Uma análise comparativa e implementação do método de Newton-Raphson para solução de equações de uma variável

Métodos Numéricos

Professora: Prof^a Dr^a Claudia Mazza Dias
Alunos: Alexsander Andrade de Melo e Ygor de Mello Canalli

Universidade Federal Rural do Rio de Janeiro
Instituto Multidisciplinar

Nova Iguaçu - RJ, 19 de março de 2013

Resumo

Neste trabalho, apresentaremos um breve resumo teórico alguns dos principais métodos numéricos para solução de equações de uma variável. O Método principal deste estudo éo método de Newton-Raphson. Além de explicarmos o funcionameto de cada método, apresentaremos uma implementação em C e um estudo comparativo dos demais métodos com Newton-Raphson.

Sumário

1 Soluções de equações de uma variável

Os métodos numéricos aqui discutidos são utilizados para aproximar solução de equações que não são possíveis de ser solucionadas com valor exato através de método algébricos. Em especial, nos deteremos ao problema de encontrar a raiz de uma determinada função, isto é, encontrar x tal que $f(x) = 0$, denominado *zero* da função.

1.1 Método da bissecção

O primeiro método se baseia no Teorema do Valor Intermediário, a saber,

Teorema 1.1 (Teorema do Valor Intermediário)**.** *Se $f \in C[a, b]$ e k for qualquer número entre $f(a)$ e $f(b)$ (isto é, $f(a) \leq k \leq f(b)$), então existe $c \in (a, b)$ para o qual*

$$f(c) = K.$$

O método divide iterativamente o intervalo em subintervalor $[a, b]$ e a cada passo localizando a metado do intervalor p. Encontramos o ponto médio p_1 de $[a, b]$, dado por

$$p_1 = a_1 + \frac{b_1 - a_1}{2} = \frac{a_1 + b_1}{2}.$$

Se $f(p_1) = 0$, temos a solução exata de nosso problema. Caso $\frac{a_1+b_1}{2}$ seja menor que a tolerância, significa que temos uma solução p_1 dentro desta tolerância de erro.

Segue abaixo o método da bissecção:

Algoritmo 1.1 Método da bissecção

Entrada: Função f; Extremidades a, b; Tolerância $TOL \in \mathbb{R}$; Número máximo de iterações $N \in \mathbb{Z}$.

Saída: Solução aproximada p ou erro, onde $f(p) = 0$.

função BISSECÇÃO(f, a, b, TOL, N)
    $i = 1$
    $FA = f(a)$
    enquanto $i \leq N$ **faça**
        $p = a + (b - a)/2$
        $FP = f(p)$
        se $FP = 0$ ou $(b - a)/2 < TOL$ **então**
            retorne p
        fim se
        se $FA \cdot FP > 0$ **então**
            $a = p$
            $FA = FP$
        senão
            $b = p$
        fim se
        $i = i + 1$
    fim enquanto
    retorne ERRO
fim função

1.2 Método de Newton-Raphson

O método de Newton-Raphson é um dos métodos numéricos mais eficientes para o cálculo de raízes de uma equação. Seu funcionamento se baseia nos polinomios de Taylor, e é explicado abaixo.

Teorema 1.2. *Seja $f : \mathbb{R} \to \mathbb{R}$ com derivadas contínuas até ordem n:*

$$f(x+h) = f(x) + hf'(x) + \frac{h^2}{2!}f'(x) + \cdots + \frac{h^n}{n!}f^n(x) + R_n,$$

onde $h = x_n - x_{n-1}$ e

$$R_n = \frac{h^{n+1}}{(n+1)!}f^{n-1}(\xi)$$

para algum $\xi \in [x, x+h]$.

Logo, se $x_n = x_{n-1} + \alpha$ e $f(x_{n-1} + \alpha) = 0$, então

$$\begin{aligned} 0 &= f(x_{n-1} + \alpha) = f(x_{n-1}) + \alpha f'(x_{n-1}) + \frac{\alpha^2}{2}f''(x_{n-1}) + \cdots \\ 0 &= f(x_{n-1} + \alpha) \approx f(x_{n-1}) + \alpha f'(x_{n-1}). \end{aligned}$$

Portanto,

$$\alpha \approx -\frac{f(x_{n-1})}{f'(x_{n-1})} \implies x_n \approx x_{n-1} - \frac{f(x_{n-1})}{f'(x_{n-1})}. \tag{1.1}$$

