Universidade Federal de Goiás - UFG
Regional Catalão - REC
Instituto de Biotecnologia - IBiotec - DCC
Ciência da Computação

Desenvolvimento de um algoritmo para interpolação polinomial da saída de um conversor analógico-digital

Trabalho Final de Arquitetura de Computadores

Autor: Arthur Borges Gonçalves, Luis Vinicius Costa Silva

Professor: Tércio Alberto Santos Filho

Relatório referente ao projeto final da disciplina de Arquitetura de Computadores como requisito parcial para aprovação na dita disciplina

Catalão - GO

11 de Março de 2016

Sumário

1 Resumo

Este trabalho consistiu em ampliar as capacidades de um microcontrolador a fim de que este possa representar de forma mais fidedigna os dados lidos por um ADC(Analogic-Digital Converter) através de um algoritmo baseado no método de Gregory-Newton. Neste projeto foi utilizado um microcontrolador ATMega8 para processamento dos dados, um chip MAX232 para transmissão dos dados entre o ATMega8 e um dispositivo para visualização dos dados e um potenciometro linear conectado ao ADC do ATMega8, este potenciômetro é responsável por alterar a resistência elétrica numa dada parte do circuito. O ADC por sua vez lê a resistência elétrica e transmite um sinal digital que representa a entrada analógica do mesmo.
A partir daí um programa realiza a interpolação dos valores convertidos pelo ADC e retorna um polinômio interpolador que permite que partes intermediárias entre uma leitura e outra do ADC sejam computadas sem serem necessariamente lidas.
Este trabalho se limitou a descrever a lógica do algoritmo de implementação assim como uma descrição básica do circuito elétrico da protoboard e os experimentos conduzidos para avaliação da precisão do algoritmo.

2 Introdução

O universo computacional, diferente do "mundo real"no qual existem infinitos valores contínuos, se restringe à um conjunto finito de valores discretos. É sabido que grandezas do mundo real não podem ser representadas de forma exata em um ambiente digital. Tal lacuna entre mundo real e mundo digital se torna um problema quando problemas envolvendo representações de valores do mundo real em um meio computacional(e.g: a leitura de uma dada amostra de valores por um sensor conectado a um conversor analógico-digital).
Este trabalho tem como objetivo criar um meio de transpor tal lacuna entre mundo real e mundo computacional no que tange a representação de um conjunto de valores lidos por um sensor qualquer conectado a um ADC(Analogic-Digital Converter). A técnica utilizada para este projeto foi a implementação de um algoritmo de interpolação polinomial baseado no método de Gregory-Newton para interpolação de valores equidistantes na abscissa de um plano cartesiano.
Através deste algoritmo, pode-se construir uma função que "encaixe"nos dados pontuais fornecidos pelo ADC, conferindo-lhes, então, a continuidade desejada. O objetivo deste trabalho é obter uma função, à partir de valores obtidos através do conversor AD, e com esta função, ser capaz de determinar os infinitos valores intermediários assumidos pelos dados recebidos antes da conversão efetuada pelo ADC.

3 Materiais e métodos

Neste projeto foi escolhido um potenciômetro linear que incrementa ou decrementa a resistência elétrica na região do circuito.

O ADC do AtMega8 realiza a leitura da resistência resultante a cada pulso de clock e converte para um valor discreto com tamanho de palavra de 8 bits(valores de 0x00 a 0xFF) o valor analógico lido. Cada valor discreto representa um intervalo de $\frac{1}{2^8}$.
Por exemplo, suponha que a resolução do ADC seja de 2 bits, logo os valores de saída discretos do ADC representarão os seguintes intervalos contínuos respectivamente:

00 →[0%,25%)
01 →[25%,50%)
10 →[50%,75%)
11 →[75%,100%]

O algoritmo de interpolação utilizado neste trabalho foi implementado em linguagem C e se baseia no método de Gregory-Newton para interpolação polinomial para valores da abscissa igualmente espaçados. A saída deste algoritmo para n pontos como entrada será um polinômio do tipo:

$$P_n(x) = \Delta^0_{y_0} + \Delta^1_{y_0}(x - x_0) + \Delta^2_{y_0}(x - x_0)(x - x_1) + \cdots + \Delta^n_{y_0}(x - x_0)(x - x_1)\dots(x - x_{n-1})$$

