vetor_decomp *lista_destino, vetor_decomp *lista_origem)
3384 {
3385     vetor_decomp *ptr_lista_origem, *ptr_lista_destino;
3386
3387     ptr_lista_origem = lista_origem;
3388     if (lista_destino == NULL)
3389     {
3390         //aponto para o final da lista a ser inserida
3391         while (ptr_lista_origem->prox_decomp != NULL)
3392             ptr_lista_origem = ptr_lista_origem->prox_decomp;
3393
3394         //retorno o fim da lista destino como o fim da lista de origem
3395         return ptr_lista_origem;
3396
3397     }
3398     else
3399     {
3400         ptr_lista_destino = lista_destino;
3401         //avancar o ponteiro para fim da lista destino
3402         while (ptr_lista_destino->prox_decomp != NULL)
3403             ptr_lista_destino = ptr_lista_destino->prox_decomp;
3404
3405         //rebobinar o ponteiro da lista de origem
3406         while (ptr_lista_origem->ant_decomp != NULL)
3407             ptr_lista_origem = ptr_lista_origem->ant_decomp;
3408
3409         //ligar as 2
3410         ptr_lista_destino->prox_decomp = ptr_lista_origem;
3411         ptr_lista_origem->ant_decomp = ptr_lista_destino;
3412
3413         //avancar o ponteiro ate o final
3414         while (ptr_lista_destino->prox_decomp != NULL)
3415             ptr_lista_destino = ptr_lista_destino->prox_decomp;
3416
3417         return ptr_lista_destino;
3418     }
3419 }
3420
3421 //funcao que pega os dados de um vetor semente e os transfere para um vetor decomp
3422 vetor_decomp *copia_vetor_semente(vetor_sementes *entrada)
3423 {
3424     vetor_sementes *ptr_sementes;
3425     vetor_decomp   *ptr_decomp = NULL;
3426     vetor_decomp   *lista_decomp = NULL;
3427
3428     if (entrada == NULL)
3429     {
3430         return NULL;
3431     }
```

```
3432
3433     ptr_sementes = entrada;
3434     while (ptr_sementes != NULL)
3435     {
3436         ptr_decomp = novo_vetor_decomp();
3437         ptr_decomp->poly_pares = novo_vetor_polinomios();
3438         ptr_decomp->poly_pares->polinomio = &(ptr_sementes->P2);
3439         ptr_decomp->poly_impares = novo_vetor_polinomios();
3440         ptr_decomp->poly_impares->polinomio = &(ptr_sementes->P1);
3441         ptr_decomp->resto_par = copia_lista_expr(ptr_sementes->R2) ;
3442         ptr_decomp->resto_impar = copia_lista_expr(ptr_sementes->R1) ;
3443
3444         //insere o elemento criado na lista
3445         lista_decomp = insere_lista_decomp(lista_decomp, ptr_decomp);
3446
3447         //incrementa o ponteiro
3448         ptr_sementes = ptr_sementes->conjunto_prox;
3449
3450     }
3451
3452     //rebobinar a lista_decomp
3453     while (lista_decomp->ant_decomp != NULL)
3454         lista_decomp = lista_decomp->ant_decomp;
3455
3456     return lista_decomp;
3457 }
3458
3459
3460 //funcao que implementa a decomposicao em si - versao recursiva
3461 vetor_decomp *encontra_decomp(vetor_sementes *entrada, int grau, lista_expr *expr_simplificada, tabela_literais *
         lista_literais)
3462 {
3463     vetor_sementes *secundario_ptr; //ponteiro para os loops internos
3464
3465     vetor_decomp *decomp_atual; //lista que sera gerada dentro do loop atual
3466     vetor_decomp *decomp_referencia; //lista de refrencia para o loop atual
3467     vetor_decomp *lista_decomp = NULL; //elemento manipulado dentro do loop
3468
3469     int total_decomp = 0;
3470
3471
3472     //inicializacao da lista de referencia
3473     decomp_referencia = copia_vetor_semente(entrada);
3474     decomp_atual = decomp_referencia;
3475     decomp_atual = NULL;
3476
3477     //inicializacao do ponteiro de busca secundario, pode ser que o par de polinomios inicial possa combinar com si mesmo
3478     secundario_ptr = entrada;
3479
3480     //loop de construcao das decomposicoes
3481     while (decomp_referencia != NULL)
3482     {
3483         //para cada vetor incial de decomp referencia, encontro todas as decomposicoes que possam ser geradas a partir
         dele
3484         encontra_decomp_recursiva(decomp_referencia, decomp_referencia, &lista_decomp, grau, expr_simplificada,
         lista_literais, &total_decomp );
3485
3486         //apagar o elemento testado e atualizar o ponteiro de referencia
3487         decomp_atual = decomp_referencia->prox_decomp;
3488         destroi_decomp(decomp_referencia);
3489         decomp_referencia = decomp_atual;
3490         if (decomp_referencia!= NULL)
3491             decomp_referencia->ant_decomp = NULL;
3492     }
3493
3494
3495     //rebobinar a lista de decomposicoes
3496     if (lista_decomp != NULL)
3497         while (lista_decomp->ant_decomp != NULL)
3498             lista_decomp = lista_decomp->ant_decomp;
3499
3500     //imprimir numero de decomposicoes
3501     printf("\n\nThe number of valid Translinear decompositions found is: %d", total_decomp);
3502     printf("\n Number of partial fraction operations performed is: %d\n",global_num_parfrac);
3503
3504     //retornar
3505     return lista_decomp;
3506 }
3507
3508
3509 //funcao que pega os dados de um vetor semente e os transfere para um vetor decomp
```

```
3510 vetor_decomp *copia_semente(vetor_sementes *entrada)
3511 {
3512     vetor_decomp   *ptr_decomp = NULL;
3513
3514     ptr_decomp = novo_vetor_decomp();
3515     ptr_decomp->poly_pares = novo_vetor_polinomios();
3516     ptr_decomp->poly_pares->polinomio = &entrada->P2;
3517     ptr_decomp->poly_impares = novo_vetor_polinomios();
3518     ptr_decomp->poly_impares->polinomio = &entrada->P1;
3519     ptr_decomp->resto_par = copia_lista_expr(entrada->R2);
3520     ptr_decomp->resto_impar = copia_lista_expr(entrada->R1);
3521     ptr_decomp->ant_decomp = NULL;
3522     ptr_decomp->prox_decomp = NULL;
3523
3524     return ptr_decomp;
3525 }
3526
3527 vetor_decomp *copia_decomp(vetor_decomp *entrada)
3528 {
3529     vetor_decomp *nova_decomp;
3530     vetor_polinomios *poly_ptr;
3531
3532     nova_decomp = novo_vetor_decomp();
3533     //copio os polinomios do primario em nova_decomp, primeiro a parte par
3534     poly_ptr = entrada->poly_pares;
3535     while (poly_ptr != NULL)
3536     {
3537         insere_polinomio(&(nova_decomp->poly_pares), poly_ptr->polinomio);
3538         poly_ptr = poly_ptr->proximo_polinomio;
3539     }
3540
3541     //copio os polinomios do primario em nova_decomp, depois a parte impar
3542     poly_ptr =  entrada->poly_impares;
3543     while (poly_ptr != NULL)
3544     {
3545         insere_polinomio(&nova_decomp->poly_impares, poly_ptr->polinomio);
3546         poly_ptr = poly_ptr->proximo_polinomio;
3547     }
3548
3549     //copia os outros parametros
3550     nova_decomp->resto_impar = NULL; //copia_lista_expr(entrada->resto_impar); estes parametros serao inseridos dentro da
       funcao recursiva
3551     nova_decomp->resto_par = NULL; //copia_lista_expr(entrada->resto_par);
3552     nova_decomp->ant_decomp = NULL;
3553     nova_decomp->prox_decomp = NULL;
3554
3555     return nova_decomp;
3556 }
3557
3558 int decomp_size(vetor_decomp *entrada)
3559 {
3560     int count = 0;
3561
3562     //inicia com o esqueleto da estrutura
3563     count = sizeof(vetor_decomp);
3564
3565     //soma os polinomios pares
3566     count+= poly_size(entrada->poly_pares);
3567
3568     //soma os polinomios impares
3569     count+= poly_size(entrada->poly_impares);
3570
3571     //soma os restos
3572     count+= expr_size(entrada->resto_par);
3573     count+= expr_size(entrada->resto_impar);
3574
3575     return count;
3576 }
3577
3578 int poly_size(vetor_polinomios *entrada)
3579 {
3580     int count = 0;
3581
3582     //soma o tamanho em memoria de todos os polinomios afrente
3583     if (entrada->proximo_polinomio != NULL)
3584     {
3585         count+= poly_size(entrada->proximo_polinomio);
3586     }
3587      //soma o proprio tamanho
3588     count += sizeof(vetor_polinomios);
3589
```