Teorema 1.3. *Seja $f \in C^2[a,b]$. Se $p \in [a,b]$ é tal que $f(p) = 0$ e $f'(p) \neq 0$, então, existe um $\delta > 0$ de forma que o método de Newton-Raphson gera uma sequência $\{p_n\}_{n=0}^{\infty}$ que converge para p para qualquer aproximação inicial $p_0 \in [p - \delta, p + \delta]$.*

Algoritmo 1.2 Método de Newton-Raphson

Entrada: Função f; Aproximação inicial p_0; Tolerância $TOL \in \mathbb{R}$; Número máximo de iterações $N \in \mathbb{Z}$.

Saída: Solução aproximada p ou erro, onde $f(p) = 0$.

```
função NEWTON-RAPHSON(f, p0, TOL, N)
    i = 1
    enquanto i ≤ N faça
        p = p0 − f(p0)/f′(p0)
        se |p − p0| < TOL então
            retorne p
        fim se
        p = p0
        i = i + 1
    fim enquanto
    retorne ERRO
fim função
```

1.3 Método da secante

Apesar do método de Newton-Raphson ser extremamente eficiente e possuir uma ótima convergência, nem sempre é possível calcular a derida da função que se deseja encontrar o zero. Para isso, apresentamos o método da secante, que é uma alternativa para contornar tal problema.

Temos que

$$f'(x_{n-1}) = \lim_{x \to x_{n-1}} \frac{f(x) - f(x_{n-1})}{x - x_{n-1}}.$$

Tomando $x = x_{n-2}$, temos que

$$\begin{aligned} f'(x_{n-1}) &\approx \frac{f(x_{n-2}) - f'(x_{n-1})}{x_{n-1} - x_{n-2}} \\ &= \frac{f(x_{n-2}) - f'(x_{n-1})}{x_{n-1} - x_{n-2}}. \end{aligned}$$

Substituindo f' na fórmula de Newton (1.1), obtemos

$$x_n = x_{n-1} - \frac{f(x_{n-1} - x_{n-2})}{f(x_{n-1}) - f(x_{n-2})}. \tag{1.2}$$

Algoritmo 1.3 Método da secante

Entrada: Função f; Aproximações inicial p_0 e p_1; Tolerância $TOL \in \mathbb{R}$; Número máximo de iterações $N \in \mathbb{Z}$.

Saída: Solução aproximada p ou erro, onde $f(p) = 0$.

função SECANTE(f, p_0, p_1, TOL, N)
  $i = 2$
  $q_0 = f(p_0)$
  $q_1 = f(p_1)$
  enquanto $i \leq N$ **faça**
    $p = p_1 - q_1(p_1 - p_0)/(q_1 - q_0)$
    se $|p - p_1| < TOL$ **então**
      retorne p
    fim se
    $p_0 = p_1$
    $q_0 = q_1$
    $p_1 = p$
    $p_0 = p_1$
    $q_i = f(p)$
    $i = i + 1$
  fim enquanto
  retorne ERRO
fim função

1.4 Método da Falsa Posição

O método da falsa funciona da mesma maneira que o método da secante. A única modificação é que ele inclui um teste que garante que a aproximação gerada não extrapolará o intervalo definido em cada iteração. O método da falsa posição geralmente não é indicado para fins práticos, pois sua maior segurança faz com que sejam necessários mais iterações que no método da secante. Entretanto, o apresentamos com fins teóricos, para ilustrar como podemos delimitar o intervalo no qual a solução aproximada será gerada.

Algoritmo 1.4 Método da falsa posição

Entrada: Função f; Aproximações inicial p_0 e p_1; Tolerância $TOL \in \mathbb{R}$; Número máximo de iterações $N \in \mathbb{Z}$.