Ou de uma forma mais simplificada:

$$P_n(x) = y_0 + \sum_{i=1}^{n} \Delta^i_{y_0} \prod_{j=0}^{i-1}(x - x_j)$$

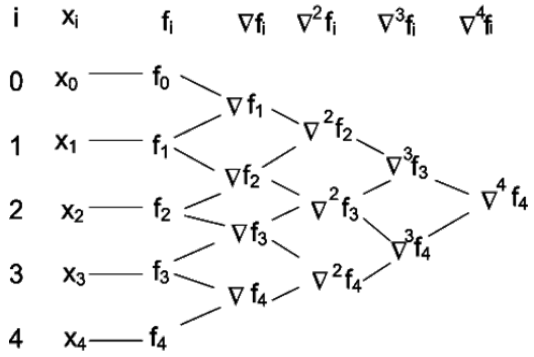
onde:

$$\Delta^k_{y_0} = [x_0,\ x_1,\ x_2,\ \dots,\ x_k]$$

são operadores de ordem dividida de ordem k entre os pontos fornecidos na entrada do algoritmo.
O algoritmo consiste em construir uma tabela com os intervalos regulares e os valores das abscissas e suas respectivas imagens em cada coluna, após isso, uma série de diferenças entre $f(i+1) - f(i)$ constrói uma coluna, este processo de diferenças sucessivas continuará até a coluna possuir valores iguais

ou a diferença entre esses valores for desprezível.
Em nossa implementação, esta diferença desprezível é definida pelo campo "#define margemDeErro" no arquivo que contém a função "main" do programa.
Após o valor ser alcançado após as diferenças sucessivas, este valor "a" é dividido pelo fatorial do número de iterações que o algoritmo levou na etapa anterior até construir a última coluna, este valor é denotado neste trabalho de "k".
No próximo passo um termo do polinômio é criado, o valor obtido denotado "a" será o coeficiente que acompanhará a variável "x" no primeiro termo do polinômio, o valor "k" será o expoente da variável "x".

Abaixo podemos observar uma representação destas diferenças sucessivas:



Um novo conjunto de valores é criado, a fim de que os outros termos do polinômio sejam calculados, estes valores serão a imagem de cada abscissa menos o valor correspondente da ordenada elevado a "k". O algoritmo então é repetido até que se obtenha um termo de grau 0 no polinômio, ou seja, até que a tabela de diferenças sucessivas comece com valores iguais ou com uma diferença desprezível.
Por exemplo, considere a entrada $A = \{(1,2),(2,8),(3,9),(4,11),(5,20)\}$, a execução do algoritmo de interpolação se dará da seguinte maneira:

Por fim é computado que o $f(x)$ que interpola tais pontos é $x^3 - \frac{17}{2!} * x^2 + \frac{49}{2!} * x^1 - 15$

Particularidades na implementação possibilitaram que tal algoritmo fosse executado em um ATMega8, as particularidades tangem em geral o gerenciamento de memória, uma das particularidades foi a alocação de apenas 2 colunas com o número de células já definido para armazenar os valores do processamento, após seu uso, estas eram imediatamente liberadas para liberar espaço para novas alocações. Outra particularidade foi na conversão de valores hexadecimais processados pelo conversor AD para valores de ponto

	0	1	2	3
1	2			
		6		
2	8		-5	
		1		6
3	9		1	
		2		6
4	11		7	
		9		
5	20			

Figura 1: Ao fim desta iteração temos que o primeiro termo do polinômio interpolador corresponte a $\frac{6}{3!} * x^3$

	Original - n^3
1	$2 - 1^3 = 2 - 1 = 1$
2	$8 - 2^3 = 8 - 8 = 0$
3	$9 - 3^3 = 9 - 27 = -18$
4	$11 - 4^3 = 11 - 64 = -53$
5	$20 - 5^3 = 20 - 125 = -105$

Figura 2: Novos valores são calculados para a criação da nova tabela a fim de obter o termo de grau 2