```
3590     return count;
3591
3592 }
3593
3594 int expr_size(lista_expr *entrada)
3595 {
3596     int count = 0;
3597
3598     //soma a memoria dos proximos monomios
3599     if (entrada->proximo != NULL)
3600     {
3601         count+= expr_size(entrada->proximo);
3602     }
3603
3604     //soma os proprios campos
3605     count+= sizeof(lista_expr);
3606
3607     if (entrada->codigos_numerador != NULL)
3608     {
3609         count += (strlen(entrada->codigos_numerador) + 1)*sizeof(char);
3610     }
3611
3612     if (entrada->codigos_denominador != NULL)
3613     {
3614         count+= expr_size(entrada->codigos_denominador);
3615     }
3616
3617     return count;
3618 }
3619
3620
3621 //Funcao que testa se um par de polinomios pode ser combinado com outro para formar uma decomposicao, retornando a
         decomposicao parcial
3622 void encontra_decomp_recursiva(vetor_decomp *primario, vetor_decomp *secundario, vetor_decomp **retorno, int grau,
         lista_expr *expr_simplificada, tabela_literais *lista_literais, int *total_decomp)
3623 {
3624     lista_expr *Q = NULL;
3625     lista_expr *R1= NULL;
3626     lista_expr *R2= NULL;
3627     lista_expr *R_dummy = NULL;
3628     vetor_decomp *decomp_atual = NULL;
3629     vetor_polinomios *poly_ptr = NULL;
3630     vetor_decomp *ptr_sementes = NULL;
3631     vetor_decomp *ptr_decomp = NULL;
3632     int contador = 0;
3633     int flag = 0;
3634     int flag_first = 0;
3635     int flag_continue = 0;
3636
3637     //aponto para o inicio do vetor sementes
3638     ptr_sementes = secundario;
3639
3640     //inicializo a flag para pular a primeira decomposicao, que e de uma semente consigo mesma
3641     flag_first = 1;
3642     //vou tentar combinar o primario com TODOS os elementos do secundario
3643     while (ptr_sementes != NULL)
3644     {
3645         //pula este bloco na primeira passada
3646         if (!flag_first)
3647         {
3648             R_dummy = NULL;
3649             Q = NULL;
3650             R1 = NULL;
3651             R2 = NULL;
3652             flag_continue = 0;
3653             //testar P1 principal com P1 atual
3654             if(partial_fraction_expansion(primario->resto_impar, primario->poly_impares->polinomio->P, ptr_sementes->
         poly_impares->polinomio->P, &Q,&R1, &R_dummy))
3655             {
3656                 //limpar o R_dummy e o &Q
3657                 destroi_lista_expr_expandida(R_dummy);
3658                 destroi_lista_expr_expandida(Q);
3659                 R_dummy = NULL;
3660                 Q = NULL;
3661
3662                 //testar P2 principal com P2 atual
3663                 if(partial_fraction_expansion(primario->resto_par, primario->poly_pares->polinomio->P, ptr_sementes->
         poly_pares->polinomio->P, &Q,&R2, &R_dummy))
3664                 {
3665                     //limpar o R_dummy e o &Q
3666                     destroi_lista_expr_expandida(R_dummy);
```

```
3667                    destroi_lista_expr_expandida(Q);
3668
3669                    //copio o primario, pois pode ser semente para outras decomposicoes
3670                    decomp_atual = copia_decomp(primario);
3671
3672                    //insiro os polinomios que fazem parte da decomposicao
3673                    //significa que secundario->P1 e o proximo polinomio Par, e seconudario->P2 o proximo polinomio impar
    , e e invertido mesmo
3674                    insere_polinomio(&decomp_atual->poly_pares,ptr_sementes->poly_impares->polinomio);
3675                    insere_polinomio(&decomp_atual->poly_impares,ptr_sementes->poly_pares->polinomio);
3676
3677                    //atualiza Resto par e Resto impar
3678                    decomp_atual->resto_impar = R1;
3679                    decomp_atual->resto_par = R2;
3680
3681                    //quando o numero de polinomios pares e impares for igual ao grau da equacao de entrada pode ser que
    ja tenha terminado
3682                    contador = 0;
3683                    poly_ptr = decomp_atual->poly_pares;
3684                    while (poly_ptr != NULL)
3685                    {
3686                        ++contador;
3687                        poly_ptr = poly_ptr->proximo_polinomio;
3688                    }
3689                    if (contador == grau)
3690                    {
3691                        //testo se a decomposicaO encontrada e valida
3692                        //tirar a prova real
3693                        if(prova_real(decomp_atual, expr_simplificada))
3694                        {
3695                            //ordenar os polinomios antes de inserir a decomposicao
3696                            decomp_atual->poly_pares = ordena_polinomios(decomp_atual->poly_pares);
3697                            decomp_atual->poly_impares = ordena_polinomios(decomp_atual->poly_impares);
3698
3699                            //normalizar os restos->para
3700                            if (fabs(decomp_atual->resto_par->parametro) >= fabs(decomp_atual->resto_impar->parametro))
3701                            {
3702                                decomp_atual->resto_par->parametro = decomp_atual->resto_par->parametro/decomp_atual->
    resto_impar->parametro;
3703                                decomp_atual->resto_impar->parametro = 1.0;
3704
3705                            }
3706                            else
3707                            {
3708                                decomp_atual->resto_impar->parametro = decomp_atual->resto_impar->parametro/decomp_atual
    ->resto_par->parametro;
3709                                decomp_atual->resto_par->parametro = 1.0;
3710                            }
3711
3712                            //alterar sinais para melhor impressao de resultados, priorizando resto par ser positivo:
3713                            if (decomp_atual->resto_par->parametro < 0.0)
3714                            {
3715                                decomp_atual->resto_par->parametro*= -1.0;
3716                                decomp_atual->resto_impar->parametro*= -1.0;
3717                            }
3718
3719                            //procuro por uma decomposicao equivalente no vetor de retorno
3720                            ptr_decomp = *retorno;
3721
3722                            //inicializo uma flag de busca
3723                            flag = 0;
3724                            while (ptr_decomp != NULL)
3725                            {
3726                                //se eu encontro uma decomposicao equivalente, interrompo a busca e nao insiro decomp
    atual no vetro de retorno
3727                                if (compara_decomp(decomp_atual, ptr_decomp))
3728                                {
3729                                    flag = 1;
3730                                    break;
3731                                }
3732                                ptr_decomp = ptr_decomp->ant_decomp;
3733                            }
3734                            //se nao encontrou nenhuma decomposicao equivalente, inserir decomp autal no vetor
3735                            if (!flag)
3736                            {
3737                                *retorno = insere_lista_decomp(*retorno, decomp_atual);
3738                                //ja imprimo a decomposicao encontrada
3739                                imprime_decomposicao(decomp_atual, lista_literais);
3740                                //imcremento as decomp validas
3741                                (*total_decomp)++;
3742                                //sinalizo o flag de que nao e necessario fazer o teste invertido
```