Saída: Solução aproximada p ou erro, onde $f(p) = 0$.

```
função FALSA-POSIÇÃO(f, p0, p1, TOL, N)
    i = 2
    q0 = f(p0)
    q1 = f(p1)
    enquanto i ≤ N faça
        p = p1 − q1(p1 − p0)/(q1 − q0)
        se |p − p1| < TOL então
            retorne p
        fim se
        q = f(p)
        se q · q1 < 0 então
            p0 = p1
            q0 = q1
        fim se
        p1 = p
        q1 = q
        i = i + 1
    fim enquanto
    retorne ERRO
fim função
```

2 Análise Comparativa

Dos métodos apresentados acima, um dos mais utilizados por, de forma geral, possuir uma melhor convergência, como talvez já esperado, é o método de Newton-Rapson. No entanto, como vimos, nem sempre sua utilização é viável devido ao agravante da dificuldade de determinar a derivada de uma função, e quando isso ocorre, é preferível a utilização dos demais método. Dessa forma, não existe um método método mais apropriado para o contexto geral, mas sim um método que é preferível de ser utilizado para um determinado problema, cabendo, então, uma análise detalhada do problema antes de se aplicar um método. Além disso, em muitos momentos é feito a utlização de mais de um método sucessivamente, por exemplo, é de comum ocorrência a utlização do método da Bisseção para uma aproximaçao inicial e após isso a utilização do método de Newton-Rapson usando os valores obtidos do método da Bisseção.

Exemplo 2.1. *Considere:*

1. $f(x) = 2x^3 + x^2 - 2$

2. *Para* $f(x) = 0, x \simeq 0.858094$

3. $TOL = 10^{-3}$

4. $N = 30$

5. *Bissecção:* $[-5, 5]$

6. *Secante/Falsa Posição:* $p_0 = -5; p_1 = 5$

7. *Newton-Raphson:* $p_0 = 5$

Iteraçoes	Newton-Raphson	Bissecção	Secante	Falsa Posição
1	3.293750	0.000000	-0.460000	-0.460000
2	2.173285	2.500000	-0.420625	-0.420625
3	1.461878	1.250000	6.537428	-0.381752
4	1.056359	0.625000	-0.397814	-0.343281
5	0.889074	0.937500	-0.375127	-0.305134
6	0.859017	0.781250	15.538132	-0.267246
7	0.858095	0.859375	-0.371089	-0.229571
8		0.820312	-0.367054	-0.192076
9		0.839844	24.449435	-0.154741
10		0.849609	-0.365420	-0.117560
11		0.854492	-0.363786	-0.080536
12		0.856934	28.346145	-0.043687
13		0.858154	-0.362569	-0.007037
14		0.857544	-0.361353	0.029377
15			31.224187	0.065510
16			-0.360350	0.101311
17			-0.359348	0.136720
18			33.940445	0.171671
19			-0.358499	0.206096
20			-0.357650	0.239922
21				0.273077
22				0.305486
23				0.337079
24				0.367787
25				0.397547
26				0.426301
27				0.453998
28				0.480597
29				0.506063
30				0.506063
31				ERRO!