	0	1	2
1	1		
		-1	
2	0		-17
		-18	
3	-18		-17
		-35	
4	-53		-17
		-52	
5	-105		

Figura 3: Ao fim desta iteração temos que o próximo termo do polinômio interpolador corresponte a $-\frac{17}{2!} * x^2$

	Original - -17/2 n^2
1	1 + 17/2 1^2 = 1 + 17/2 = 19/2
2	0 + 17/2 2^2 = 0 + 34 = 34
3	-18 + 17/2 3^2 = -18 + 153/2 = 117/2
4	-53 + 17/2 4^2 = -53 + 272/2 = 83
5	-105 + 17/2 5^2 = -105 + 425/2 = 215/2

Figura 4: Como de costume novos valores são gerados a partir do expoente obtido pela iteração anterior

	0	1
1	19/2	
		49/2
2	34	
		49/2
3	117/2	
		49/2
4	83	
		49/2
5	215/2	

Figura 5: Ao fim desta iteração temos que o próximo termo do polinômio interpolador corresponte a $\frac{49}{2!} * x^1$

	Original - 49/2 n
1	19/2 - 49/2 x 1 = 19/2 - 49/2 = -15
2	34 - 49/2 x 2 = 34 - 49 = -15
3	117/2 - 49/2 x 3 = 117/2 - 147/2 = -15
4	83 - 49/2 x 4 = 83 - 98 = -15
5	215/2 - 49/2 x 5 = 215/2 - 245/2 = -15

Figura 6: Novos valores serão gerados para que a própria iteração crie o termo de grau 0 do polinômio

	0
1	-15
2	-15
3	-15
4	-15
5	-15

Figura 7: Ao fim desta iteração temos que o próximo termo do polinômio interpolador corresponte a -15

flutuante, que seriam manipulados no algoritmo, estes foram limitados a 8 bits através de um mascaramento e limitados entre intervalos de 0.0 a 10.0 com 6 casas decimais de precisão. Estes limites foram definidos experimentalmente para que o algoritmo fosse executado sem comportamentos inesperados, um fator agravante é que o ATMega8 não possui FPU(floating point unit) e operações de ponto flutuante tiveram que ser feitas totalmente via software através de algumas adaptações na forma de manipular valores. Abaixo a implementação da função de conversão de hexadecimal para decimal pode ser observada assim como a declaração do fluxo para conversão do stream de dados para um formato compatível com a comunicação RS232:

```
1 double hex2Per (unsigned char x) {
2 return ((double) ((2*((float) (x & 0xFF)))/51));
3 }
4
5 static FILE mystdout = FDEV_SETUP_STREAM(uart_putchar,
    NULL, _FDEV_SETUP_WRITE);
```

Outro problema foi a gravação do programa no ATMega8, visto que inicialmente o arquivo excedia a região de gravação. Foi usada a opção "-Os" (Optimize for size) para que o código gerado pelo GCC Compiler fosse otimizado. Transformamos também todas as variáveis "flag" para tipos "unsigned char" para que ocupassem menos memória durante a execução e no arquivo gerado. Muitas diretivas de otimização usadas neste projeto foram retiradas da documentação da Atmel disponível em `http://www.atmel.com/images/doc8453.pdf`

Também foi usado links para que a função "printf" da biblioteca "stdio.h" fosse utlizado para impressão através do fluxo de dados da USART.

Um "#define tamanhoAmostragem"nas linhas iniciais do arquivo principal do código define quantos valores serão lidos do ADC e convertidos, por padrão foi definido 5, mas esse valor pode ser escalado até 15 obtendo em alguns casos uma interpolação aceitável da entrada.
Na imagem abaixo pode-se observar o programa sendo executado após uma leitura bem sucedida de 5 valores pelo ADC, exibindo assim o polinômio interpolador e uma plotagem do mesmo, note que o $f(2)$ do polinômio interpolador corresponde com o valor lido pelo ADC.