```
3743                             flag_continue = 1;
3744                         }
3745                         else
3746                             destroi_decomp(decomp_atual);
3747 
3748                         //limpa o ponteiro para o proximo teste
3749                         decomp_atual = NULL;
3750 
3751                     }
3752                     else
3753                     {
3754                         destroi_decomp(decomp_atual);
3755                         decomp_atual = NULL;
3756                     }
3757                 }
3758                 else
3759                 {
3760                     //caso contrario, deve-se proceder com a decomposicao recursivamente
3761                     encontra_decomp_recursiva(decomp_atual, ptr_sementes, retorno, grau, expr_simplificada,
     lista_literais, total_decomp);
3762                     //como as decomposicoes validas serao adicionadas no if acima, quando o programa chegar aqui,
     significa
3763                     //que nao preciso mais de decomp_atual;
3764                     destroi_decomp(decomp_atual);
3765                     decomp_atual = NULL;
3766                 }
3767 
3768                 //se chegou aqui
3769 
3770                 //se ja tiver encontrado uma decomposicao valida neste passo,, signigica que o segundo teste, com os
     polinomios invertidos, nao e necessario
3771                 if (flag_continue)
3772                 {
3773                     //atualiza o ponteiro
3774                     ptr_sementes = ptr_sementes->prox_decomp;
3775                     continue;
3776                 }
3777             }
3778             else
3779             {
3780                 //limpar tudo
3781                 destroi_lista_expr_expandida(R_dummy);
3782                 destroi_lista_expr_expandida(Q);
3783                 destroi_lista_expr_expandida(R1);
3784                 destroi_lista_expr_expandida(R2);
3785             }
3786         }
3787         else
3788         {
3789             //limpar tudo
3790             destroi_lista_expr_expandida(R_dummy);
3791             destroi_lista_expr_expandida(Q);
3792             destroi_lista_expr_expandida(R1);
3793 
3794         }
3795     }
3796     else
3797     {
3798         //reseta o flag de primeira passada
3799         flag_first = 0;
3800     }
3801 
3802     //limpar as variaveis de retorno
3803     Q = NULL;
3804     R1= NULL;
3805     R2= NULL;
3806     R_dummy = NULL;
3807     //testar tambem P1 principal com P2 atual
3808     if(partial_fraction_expansion(primario->resto_impar, primario->poly_impares->polinomio->P, ptr_sementes->
     poly_pares->polinomio->P, &Q,&R1, &R_dummy))
3809     {
3810         //if dummie para por um breakpoint exatamente onde esta dando pau no windows
3811         //limpar o R_dummy e o &Q
3812         destroi_lista_expr_expandida(R_dummy);
3813         destroi_lista_expr_expandida(Q);
3814         R_dummy = NULL;
3815         Q = NULL;
3816 
3817         //testar P2 principal com P1 atual
3818         if(partial_fraction_expansion(primario->resto_par, primario->poly_pares->polinomio->P, ptr_sementes->
     poly_impares->polinomio->P, &Q,&R2, &R_dummy))
```

```
3819            {
3820                //limpar o R_dummy e o &Q
3821                destroi_lista_expr_expandida(R_dummy);
3822                destroi_lista_expr_expandida(Q);
3823                R_dummy = NULL;
3824                Q = NULL;
3825
3826                //copio o primario, pois pode ser semente para outras decomposicoes
3827                decomp_atual = copia_decomp(primario);
3828
3829                //insiro os polinomios que fazem parte da decomposicao
3830                //significa que secundario->P1 e o proximo polinomio Par, e seconudario->P2 o proximo polinomio impar, e
        e invertido mesmo
3831                insere_polinomio(&decomp_atual->poly_pares,ptr_sementes->poly_pares->polinomio);
3832                insere_polinomio(&decomp_atual->poly_impares,ptr_sementes->poly_impares->polinomio);
3833
3834                //atualiza Resto par e Resto impar
3835                decomp_atual->resto_impar = R1;
3836                decomp_atual->resto_par = R2;
3837
3838                //quando o numero de polinomios pares e impares for igual ao grau da equacao de entrada pode ser que ja
        tenha terminado
3839                contador = 0;
3840                poly_ptr = decomp_atual->poly_pares;
3841                while (poly_ptr != NULL)
3842                {
3843                    ++contador;
3844                    poly_ptr = poly_ptr->proximo_polinomio;
3845                }
3846                if (contador == grau)
3847                {
3848                    //testo se a decomposicap encontrada e valida
3849
3850                    //tirar a prova real
3851                    if(prova_real(decomp_atual, expr_simplificada))
3852                    {
3853                        //ordeno o polinomio antes de inserir
3854                        decomp_atual->poly_pares = ordena_polinomios(decomp_atual->poly_pares);
3855                        decomp_atual->poly_impares = ordena_polinomios(decomp_atual->poly_impares);
3856
3857
3858                        //normalizar os restos->para
3859                        if (fabs(decomp_atual->resto_par->parametro) >= fabs(decomp_atual->resto_impar->parametro))
3860                        {
3861                            decomp_atual->resto_par->parametro = decomp_atual->resto_par->parametro/decomp_atual->
        resto_impar->parametro;
3862                            decomp_atual->resto_impar->parametro = 1.0;
3863
3864                        }
3865                        else
3866                        {
3867                            decomp_atual->resto_impar->parametro = decomp_atual->resto_impar->parametro/decomp_atual->
        resto_par->parametro;
3868                            decomp_atual->resto_par->parametro = 1.0;
3869                        }
3870
3871                        //alterar sinais para melhor impressao de resultados, priorizando resto par ser positivo:
3872                        if (decomp_atual->resto_par->parametro < 0.0)
3873                        {
3874                            decomp_atual->resto_par->parametro*= -1.0;
3875                            decomp_atual->resto_impar->parametro*= -1.0;
3876                        }
3877
3878                        //procuro por uma decomposicao equivalente no vetor de retorno
3879                        ptr_decomp = *retorno;
3880
3881                        //inicializo uma flag de busca
3882                        flag = 0;
3883                        while (ptr_decomp != NULL)
3884                        {
3885                            //se eu encontro uma decomposicao equivalente, interrompo a busca e nao insiro decomp atual
        no vetro de retorno
3886                            if (compara_decomp(decomp_atual, ptr_decomp))
3887                            {
3888                                flag = 1;
3889                                break;
3890                            }
3891                            ptr_decomp = ptr_decomp->ant_decomp;
3892                        }
3893
3894                        //se nao encontrou nenhuma decomposicao equivalente, inserir decomp autal no vetor
```

```
3895                        if (!flag)
3896                        {
3897                            *retorno = insere_lista_decomp(*retorno, decomp_atual);
3898                            //ja imprimo a decomposicao encontrada
3899                            imprime_decomposicao(decomp_atual, lista_literais);
3900                            //imcremento as decomp validas
3901                            (*total_decomp)++;
3902
3903                        }
3904                        else
3905                            destroi_decomp(decomp_atual);
3906
3907                        //limpa o ponteiro para o proximo teste
3908                        decomp_atual = NULL;
3909                        }
3910                    else
3911                    {
3912                        destroi_decomp(decomp_atual);
3913                        decomp_atual = NULL;
3914                    }
3915                }
3916                else
3917                {
3918                    //caso contrario, deve-se proceder com a decomposicao recursivamente
3919                    encontra_decomp_recursiva(decomp_atual, ptr_sementes, retorno, grau, expr_simplificada,lista_literais
         , total_decomp);
3920
3921                    //como as decomposicoes validas serao adicionadas no if acima, quando o programa chegar aqui,
         significa
3922                    //que nao preciso mais de decomp_atual;
3923                    destroi_decomp(decomp_atual);
3924                    decomp_atual = NULL;
3925                }
3926
3927            }
3928            else
3929            {
3930                //limpar tudo
3931                destroi_lista_expr_expandida(R_dummy);
3932                destroi_lista_expr_expandida(R2);
3933                destroi_lista_expr_expandida(R1);
3934                destroi_lista_expr_expandida(Q);
3935
3936            }
3937        }
3938        else
3939        {
3940            //limpar tudo
3941            destroi_lista_expr_expandida(R_dummy);
3942            destroi_lista_expr_expandida(Q);
3943            destroi_lista_expr_expandida(R1);
3944        }
3945        //return decomp_atual;
3946        Q = NULL;
3947        R1= NULL;
3948        R2= NULL;
3949        R_dummy = NULL;
3950
3951        //atualiza o ponteiro
3952        ptr_sementes = ptr_sementes->prox_decomp;
3953    }
3954 }
3955
3956 //funcao que elimina as decomp redundantes nao previsiveis
3957 void elimina_decomp_redundantes(vetor_decomp *entrada)
3958 {
3959     vetor_decomp *ptr_entrada = entrada;
3960     vetor_decomp *ptr_aux, *ptr_remove;
3961     int flag;
3962
3963     //primeiro reordeno os polinomios pares e impares de cada decomp
3964     while (ptr_entrada!= NULL)
3965     {
3966         //reordeno os polinomios pares e impares
3967         ptr_entrada->poly_pares = ordena_polinomios(ptr_entrada->poly_pares);
3968         ptr_entrada->poly_impares = ordena_polinomios(ptr_entrada->poly_impares);
3969
3970         ptr_entrada = ptr_entrada->prox_decomp;
3971     }
3972
3973     ptr_entrada = entrada;
```