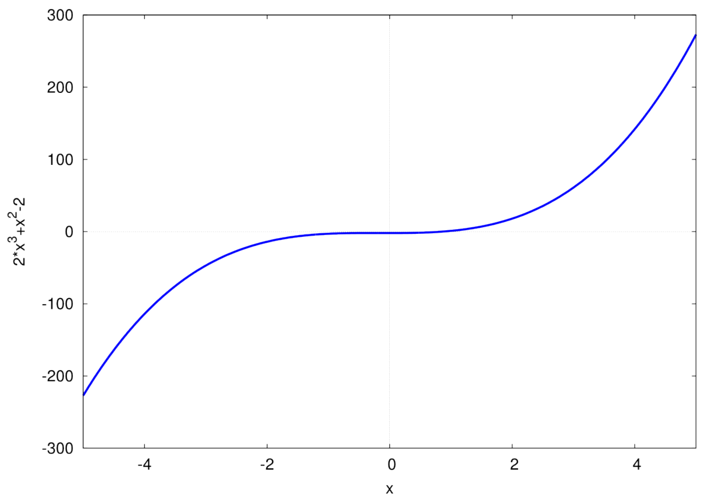

Figura 1: $f(x) = 2x^3 + x^2 - 2$

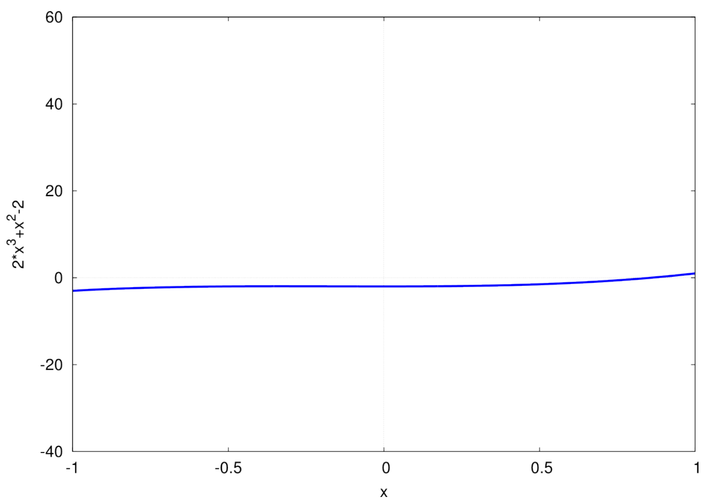

Figura 2: $f(x) = 2x^3 + x^2 - 2$

Como podemos notar pelos gráficos expostos acima e pela Tabela 2.1, que embora o método da secante tenha convergido para uma solução, o resultado retornando por este está incorreto. Isso ocorre, pois o Teorema 1.3 garante apenas que, sob as hipóteses aceitáveis, o método de Newton-Raphson (aqui, equivalentemente, estendido para o método da secante) convergirá se a aproximação inicial escolhida for suficientemente precisa.

Por outro lado, através ainda de uma análise da Tabela 2.1, percebemos nitidamente que, de modo geral, o método de Newton-Raphson converge muito mais rápido do que os demais métodos apresentados. Isso deve-se ao fato de que o método de Newton-Raphson é, na maioria dos casos, *quadraticamente convergente*, enquanto os outros não o são. Os conceitos de *quadraticamente convergente* e *linearmente convergente* são definidos abaixo.

Definição 2.1. Seja $\{p_n\}_{n=0}^{\infty}$ uma sequência que convirja para p, com $p_n \neq p$ para todo n. Se existem cosntantes positivas λ e α com

$$\lim_{n\to\infty} \frac{|p_{n+1} - p|}{|p_n - p|^{\alpha}} = \lambda,$$

então $\{p_n\}_{n=0}^{\infty}$ *converge para* p *com ordem* α.

Definição 2.2. Uma técnica iterativa da forma $p_n = g(p_{n-1})$ é dita da *ordem* α se a sequência $\{p_n\}_{n=0}^{\infty}$ convergir para a solução $p = g(p)$ com ordem α.

De modo geral, uma sequência com alta ordem de convergência converge mais rápido do que uma com menor ordem de convergência (ou seja, se $\alpha_1 > \alpha_2$, onde α_1 é a ordem de convergência uma determinada sequência $\{p_n\}_{n=0}^{\infty}$ e α_2 é a odem de convergência de uma outra sequência $\{q_n\}_{n=0}^{\infty}$, então espera-se que $\{p_n\}_{n=0}^{\infty}$ convirja mais rapidamente do que $\{q_n\}_{n=0}^{\infty}$). Temos dois casos especiais de ordem de convergência, os quais são expostos abaixo:

- Se $\alpha = 1$, dizemos que a sequência é *linearmente convergente.*
- Se $\alpha = 2$, dizemos que a sequência é *quadráticamente convergente.*

Já o método da secante possui *convergência supralinear*, isto é,

$$1 < \alpha < 2.$$

Mais precisamente, neste caso,

$$\alpha = \frac{1}{2}\sqrt{1 + \sqrt{5}} \approx 1.6818.$$

3 Implementações

Por fim, apresentaremos nesta seção uma implementação dos algoritmos expostos acima. Optamos utilizar a *linguagem C* para este feito, devido a mesma ser uma linguagem de mais baixo nível (se comparado com as demais linguagens usuais como *Java*, *Python*, etc.), o que interfere diretamente, de forma positiva, no desempenho da execução do programa.

Tais implementações foram feitas seguindo a risca (com pequenas "modificações" apenas por conta da linguagem utilizada e da apresentação gŕafica dos resultados) os algoritmos expostos ao longo deste trabalho. Cada implementação possui os respectivos métodos para determinar $f(x) = 0$, dada a função $f(x)$. Além desta determinação, a cada passo da execução dos métodos é gerado um gráfico da função $f(x)$ e o respectivo ponto $(x, f(x))$ para o qual $f(x) = 0$ (caso este seja determinado pelo método executado).