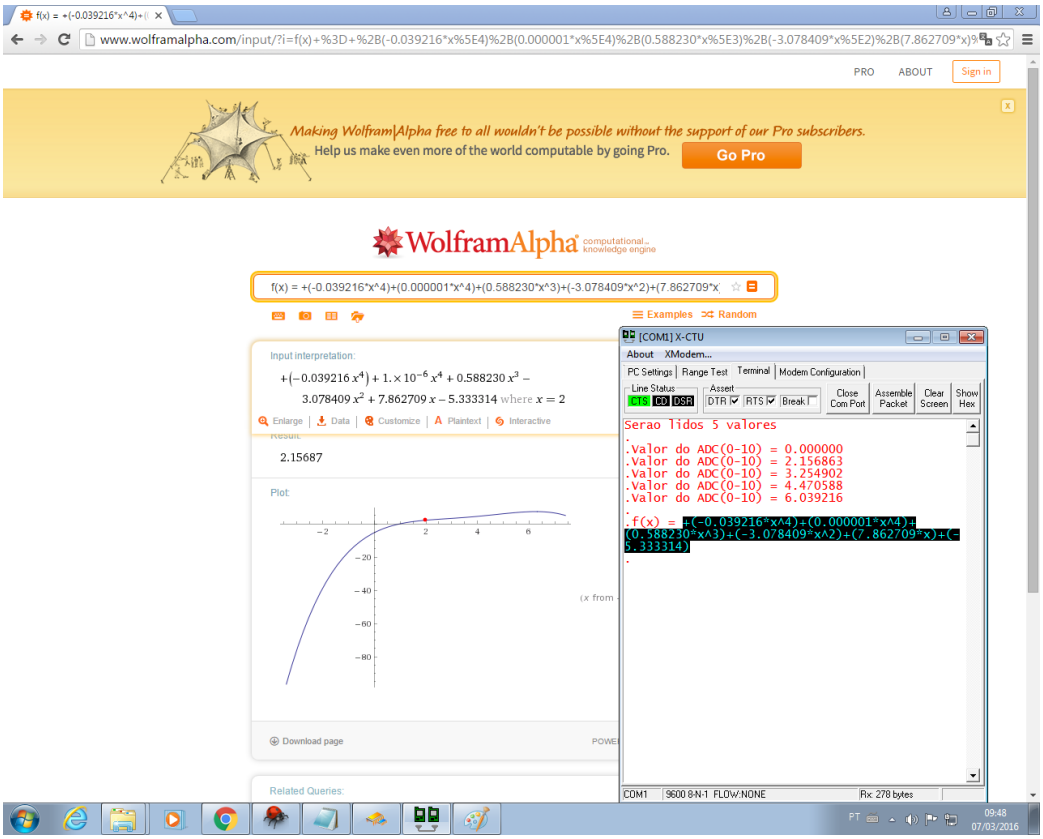


Figura 8: Screenshot do programa sendo executado e uma plotagem do gráfico ao fundo usando o Wolfram Alpha.

4 Experimentos

Um experimento foi montado para observar quão fidedigna é a saída do algoritmo interpolador em relação aos valores contínuos, foi escrito um programa em C no qual ele executa a leitura de 5 valores do ADC a cada 2 segundos, e a partir daí deve interpolar tais valores e obter todos os valores que ocorreram neste intervalo, mas não foram registrados pelo algoritmo.Durante estes 10 segundos o programa lê e grava continuamente valores do ADC na EEPROM a fim de que seja possível comparar a disparidade entre os valores interpolados e os valores lidos em tempo real. A tabela de valores lidos pode ser observada abaixo:

1	3,72549
2	3,686276
3	3,568634
4	3,529428
5	3,568658

Tabela 1: Valores de entrada

O seguinte polinômio foi obtido: $f(x) = (-0.006536 * x^4) + (0.091504 * x^3) + (-0.424838 * x^2) + (0.692812 * x) + (3.372548)$

Quando plotamos um gráfico com 2 curvas representando o polinômio interpolador e os valores lidos em tempo real temos um resultado extremamente satisfatório, visto que as 2 curvas praticamente se sobrepõem. Veja a plotagem das duas curvas abaixo:

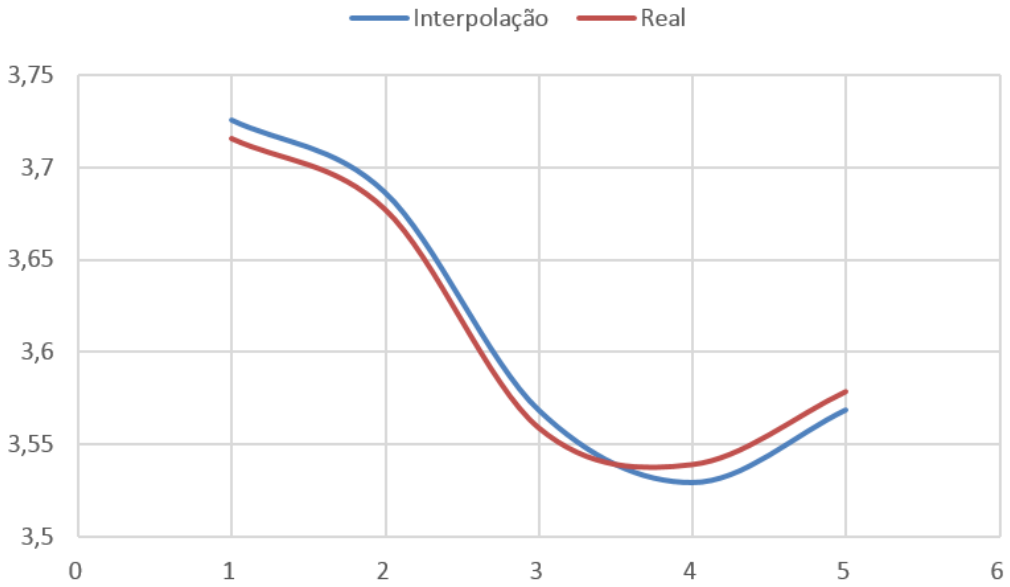


Figura 9: Comparação das curvas geradas pelos dados resultantes da interpolação e da curva representando os dados reais

5 Conclusões

Concluímos que a utilização da interpolação pode ser deveras beneficente para casos onde se necessita de uma apuração mais detalhada dos estados dos itens ou dos valores obtidos como entrada. Neste trabalho foi demonstrado que a partir de contínuas entradas obtidas do potenciômetro, selecionamos alguns valores e a partir destas utilizamos o algoritmo baseado no método de Gregory-Newton para encontrar uma função adequada para interpolação dos valores lidos. Devido a limitações de hardware este trabalho recomenda que a interpolação polinomial ocorra em microcontroladores dedicados a este propósito e que preferencialmente possuam uma FPU, principalmente se a interpolação for executada com muitos termos, visto que teremos coeficientes polinomiais extremamente pequenos e expoentes extremamente grandes.