```
3974     //percorro toda a lista de decomposicoes, comparando com todas abaixo
3975     while (ptr_entrada!= NULL)
3976     {
3977         ptr_aux = ptr_entrada->prox_decomp;
3978         while(ptr_aux != NULL)
3979         {
3980             flag = compara_decomp(ptr_entrada, ptr_aux);
3981 
3982             //finalmente, se a falg for 1, e porque os polinomios sao redundantes, entao posso excluir o polinomio
         auxiliar
3983             if (flag)
3984             {
3985                 //marco o ponteiro a ser removido
3986                 ptr_remove = ptr_aux;
3987 
3988                 //ja incremento o ponteiro auxiliar para a proxima iteracao
3989                 ptr_aux = ptr_aux->prox_decomp;
3990 
3991                 //removo p_remove
3992                 ptr_remove->ant_decomp->prox_decomp = ptr_remove->prox_decomp;
3993                 if (ptr_remove->prox_decomp != NULL)
3994                     ptr_remove->prox_decomp->ant_decomp = ptr_remove->ant_decomp;
3995                 destroi_decomp(ptr_remove);
3996             }
3997             else
3998                 ptr_aux = ptr_aux->prox_decomp;
3999         }
4000 
4001         ptr_entrada = ptr_entrada->prox_decomp;
4002 
4003     }
4004 }
4005 //ordena uma lista de polinomiosem ordem crescente
4006 vetor_polinomios *ordena_polinomios(vetor_polinomios *entrada)
4007 {
4008     vetor_polinomios *ptr, *aux;
4009 
4010     ptr = entrada;
4011     while (ptr->proximo_polinomio!= NULL)
4012     {
4013         aux = ptr->proximo_polinomio;
4014         if (aux->polinomio->id < ptr->polinomio->id)
4015         {
4016             //realizo a troca entre ptr e aux
4017             ptr->proximo_polinomio = aux->proximo_polinomio;
4018             aux->polinomio_anterior = ptr->polinomio_anterior;
4019 
4020             if (ptr->polinomio_anterior != NULL)
4021                 ptr->polinomio_anterior->proximo_polinomio = aux;
4022             if (aux->proximo_polinomio != NULL)
4023                 aux->proximo_polinomio->polinomio_anterior = ptr;
4024             aux->proximo_polinomio = ptr;
4025             ptr->polinomio_anterior = aux;
4026 
4027             //como e um bubblesort, devo voltar ao inicio da lista
4028             while (ptr->polinomio_anterior != NULL)
4029                 ptr = ptr->polinomio_anterior;
4030         }
4031         else
4032             ptr = ptr->proximo_polinomio;
4033 
4034     }
4035     //ao final do processo, basta rebobinar o polinomio
4036     while (ptr->polinomio_anterior != NULL)
4037         ptr = ptr->polinomio_anterior;
4038 
4039     return ptr;
4040 }
4041 
4042 //funcao que imprime uma decomposicao
4043 void imprime_decomposicao(vetor_decomp *decomposicao, tabela_literais *lista_literais)
4044 {
4045     vetor_polinomios *percorre_polinomios;
4046 
4047     printf("\n %+3.2f*",decomposicao->resto_par->parametro);
4048     percorre_polinomios = decomposicao->poly_impares;
4049     while (percorre_polinomios!= NULL)
4050     {
4051         printf("(");
4052         imprime_lista_expr_expandida(percorre_polinomios->polinomio->P, lista_literais);
4053         printf(")");
```

```
4054         percorre_polinomios = percorre_polinomios->proximo_polinomio;
4055     }
4056
4057     printf(" %+3.2f*",decomposicao->resto_impar->parametro);
4058     percorre_polinomios = decomposicao->poly_pares;
4059     while (percorre_polinomios!= NULL)
4060     {
4061         printf("(");
4062         imprime_lista_expr_expandida(percorre_polinomios->polinomio->P, lista_literais);
4063         printf(")");
4064         percorre_polinomios = percorre_polinomios->proximo_polinomio;
4065     }
4066 }
4067
4068 //funcao que retorna 1 caso as decomposicoes sejam equivalentes e 0 caso nao sejam
4069 int compara_decomp(vetor_decomp *decomp1, vetor_decomp *decomp2)
4070 {
4071
4072     vetor_polinomios *ptr_impar1, *ptr_impar2;
4073     vetor_polinomios *ptr_par1, *ptr_par2;
4074
4075     int flag;
4076
4077     //comparo elemento a elemento de cada conjunto de polinomios para ver se sao iguais
4078     ptr_impar1 = decomp1->poly_impares;
4079     ptr_impar2 = decomp2->poly_impares;
4080     ptr_par1 = decomp1->poly_pares;
4081     ptr_par2 = decomp2->poly_pares;
4082
4083     //inicializo minha flag
4084     flag = 1;
4085
4086     while (ptr_par1 != NULL)
4087     {
4088         //se algum for diferente, limpa a flag
4089         if (ptr_par1->polinomio->id != ptr_par2->polinomio->id)
4090             flag = 0;
4091         if (ptr_impar1->polinomio->id != ptr_impar2->polinomio->id)
4092             flag = 0;
4093
4094         //atualiza os dois ponteiros
4095         ptr_par1 = ptr_par1->proximo_polinomio;
4096         ptr_par2 = ptr_par2->proximo_polinomio;
4097         ptr_impar1 = ptr_impar1->proximo_polinomio;
4098         ptr_impar2 = ptr_impar2->proximo_polinomio;
4099     }
4100
4101     //se a flag for 0, testar com os vetores trocados
4102     if (!flag)
4103     {
4104         //comparo elemento a elemento de cada conjunto de polinomios para ver se sao iguais
4105         ptr_impar1 = decomp1->poly_impares;
4106         ptr_impar2 = decomp2->poly_impares;
4107         ptr_par1 = decomp1->poly_pares;
4108         ptr_par2 = decomp2->poly_pares;
4109
4110         //inicializo minha flag
4111         flag = 1;
4112
4113         while (ptr_par1 != NULL)
4114         {
4115             //se algum for diferente, limpa a flag
4116             if (ptr_par1->polinomio->id != ptr_impar2->polinomio->id)
4117                 flag = 0;
4118             if (ptr_impar1->polinomio->id != ptr_par2->polinomio->id)
4119                 flag = 0;
4120
4121             //atualiza os dois ponteiros
4122             ptr_par1 = ptr_par1->proximo_polinomio;
4123             ptr_par2 = ptr_par2->proximo_polinomio;
4124             ptr_impar1 = ptr_impar1->proximo_polinomio;
4125             ptr_impar2 = ptr_impar2->proximo_polinomio;
4126         }
4127     }
4128
4129     return flag;
4130
4131 }
4132
4133 vetor_polinomios *reconta_polinomios(vetor_sementes *lista_sementes,vetor_polinomios *lista_polinomios)
4134 {
```

```
4135     vetor_polinomios *p_lista_polinomios;
4136     int id_impar = 0;
4137     int id_par = 0;
4138     vetor_sementes  *p_lista_sementes=lista_sementes;
4139 
4140     //percorrer todo o vetor sementes
4141     while (p_lista_sementes != NULL)
4142     {
4143         //procura cada polinomio da semente na lista de polinomios e seta a flag de aprovado se o encontrar;
4144         id_par = p_lista_sementes->P1.id;
4145         id_impar = p_lista_sementes->P2.id;
4146 
4147         //inicializa a lista de polinomios
4148         p_lista_polinomios = lista_polinomios;
4149         while (p_lista_polinomios!= NULL)
4150         {
4151             //a cada polinomio da lista, se o id for igual a P1 ou P2, seta a flag de aprovado
4152             if (p_lista_polinomios->polinomio->id == id_impar || p_lista_polinomios->polinomio->id == id_par)
4153             {
4154                 p_lista_polinomios->polinomio->flag_approved = 1;
4155             }
4156 
4157             p_lista_polinomios = p_lista_polinomios->proximo_polinomio;
4158         }
4159 
4160         //incrementa ponteiro
4161         p_lista_sementes = p_lista_sementes->conjunto_prox;
4162     }
4163 
4164     //apos setar flag todos os polinomios que podem fazer parte de alguma decomposicao, remover aqueles que nao tem a
      flag setada
4165     p_lista_polinomios = lista_polinomios;
4166     while (1)
4167     {
4168         if (!p_lista_polinomios->polinomio->flag_approved)
4169         {
4170             p_lista_polinomios = remove_polinomio(p_lista_polinomios);
4171         }
4172         else if (p_lista_polinomios->proximo_polinomio!= NULL)
4173         {
4174             p_lista_polinomios = p_lista_polinomios->proximo_polinomio;
4175         }
4176         else
4177             break;
4178     }
4179 
4180     //reboboina a lista
4181     while (p_lista_polinomios->polinomio_anterior != NULL)
4182     {
4183         p_lista_polinomios = p_lista_polinomios->polinomio_anterior;
4184     }
4185 
4186     return p_lista_polinomios;
4187 }
4188 
4189 vetor_polinomios *remove_polinomios_nao_pertencentes(vetor_decomp *lista_decomp,vetor_polinomios *lista_polinomios)
4190 {
4191     vetor_polinomios *p_lista_polinomios;
4192     int id_impar = 0;
4193     int id_par = 0;
4194     int ok_par = 0;
4195     int ok_impar = 0;
4196     vetor_decomp  *p_lista_decomp =lista_decomp;
4197     vetor_polinomios *p_pares, *p_impares;
4198 
4199     //If a null list is given, return NULL
4200     if (lista_decomp == NULL)
4201     {
4202         return NULL;
4203     }
4204 
4205     //percorrer todo o vetor de decomposicoes
4206     while (p_lista_decomp != NULL)
4207     {
4208         //procura cada polinomio do lado par e do lado impar na lista de polinomios, e os que forem sendo encontrados,
      mudar o flag de 1 para 0
4209         p_impares = p_lista_decomp->poly_impares;
4210         p_pares = p_lista_decomp->poly_pares;
4211 
4212         //como o numero de polinoimos do lado par e igual ao do lado impar em uma decomposicao encontrada, pode-se
      percorrer apenas um enquanto testa-se ambos
```

```
4213        while (p_impares != NULL)
4214        {
4215            id_par = p_pares->polinomio->id;
4216            id_impar = p_impares->polinomio->id;
4217
4218            //procuro ambos na lista de polinomios
4219            //inicializa a lista de polinomios
4220            p_lista_polinomios = lista_polinomios;
4221            ok_par = 0;
4222            ok_impar = 0;
4223            while (p_lista_polinomios!= NULL)
4224            {
4225                //a cada polinomio da lista, se o id for igual ao polinomio par, re-seta a flag de aprovado
4226                if (p_lista_polinomios->polinomio->id == id_par)
4227                {
4228                    p_lista_polinomios->polinomio->flag_approved = 0;
4229                    ok_par = 1;
4230                }
4231
4232                //a cada polinomio da lista, se o id for igual ao polinomio impar re-seta a flag de aprovado
4233                if (p_lista_polinomios->polinomio->id == id_impar )
4234                {
4235                    p_lista_polinomios->polinomio->flag_approved = 0;
4236                    ok_impar = 1;
4237                }
4238
4239                //se ja encontrou os dois, nao precisa mais continuar
4240                if (ok_impar && ok_par)
4241                {
4242                    break;
4243                }
4244
4245                p_lista_polinomios = p_lista_polinomios->proximo_polinomio;
4246            }
4247
4248            //incrementa os ponteiros
4249            p_impares = p_impares->proximo_polinomio;
4250            p_pares = p_pares->proximo_polinomio;
4251        }
4252
4253
4254        //incrementa ponteiro
4255        p_lista_decomp = p_lista_decomp->prox_decomp;
4256    }
4257
4258    //apos re-setar flag todos os polinomios que fizeram parte de alguma decomposicao, remover aqueles que tem a flag
        setada
4259    p_lista_polinomios = lista_polinomios;
4260    while (1)
4261    {
4262        if (p_lista_polinomios->polinomio->flag_approved)
4263        {
4264            p_lista_polinomios = remove_polinomio(p_lista_polinomios);
4265        }
4266        else if (p_lista_polinomios->proximo_polinomio!= NULL)
4267        {
4268            p_lista_polinomios = p_lista_polinomios->proximo_polinomio;
4269        }
4270        else
4271            break;
4272    }
4273
4274    //reboboina a lista
4275    if (p_lista_polinomios!= NULL)
4276    {
4277        while (p_lista_polinomios->polinomio_anterior != NULL)
4278        {
4279            p_lista_polinomios = p_lista_polinomios->polinomio_anterior;
4280        }
4281    }
4282
4283
4284    return p_lista_polinomios;
4285 }
4286
4287 //funcao que implementa a decomposicao em si - versao recursiva
4288 vetor_decomp *encontra_decomp_mulder(vetor_polinomios *entrada, int grau, lista_expr *expr_simplificada, tabela_literais
        *lista_literais)
4289 {
4290    vetor_polinomios *primario_ptr, *secundario_ptr; //ponteiro para os loops internos
4291
```

```
4292    vetor_decomp *decomp_atual; //lista que sera gerada dentro do loop atual
4293    vetor_decomp *lista_decomp = NULL; //elemento manipulado dentro do loop
4294    vetor_decomp_simple *poly_pares = NULL, *poly_impares = NULL;
4295    lista_expr *Q=NULL, *num_a=NULL, *num_b=NULL;
4296
4297    int total_decomp = 0;
4298
4299    //inicializacao da lista de referencia
4300    primario_ptr = entrada;
4301    decomp_atual = NULL;
4302
4303    //loop de construcao das decomposicoes
4304    while (primario_ptr != NULL)
4305    {
4306        //inicializa o proximo ponteiro de busca
4307        secundario_ptr = entrada->proximo_polinomio;
4308        while(secundario_ptr != NULL)
4309        {
4310            //inicializa variaveis
4311            Q = NULL;
4312            num_a = NULL;
4313            num_b = NULL;
4314            //tenta realizar uma expansao em fracao parcialentre primario e secundario
4315            if (partial_fraction_expansion(expr_simplificada, primario_ptr->polinomio->P, secundario_ptr->polinomio->P, &
         Q, &num_a, &num_b))
4316            {
4317                //disparo a divisao recursiva por 2 BP,s aqui
4318                encontra_decomp_parcial(NULL,primario_ptr->polinomio, num_a, entrada, grau, &poly_pares);
4319
4320
4321                //se a rotina de encontrar decomposicoes parciais nao encontrar nada, o procedimento falhou
4322                if (poly_pares != NULL)
4323                {
4324                    //rebobina os polinomios pares
4325                    while (poly_pares->ant_decomp != NULL)
4326                    {
4327                        poly_pares = poly_pares->ant_decomp;
4328                    }
4329
4330                    encontra_decomp_parcial(NULL,secundario_ptr->polinomio, num_b, entrada, grau, &poly_impares);
4331
4332                    //de posse dos lados pares e dos lados impares, encontrar quais combinacoes geram uma decomposicao
         valida
4333                    if (poly_impares != NULL)
4334                    {
4335                        //rebobina os polinomios impares
4336                        while (poly_impares->ant_decomp != NULL)
4337                        {
4338                            poly_impares = poly_impares->ant_decomp;
4339                        }
4340
4341                        combina_decomp_mulder(poly_impares, poly_pares, secundario_ptr->polinomio, primario_ptr->
         polinomio, expr_simplificada, &lista_decomp, &total_decomp, lista_literais);
4342
4343                    }
4344                }
4345
4346                destroi_vetor_decomp_simples(poly_pares);
4347                destroi_vetor_decomp_simples(poly_impares);
4348
4349                poly_pares = NULL;
4350                poly_impares = NULL;
4351            }
4352
4353            //limpa as variaveis para a proxima passagem
4354            destroi_lista_expr_expandida(Q);
4355            destroi_lista_expr_expandida(num_a);
4356            destroi_lista_expr_expandida(num_b);
4357
4358            //atualizo ponteiro
4359            secundario_ptr = secundario_ptr->proximo_polinomio;
4360        }
4361
4362        //atualizo ponteiro
4363        primario_ptr = primario_ptr->proximo_polinomio;
4364    }
4365
4366    //rebobinar a lista de decomposicoes
4367    if (lista_decomp != NULL)
4368        while (lista_decomp->ant_decomp != NULL)
4369            lista_decomp = lista_decomp->ant_decomp;
```