6 Arquivos-fonte

6.1 main.c

```
#define FOSC 8000000UL// Clock Speed
#define BAUD 9600

#include <avr/io.h>
#include <avr/eeprom.h>
#include <stdio.h>
#include <util/delay.h>
#include "processamento.h"
#define MYUBRR ((FOSC/ (BAUD * (long)16)))
#define  F_CPU 8000000
static int uart_putchar(char c, FILE *stream);
static FILE mystdout = FDEV_SETUP_STREAM(uart_putchar,
   NULL, _FDEV_SETUP_WRITE);
double hex2Per (unsigned char x);
void USART_Init( unsigned intubrr);

#define atrasoEntreLeituras 10000
#define tamanhoAmostragem 5

uint16_t ReadADC(uint8_t ch);
void InitADC();
void interpola (double *entradaOficial);

int main () {

double entradaOficial[tamanhoAmostragem];
InitADC();
USART_Init(MYUBRR);
unsigned char flagImpressao = 0x00;
unsigned char leitorADC = 0x00;
stdout = &mystdout;

while(1)
{
        if(flagImpressao==0x00)//Comeca o processamento
        {
        printf("Serao lidos %d valores\n",
           tamanhoAmostragem);
        _delay_ms(atrasoEntreLeituras);
```

```
38                  for(int i=0;i<tamanhoAmostragem;i++)
39                  {
40                  leitorADC = ReadADC(0);
41                  entradaOficial[i] = hex2Per(leitorADC);
42                  printf("\n T(%d) -> Valor do ADC = %f",i
                        +1,entradaOficial[i]);
43                  _delay_ms(atrasoEntreLeituras);
44                  }
45          interpola(entradaOficial);
46          flagImpressao = 0x01;
47          }
48  }
49  return 0;
50  }
51
52  void interpola (double *entradaOficial) {
53  printf("\n\nf(x) = ");
54  defineEntrada(entradaOficial,tamanhoAmostragem);
55  double coeficiente = 0;
56  int expoente = 0;
57  int expoenteAnterior = expoente;
58  convergencia(entradaOficial,&coeficiente,&expoente);
59
60  if(coeficiente==1)
61  {
62          if(expoente==1)
63          {
64          printf("+(x)");
65          }
66          else
67          {
68          printf(expoente==0 ? "+(x^%d)" : "(x^%d)",
                expoente);
69          }
70  }
71  else if((coeficiente!=0) || (expoente!=0))
72  {
73  printf(expoente==0 ? "+(%f)\n" : expoente==1 ? "+(%f*x)"
        : "+(%f*x^%d)",coeficiente,expoente);
74  }
75  double *saida = (double *) malloc(sizeof(double)*
       tamanhoAmostragem);
76  divergencia(entradaOficial,saida,coeficiente,expoente);
```

```
77  convergencia(saida,&coeficiente,&expoente);
78  if(coeficiente==1)
79  {
80          if(expoente==1)
81          {
82          printf("+(x)");
83          }
84          else
85          {
86          printf(expoente==0 ? "+(x^%d)" : "(x^%d)",
                expoente);
87          }
88  }
89  else if((coeficiente!=0) || (expoente!=0))
90  {
91  printf(expoente==0 ? "+(%f)\n" : expoente==1 ? "+(%f*x)"
        : "+(%f*x^%d)",coeficiente,expoente);
92  }
93  while((expoente!=0) && (expoenteAnterior!=expoente))
94  {
95  expoenteAnterior = expoente;
96  divergencia(saida,saida,coeficiente,expoente);
97  convergencia(saida,&coeficiente,&expoente);
98  if(coeficiente==1)
99  {
100         if(expoente==1)
101         {
102         printf("+(x)");
103         }
104         else
105         {
106         printf(expoente==0 ? "+(x^%d)" : "(x^%d)",
                expoente);
107         }
108 }
109 else if((coeficiente!=0) || (expoente!=0))
110 {
111 printf(expoente==0 ? "+(%f)\n" : expoente==1 ? "+(%f*x)"
        : "+(%f*x^%d)",coeficiente,expoente);
112 }
113 }
114 }
115
```

```
116 int uart_putchar(char c, FILE *stream) {
117 if (c == '\n')
118 uart_putchar('\r', stream);
119 while(!(UCSRA & (1<<UDRE)));
120 UDR = c;
121 return 0;
122 }
123
124 void USART_Init(unsigned int ubrr)
125 {
126 UBRRH=(unsigned char)(ubrr>>8);
127 UBRRL=(unsigned char)ubrr;
128 UCSRB=(1<<RXEN)|(1<<TXEN);
129 UCSRC=(1<<URSEL)|(1<<USBS)|(3<<UCSZ0);
130 }
131
132 double hex2Per (unsigned char x) {
133 return ((double) ((2*((float) (x & 0xFF)))/51));
134 }
135
136 void InitADC()
137 {
138 ADMUX=(1<<REFS0);
139 ADCSRA=(1<<ADEN)|(1<<ADPS2)|(1<<ADPS1)|(1<<ADPS0);
140 }
141
142
143 uint16_t ReadADC(uint8_t ch)
144 {
145 ch=ch&0x07;
146 ADMUX|=ch;
147 ADCSRA|=(1<<ADSC);
148 while(!(ADCSRA & (1<<ADIF)));
149 ADCSRA|=(1<<ADIF);
150 return(ADC);
151 }
```