```
4370
4371     //imprimir numero de decomposicoes
4372     printf("\n\nThe number of valid Translinear decompositions found is: %d", total_decomp);
4373     printf("\n Number of partial fraction operations performed is: %d\n",global_num_parfrac);
4374
4375     //retornar
4376     return lista_decomp;
4377 }
4378
4379 void combina_decomp_mulder(vetor_decomp_simple *decomp_impares, vetor_decomp_simple *decomp_pares, polinomio *base_par,
      polinomio *base_impar, lista_expr *expr_simplificada, vetor_decomp **lista_decomp, int *total_decomp,
      tabela_literais *lista_literais)
4380 {
4381     vetor_decomp *decomp_atual, *ptr_decomp;
4382     vetor_decomp_simple *p_decomp_pares = NULL;
4383     vetor_decomp_simple *p_decomp_impares = NULL;
4384     vetor_polinomios *poly_ptr;
4385
4386     int flag;
4387
4388
4389     //para cada decomp_par, percorrer todas as decomp_impares
4390     p_decomp_pares = decomp_pares;
4391     while (p_decomp_pares != NULL)
4392     {
4393         //inicializacao de variaveis
4394         p_decomp_impares = decomp_impares;
4395         while (p_decomp_impares != NULL)
4396         {
4397             //reescrita do procedimento
4398             //crio uma decomp com os polys pares e com polys imppares
4399             //copio o primario, pois pode ser semente para outras decomposicoes
4400             decomp_atual = novo_vetor_decomp();
4401
4402             //copio a base par
4403             insere_polinomio(&(decomp_atual->poly_pares), base_par);
4404
4405             //copio o resto dos polinoios pares
4406             poly_ptr = p_decomp_pares->polinomios;
4407             while (poly_ptr != NULL)
4408             {
4409                 insere_polinomio(&(decomp_atual->poly_pares), poly_ptr->polinomio);
4410                 poly_ptr = poly_ptr->proximo_polinomio;
4411             }
4412
4413             //copio a base impar
4414             insere_polinomio(&(decomp_atual->poly_impares), base_impar);
4415
4416             //copio o resto dos polinoios impares
4417             poly_ptr = p_decomp_impares->polinomios;
4418             while (poly_ptr != NULL)
4419             {
4420                 insere_polinomio(&(decomp_atual->poly_impares), poly_ptr->polinomio);
4421                 poly_ptr = poly_ptr->proximo_polinomio;
4422             }
4423
4424             //copia os outros parametros
4425             decomp_atual->resto_impar = copia_lista_expr(p_decomp_pares->resto);
4426             decomp_atual->resto_par = copia_lista_expr(p_decomp_impares->resto);
4427
4428             decomp_atual->ant_decomp = NULL;
4429             decomp_atual->prox_decomp = NULL;
4430
4431             //tiro a prova real
4432             if (prova_real(decomp_atual, expr_simplificada))
4433             {
4434                 //ordeno o polinomio antes de inserir
4435                 decomp_atual->poly_pares = ordena_polinomios(decomp_atual->poly_pares);
4436                 decomp_atual->poly_impares = ordena_polinomios(decomp_atual->poly_impares);
4437
4438                 //testo se esta decomposicao encontrada ja nao existe
4439                 //procuro por uma decomposicao equivalente no vetor de retorno
4440                 ptr_decomp = *lista_decomp;
4441
4442                 //inicializo uma flag de busca
4443                 flag = 0;
4444                 while (ptr_decomp != NULL)
4445                 {
4446                     //se eu encontro uma decomposicao equivalente, interrompo a busca e nao insiro decomp atual no vetro
      de retorno
4447                     if (compara_decomp(decomp_atual, ptr_decomp))
```

```
4448                    {
4449                        flag = 1;
4450                        break;
4451                    }
4452                    ptr_decomp = ptr_decomp->ant_decomp;
4453                }
4454
4455                //se nao encontrou nenhuma decomposicao equivalente, inserir decomp autal no vetor
4456                if (!flag)
4457                {
4458
4459                    //primeiro normalizar os restos
4460                    if (fabs(decomp_atual->resto_par->parametro) >= fabs(decomp_atual->resto_impar->parametro))
4461                    {
4462                        decomp_atual->resto_par->parametro = decomp_atual->resto_par->parametro/decomp_atual->resto_impar
        ->parametro;
4463                        decomp_atual->resto_impar->parametro = 1.0;
4464
4465                    }
4466                    else
4467                    {
4468                        decomp_atual->resto_impar->parametro = decomp_atual->resto_impar->parametro/decomp_atual->
        resto_par->parametro;
4469                        decomp_atual->resto_par->parametro = 1.0;
4470                    }
4471
4472                    //alterar sinais para melhor impressao de resultados, priorizando resto par ser positivo:
4473                    if (decomp_atual->resto_par->parametro < 0.0)
4474                    {
4475                        decomp_atual->resto_par->parametro*= -1.0;
4476                        decomp_atual->resto_impar->parametro*= -1.0;
4477                    }
4478
4479                    //ja imprimo a decomposicao encontrada
4480                    imprime_decomposicao(decomp_atual, lista_literais);
4481                    *lista_decomp = insere_lista_decomp(*lista_decomp, decomp_atual);
4482                    decomp_atual = NULL;
4483
4484                    //imcremento as decomp validas
4485                    (*total_decomp)++;
4486
4487                }
4488                else
4489                {
4490                    destroi_decomp(decomp_atual);
4491                    //limpa o ponteiro para o proximo teste
4492                    decomp_atual = NULL;
4493                }
4494
4495
4496            }
4497            else
4498            {
4499                //limppo as variaveis
4500                destroi_decomp(decomp_atual);
4501                decomp_atual = NULL;
4502            }
4503
4504            //atualizo o ponteiro
4505            p_decomp_impares = p_decomp_impares->prox_decomp;
4506        }
4507
4508        //atualizo o ponteiro
4509        p_decomp_pares = p_decomp_pares->prox_decomp;
4510    }
4511
4512 }
4513
4514 void encontra_decomp_parcial(vetor_decomp_simple *poly_acumulados, polinomio *base, lista_expr *resto, vetor_polinomios *
        lista_polinomios, int grau, vetor_decomp_simple **retorno)
4515 {
4516    vetor_polinomios *ptr_polinomios = lista_polinomios;
4517    lista_expr *Q, *R, *R_dummy;
4518    vetor_decomp_simple *decomp_atual;
4519    int contador;
4520    vetor_polinomios *poly_ptr;
4521
4522    //testa a base com todos os polinomios, excluindo-se a propria base
4523    while (ptr_polinomios != NULL)
4524    {
4525        //testa primeiro se o polinomio nao e igual a base
```

```
4526        if (base->id != ptr_polinomios->polinomio->id)
4527        {
4528            //limpar as variaveis
4529            Q = NULL;
4530            R = NULL;
4531            R_dummy = NULL;
4532            //testar a base principal com o polinomio atual
4533            if(partial_fraction_expansion(resto, base->P, ptr_polinomios->polinomio->P, &Q,&R, &R_dummy))
4534            {
4535                //limpar o R_dummy e o &Q
4536                destroi_lista_expr_expandida(R_dummy);
4537                destroi_lista_expr_expandida(Q);
4538
4539                //copio o primario, pois pode ser semente para outras decomposicoes
4540                decomp_atual = copia_decomp_simples(poly_acumulados);
4541
4542                //insiro os polinomios que fazem parte da decomposicao
4543                insere_polinomio(&decomp_atual->polinomios,ptr_polinomios->polinomio);
4544
4545                //atualiza Resto par e Resto impar
4546                decomp_atual->resto = R;
4547
4548                //quando o numero de polinomios acumulados for igual ao grau menos 1 da equacao de entrada pode ser que
        ja tenha terminado
4549                contador = 0;
4550                poly_ptr = decomp_atual->polinomios;
4551                while (poly_ptr != NULL)
4552                {
4553                    ++contador;
4554                    poly_ptr = poly_ptr->proximo_polinomio;
4555                }
4556
4557                //somo um ao contador
4558                contador++;
4559                //vejo se ja terminou
4560                if (contador == grau)
4561                {
4562                    //terminou de fazer uma decomposicao parcial, inseri-la no vetor de decomposicoes totais
4563                    *retorno = insere_decomp_simples(*retorno, decomp_atual);
4564
4565                }
4566                //caso contrario, prosseguir com as decomposicoes parciais de forma recursiva
4567                else
4568                {
4569                    //caso contrario, deve-se proceder com a decomposicao recursivamente
4570                    encontra_decomp_parcial(decomp_atual, base, R, ptr_polinomios,grau,retorno);
4571                    //como as decomposicoes validas serao adicionadas no if acima, quando o programa chegar aqui,
        significa
4572                    //que nao preciso mais de decomp_atual;
4573                    destroi_decomp_simples(decomp_atual);
4574                    decomp_atual = NULL;
4575                }
4576            }
4577        }
4578
4579        //incrementa o ponteiro
4580        ptr_polinomios = ptr_polinomios->proximo_polinomio;
4581    }
4582 }
4583
4584 vetor_decomp_simple *copia_decomp_simples(vetor_decomp_simple *entrada)
4585 {
4586     vetor_decomp_simple *nova_decomp;
4587     vetor_polinomios *poly_ptr;
4588
4589     nova_decomp = novo_vetor_decomp_simples();
4590     if (entrada != NULL)
4591     {
4592         //copio os polinomios do primario em nova_decomp
4593         poly_ptr = entrada->polinomios;
4594         while (poly_ptr != NULL)
4595         {
4596             insere_polinomio(&(nova_decomp->polinomios), poly_ptr->polinomio);
4597             poly_ptr = poly_ptr->proximo_polinomio;
4598         }
4599     }
4600     else
4601     {
4602         nova_decomp->polinomios = NULL;
4603     }
4604
```

```
4605     //copia os outros parametros
4606     nova_decomp->resto = NULL;
4607     nova_decomp->ant_decomp = NULL;
4608     nova_decomp->prox_decomp = NULL;
4609
4610     return nova_decomp;
4611
4612 }
4613
4614 //insere uma lista de decomposicoes dentro de outra, retornando um ponteiro para o final dela
4615 vetor_decomp_simple *insere_decomp_simples(vetor_decomp_simple *lista_destino, vetor_decomp_simple *lista_origem)
4616 {
4617     vetor_decomp_simple *ptr_lista_origem, *ptr_lista_destino;
4618
4619     ptr_lista_origem = lista_origem;
4620     if (lista_destino == NULL)
4621     {
4622         //aponto para o final da lista a ser inserida
4623         while (ptr_lista_origem->prox_decomp != NULL)
4624             ptr_lista_origem = ptr_lista_origem->prox_decomp;
4625
4626         //retorno o fim da lista destino como o fim da lista de origem
4627         return ptr_lista_origem;
4628
4629     }
4630     else
4631     {
4632         ptr_lista_destino = lista_destino;
4633         //avancar o ponteiro para fim da lista destino
4634         while (ptr_lista_destino->prox_decomp != NULL)
4635             ptr_lista_destino = ptr_lista_destino->prox_decomp;
4636
4637         //rebobinar o ponteiro da lista de origem
4638         while (ptr_lista_origem->ant_decomp != NULL)
4639             ptr_lista_origem = ptr_lista_origem->ant_decomp;
4640
4641         //ligar as 2
4642         ptr_lista_destino->prox_decomp = ptr_lista_origem;
4643         ptr_lista_origem->ant_decomp = ptr_lista_destino;
4644
4645         //avancar o ponteiro ate o final
4646         while (ptr_lista_destino->prox_decomp != NULL)
4647             ptr_lista_destino = ptr_lista_destino->prox_decomp;
4648
4649         return ptr_lista_destino;
4650     }
4651 }
4652
4653 void destroi_decomp_simples(vetor_decomp_simple *entrada)
4654 {
4655     if (entrada->resto != NULL)
4656         destroi_lista_expr_expandida(entrada->resto);
4657     //destroi os vetores de polinomios
4658     if (entrada->polinomios != NULL)
4659         destroi_vetor_polinomios(entrada->polinomios);
4660
4661     free(entrada);
4662 }
4663
4664 vetor_decomp_simple *novo_vetor_decomp_simples(void)
4665 {
4666     vetor_decomp_simple *novo;
4667     novo = (vetor_decomp_simple *)malloc(sizeof(vetor_decomp_simple));
4668     novo->polinomios= NULL;
4669     novo->resto = NULL;
4670
4671     novo->prox_decomp = NULL;
4672     novo->ant_decomp = NULL;
4673
4674     return novo;
4675 }
4676
4677 void destroi_vetor_decomp_simples(vetor_decomp_simple *entrada)
4678 {
4679     if (entrada == NULL)
4680         return;
4681     if (entrada->prox_decomp != NULL)
4682     {
4683         destroi_vetor_decomp_simples(entrada->prox_decomp);
4684     }
4685
```

```
4686     destroi_decomp_simples(entrada);
4687 }
4688
4689 //funcoes para medicao de tempo
4690 void inicia_cronometro(time_t *medidor)
4691 {
4692     *medidor = time(NULL);
4693 }
4694
4695 void para_cronometro(time_t *medidor)
4696 {
4697     time_t tms;
4698     unsigned long int horas, minutos, segundos, dias;
4699
4700     tms = time(NULL);
4701
4702     segundos = (unsigned long int)difftime(tms, *medidor);
4703
4704     minutos = (segundos/60);
4705     segundos =(unsigned long int)(segundos%60);
4706
4707     horas = minutos/60;
4708     minutos = (unsigned long int)(minutos%60);
4709
4710     dias = horas/24;
4711     horas = (unsigned long int)(horas%24);
4712
4713     printf("\ntime elapsed: %lu days, %lu hours, %lu minutes, %lu seconds",dias,horas,minutos,segundos);
4714 }
4715
4716 vetor_decomp *encontra_decomp_mulder_safe(vetor_polinomios *entrada, int grau, lista_expr *expr_simplificada,
         tabela_literais *lista_literais)
4717 {
4718     vetor_polinomios *primario_ptr, *secundario_ptr; //ponteiro para os loops internos
4719
4720     vetor_decomp *decomp_atual; //lista que sera gerada dentro do loop atual
4721     vetor_decomp *lista_decomp = NULL; //elemento manipulado dentro do loop
4722     vetor_decomp_simple *poly_pares = NULL, *poly_impares = NULL;
4723     lista_expr *Q=NULL, *num_a=NULL, *num_b=NULL;
4724
4725     int total_decomp = 0;
4726
4727
4728     //inicializacao da lista de referencia
4729     primario_ptr = entrada;
4730     decomp_atual = NULL;
4731
4732     //loop de construcao das decomposicoes
4733     while (primario_ptr != NULL)
4734     {
4735         //inicializa o proximo ponteiro de busca
4736         secundario_ptr = entrada->proximo_polinomio;
4737         while(secundario_ptr != NULL)
4738         {
4739             //inicializa variaveis
4740             Q = NULL;
4741             num_a = NULL;
4742             num_b = NULL;
4743             //tenta realizar uma expansao em fracao parcialentre primario e secundario
4744             if (partial_fraction_expansion(expr_simplificada, primario_ptr->polinomio->P, secundario_ptr->polinomio->P, &
         Q, &num_a, &num_b))
4745             {
4746                 //disparo a divisao recursiva por 2 BP,s aqui
4747                 encontra_decomp_parcial_primaria(NULL,primario_ptr->polinomio, num_a,secundario_ptr->polinomio, num_b,
         entrada, grau, &lista_decomp, &total_decomp, lista_literais,expr_simplificada );
4748
4749             }
4750             destroi_vetor_decomp_simples(poly_impares);
4751             destroi_vetor_decomp_simples(poly_pares);
4752             poly_pares = NULL;
4753             poly_impares = NULL;
4754
4755
4756             //limpa as variaveis para a proxima passagem
4757             destroi_lista_expr_expandida(Q);
4758             destroi_lista_expr_expandida(num_a);
4759             destroi_lista_expr_expandida(num_b);
4760
4761             //atualizo ponteiro
4762             secundario_ptr = secundario_ptr->proximo_polinomio;
4763         }
```