6.2 processamento.h

```
#define margemDeErro 0.00001
#include <math.h>
#include <stdlib.h>

void divergencia (double *base,double *saida,double
   coeficiente,int expoente);
void convergencia (double *base,double *coeficiente,int
   *expoente);
double fatorial (double n);
unsigned int tamanhoAmostragem = 0;

double *input;

void defineEntrada (double *base,unsigned int tamanho) {
input = base;
tamanhoAmostragem = tamanho;
}

void divergencia (double *base,double *saida,double
   coeficiente,int expoente) {
for(int i=0;i<tamanhoAmostragem;i++)
{
saida[i] = base[i]-(coeficiente*pow(i+1,expoente));
}
}

unsigned char checaIgualdadeDaBase (double *base) {
unsigned char flagIgualdade = 0x00;
for(int i=1;i<tamanhoAmostragem;i++)
{
double x = base[i-1]-base[i];
        if(!((x<=margemDeErro) && (x>=-margemDeErro)))
        {
        flagIgualdade = 0x01;
        break;
        }
}
return flagIgualdade;
}

```

```
38 void convergencia (double *base,double *coeficiente,int
      *expoente) {
39 double **tabelasIteradas;
40 int j = 0;
41 double valorFinal = 0;
42 unsigned char flagIgualdade = checaIgualdadeDaBase(base)
      ;
43
44 if(flagIgualdade==0x00)
45 {
46 *coeficiente = base[0];
47 *expoente = 0;
48 }
49
50 else
51 {
52 flagIgualdade = 0x00;
53 tabelasIteradas = (double **) malloc(sizeof(double)*1);
54 /*if(tabelasIteradas==NULL)
55 {
56 }*/
57 *(tabelasIteradas+0) = (double *) malloc(sizeof(double)
      *(tamanhoAmostragem-1));
58 /*if(*(tabelasIteradas+0)==NULL)
59 {
60 }*/
61
62 *(*(tabelasIteradas+(0))+0) = base[1]-base[0];
63
64 for(int i=1;i<tamanhoAmostragem-1;i++)
65 {
66 *(*(tabelasIteradas+(0))+i) = base[i+1]-base[i];
67         if(*(*(tabelasIteradas+(0))+i)!=*(*(
              tabelasIteradas+(0))+(i-1)))
68         {
69         flagIgualdade = 0x01;
70         }
71 }
72
73 if(flagIgualdade==0x00)
74 {
75 valorFinal = *(*(tabelasIteradas+(0))+0);
76 j = 1;
```

```
77 }
78 else
79 {
80 flagIgualdade = 0x00;
81         for(j=0;;j++)
82         {
83         tabelasIteradas = (double **) realloc(
              tabelasIteradas,sizeof(double *)*2);
84         /*if(tabelasIteradas==NULL)
85         {
86         }*/
87 *(tabelasIteradas+((j+1)%2)) = (double *) malloc(sizeof(
    double)*(tamanhoAmostragem-j-1));
88         /*if(*(tabelasIteradas+((j+1)%2))==NULL)
89         {
90         }*/
91
92         flagIgualdade = 0x00;
93
94         *(*(tabelasIteradas+((j+1)%2))+0) = *(*(
              tabelasIteradas+(j%2))+(0+1)) - *(*(
              tabelasIteradas+(j%2))+(0));
95
96         if((*(*(tabelasIteradas+((j+1)%2))+0)<=
              margemDeErro) && ((*(*(tabelasIteradas+((j+1)
              %2))+0)>=-margemDeErro)))
97         {
98         flagIgualdade = 0x00;
99         valorFinal = *(*(tabelasIteradas+j%2)+0);
100        j++;
101        break;
102        }
103
104        for(int i=1;i<tamanhoAmostragem-j-2;i++)
105        {
106        *(*(tabelasIteradas+((j+1)%2))+i) = *(*(
              tabelasIteradas+(j%2))+(i+1)) - *(*(
              tabelasIteradas+(j%2))+(i));
107                if( (*(*(tabelasIteradas+((j+1)%2))+i)
                      !=*(*(tabelasIteradas+((j+1)%2))+(i
                      -1))) )
108                {
109                flagIgualdade = 0x01;
```

```
110                     }
111             }
112             if(flagIgualdade==0x00)
113             {
114             valorFinal = *(*(tabelasIteradas+((j+1)%2))+0);
115             j += 2;
116             break;
117             }
118 }
119 }
120 free(tabelasIteradas);
121 *coeficiente = valorFinal/fatorial(j);
122 *expoente = j;
123 }
124 }
125
126 double fatorial (double n) { return n<=2 ? (n<=0 ? 1 : n
      ) : n*fatorial(n-1);}
```