```
4764
4765        //atualizo ponteiro
4766        primario_ptr = primario_ptr->proximo_polinomio;
4767    }
4768
4769
4770    //rebobinar a lista de decomposicoes
4771    if (lista_decomp != NULL)
4772        while (lista_decomp->ant_decomp != NULL)
4773            lista_decomp = lista_decomp->ant_decomp;
4774
4775    //imprimir numero de decomposicoes
4776    printf("\n\nThe number of valid Translinear decompositions found is: %d", total_decomp);
4777    printf("\n Number of partial fraction operations performed is: %d\n",global_num_parfrac);
4778
4779    //retornar
4780    return lista_decomp;
4781 }
4782 void encontra_decomp_parcial_primaria(vetor_decomp_simple *poly_acumulados, polinomio *base_primario, lista_expr *
     resto_primario, polinomio *base_secundario, lista_expr *resto_secundario, vetor_polinomios *lista_polinomios, int
     grau, vetor_decomp **retorno, int *total_decomp,tabela_literais *lista_literais,lista_expr *eq_entrada)
4783 {
4784    vetor_polinomios *ptr_polinomios = lista_polinomios;
4785    lista_expr *Q, *R, *R_dummy;
4786    vetor_decomp_simple *decomp_atual;
4787    int contador;
4788    vetor_polinomios *poly_ptr;
4789
4790    //sai imediatamente se tiver encontrado uma decomp
4791
4792    //testa a base_primario com todos os polinomios, excluindo-se a propria base_primario
4793    while (ptr_polinomios != NULL)
4794    {
4795        //testa primeiro se o polinomio nao e igual a base_primario
4796        if (base_primario->id != ptr_polinomios->polinomio->id)
4797        {
4798            //limpar as variaveis
4799            Q = NULL;
4800            R = NULL;
4801            R_dummy = NULL;
4802            //testar a base_primario principal com o polinomio atual
4803            if(partial_fraction_expansion(resto_primario, base_primario->P, ptr_polinomios->polinomio->P, &Q,&R, &R_dummy
     ))
4804            {
4805                //limpar o R_dummy e o &Q
4806                destroi_lista_expr_expandida(R_dummy);
4807                destroi_lista_expr_expandida(Q);
4808
4809                //copio o primario, pois pode ser semente para outras decomposicoes
4810                decomp_atual = copia_decomp_simples(poly_acumulados);
4811
4812                //insiro os polinomios que fazem parte da decomposicao
4813                insere_polinomio(&decomp_atual->polinomios,ptr_polinomios->polinomio);
4814
4815                //atualiza resto_primario par e resto_primario impar
4816                decomp_atual->resto = R;
4817
4818                //quando o numero de polinomios acumulados for igual ao grau menos 1 da equacao de entrada pode ser que
     ja tenha terminado
4819                contador = 0;
4820                poly_ptr = decomp_atual->polinomios;
4821                while (poly_ptr != NULL)
4822                {
4823                    ++contador;
4824                    poly_ptr = poly_ptr->proximo_polinomio;
4825                }
4826
4827                //somo um ao contador
4828                contador++;
4829                //vejo se ja terminou
4830                if (contador == grau)
4831                {
4832                    //terminou de fazer uma decomposicao parcial, comecar a procurar decomposicoes parciais no outro
     vetor
4833                    encontra_decomp_parcial_secundaria(NULL,base_secundario, resto_secundario, lista_polinomios, grau,
     retorno, decomp_atual, base_primario, total_decomp,lista_literais,eq_entrada);
4834                    //aqui todas as decomposicoes terao sido inseridas dentro sda rotina da secundaria
4835                    //destroi o vetor simple primario
4836                    destroi_decomp_simples(decomp_atual);
4837
4838                }
```

```
4839                //caso contrario, prosseguir com as decomposicoes parciais de forma recursiva
4840                else
4841                {
4842                    //caso contrario, deve-se proceder com a decomposicao recursivamente
4843                    encontra_decomp_parcial_primaria(decomp_atual, base_primario, R, base_secundario, resto_secundario,
     ptr_polinomios, grau,retorno, total_decomp,lista_literais,eq_entrada);
4844
4845                    //como as decomposicoes validas serao adicionadas no if acima, quando o programa chegar aqui,
     significa
4846                    //que nao preciso mais de decomp_atual;
4847                    destroi_decomp_simples(decomp_atual);
4848                    decomp_atual = NULL;
4849                }
4850
4851            }
4852
4853        }
4854
4855        //incrementa o ponteiro
4856        ptr_polinomios = ptr_polinomios->proximo_polinomio;
4857    }
4858 }
4859
4860 void encontra_decomp_parcial_secundaria(vetor_decomp_simple *poly_acumulados, polinomio *base, lista_expr *resto,
     vetor_polinomios *lista_polinomios, int grau, vetor_decomp **retorno, vetor_decomp_simple *decomp_simples_primario,
     polinomio *base_primario, int *total_decomp,tabela_literais *lista_literais, lista_expr *eq_entrada)
4861 {
4862    vetor_polinomios *ptr_polinomios = lista_polinomios;
4863    lista_expr *Q, *R, *R_dummy;
4864    vetor_decomp_simple *decomp_atual;
4865    int contador;
4866    vetor_polinomios *poly_ptr;
4867
4868    //testa a base com todos os polinomios, excluindo-se a propria base
4869    while (ptr_polinomios != NULL)
4870    {
4871        //testa primeiro se o polinomio nao e igual a base
4872        if (base->id != ptr_polinomios->polinomio->id)
4873        {
4874            //limpar as variaveis
4875            Q = NULL;
4876            R = NULL;
4877            R_dummy = NULL;
4878            //testar a base principal com o polinomio atual
4879            if(partial_fraction_expansion(resto, base->P, ptr_polinomios->polinomio->P, &Q,&R, &R_dummy))
4880            {
4881                //limpar o R_dummy e o &Q
4882                destroi_lista_expr_expandida(R_dummy);
4883                destroi_lista_expr_expandida(Q);
4884
4885                //copio o primario, pois pode ser semente para outras decomposicoes
4886                decomp_atual = copia_decomp_simples(poly_acumulados);
4887
4888                //insiro os polinomios que fazem parte da decomposicao
4889                insere_polinomio(&decomp_atual->polinomios,ptr_polinomios->polinomio);
4890
4891                //atualiza Resto par e Resto impar
4892                decomp_atual->resto = R;
4893
4894                //quando o numero de polinomios acumulados for igual ao grau menos 1 da equacao de entrada pode ser que
     ja tenha terminado
4895                contador = 0;
4896                poly_ptr = decomp_atual->polinomios;
4897                while (poly_ptr != NULL)
4898                {
4899                    ++contador;
4900                    poly_ptr = poly_ptr->proximo_polinomio;
4901                }
4902
4903                //somo um ao contador
4904                contador++;
4905                //vejo se ja terminou
4906                if (contador == grau)
4907                {
4908                    //terminou de fazer uma decomposicao parcial, chamar a compara_Decomp
4909                    combina_decomp_mulder(decomp_atual, decomp_simples_primario, base, base_primario, eq_entrada, retorno
     , total_decomp, lista_literais);
4910                    //*retorno = insere_decomp_simples(*retorno, decomp_atual);
4911                    //apagar os vetores decomp_simple secundario
4912                    destroi_decomp_simples(decomp_atual);
4913                }
```

```
4914                //caso contrario, prosseguir com as decomposicoes parciais de forma recursiva
4915                else
4916                {
4917                    //caso contrario, deve-se proceder com a decomposicao recursivamente
4918                    encontra_decomp_parcial_secundaria(decomp_atual, base, R, ptr_polinomios,grau, retorno,
        decomp_simples_primario,base_primario,total_decomp,lista_literais, eq_entrada);
4919                    //como as decomposicoes validas serao adicionadas no if acima, quando o programa chegar aqui,
        significa
4920                    //que nao preciso mais de decomp_atual;
4921                    destroi_decomp_simples(decomp_atual);
4922                    decomp_atual = NULL;
4923                }
4924            }
4925        }
4926
4927        //incrementa o ponteiro
4928        ptr_polinomios = ptr_polinomios->proximo_polinomio;
4929    }
4930 }
```