Ademais, para cada método foram feitas duas implementações, uma para quaisquer $f(x)$ que são funções polinomiais, onde o usuário entra via teclado com o valor de $f(x)$ e outra implementação para quaisquer funções $f(x)$ aceita pela biblioteca do C *math.h*. No entanto, se nesta última implementação $f(x)$ for alterado, então esta implementação deve ser sempre recompilada com a nova função $f(x)$ e, no caso do Newton-Raphson, também com a função $f'(x)$ correspondente. Segue abaixo cada uma dessas implementações.

Implementação 3.1: newtonRaphson.c

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#include "gnuplot_i.h"

#define MAX_STRING 80
#define SLEEP_LGTH 2


double polyValue(double* poly, int order, double value);
double* polyDerivative(double* poly, int order);
double newtonRaphson(double* poly, int polyOrder, double p0, double TOL, int N,
      int plot);
double doubleAbs(double a);
char* linePlotStringPoly(double x0, double y0, double x1, double y1);
double charToDouble(char* doubleInChar);
double* stringToDoublePoly(char* poly, int order);
char* doubleToStringPoly(double* poly, int order);

int main(int argc, char *argv[])
{

        double zero;
        double *poly;
        double tol;
        double p0;
        int order;
        int n;
        int plot;

        order = (int) strtoul(argv[2], 0, 10);
        p0 = charToDouble(argv[3]);
        tol = charToDouble(argv[4]);
        n = (int) strtoul(argv[5], 0, 10);
        plot = (int) strtoul(argv[6], 0, 10);

        poly = stringToDoublePoly(argv[1], order);

        zero = newtonRaphson(poly, order, p0, tol, n, plot);
        printf("\n\nf(x) = 0; x = %f\n", zero);


        return 0;
}

double doubleAbs(double a)
{
        return (a < 0) ? (-1*a) : a;
}

double polyValue(double* poly, int order, double value)
{
        int i;
        double sum = 0;
        for (i = 0; i < order+1; i++)
                sum += poly[i]*pow(value,(double) i);
        return sum;
}

double* polyDerivative(double* poly, int order)
```

```
{
	int i;
	double* derivative = malloc(order*sizeof(double));
	for (i = 0; i < order; i++)
		derivative[i] = (i+1)*poly[i+1];
	derivative[order] = 0;
	return derivative;
}

double newtonRaphson(double* poly, int polyOrder, double p0, double TOL, int N, int plot)
{
	double* derivative = polyDerivative(poly, polyOrder);
	int i = 0;
	double p = 0;
	double f = 0;
	double fLinha = 0;
	char* functLine;
	gnuplot_ctrl *h1;


	if (plot)
	{
	    h1 = gnuplot_init();
		gnuplot_plot_slope(h1, 1.0, 0.0, "f(x)") ;
		gnuplot_plot_slope(h1, 2.0, 0.0, "f'(x)") ;

		gnuplot_setstyle(h1, "lines");
		gnuplot_cmd(h1, "set style line 1 lt 2 lw 15");

		gnuplot_resetplot(h1);
	gnuplot_plot_equation(h1, doubleToStringPoly(poly, polyOrder), "f(x)");
	    //sleep(SLEEP_LGTH);
	}

	while (i < N)
	{

		f = polyValue(poly, polyOrder, p0);
		fLinha = polyValue(derivative, polyOrder, p0);
		p = p0 - (f/fLinha);

		if (plot)
		{
			functLine = linePlotStringPoly(p0, f, p, 0.00);
			gnuplot_setstyle(h1, "lines");
			gnuplot_plot_equation(h1, functLine, "");
			sleep(SLEEP_LGTH);
		}

		if (doubleAbs(p - p0) < TOL)
		{
			if (plot)
			{
				gnuplot_resetplot(h1);
				gnuplot_cmd(h1, "set label '(%f,0)' at %f,%f point pointtype 2",p, p, 0.00);
			gnuplot_plot_equation(h1, doubleToStringPoly(poly, polyOrder), "f(x)");
				getchar();
				gnuplot_close(h1);
			}
```

```
                printf("\nSolucao encontrada em %d iteracoes", i+1);
                return p;
            }

            if (plot)
                printf("\n%f", p);

            p0 = p;
            i++;
        }
        printf("\nLimite de %d iterecoes atingido. Nao foi possivel determinar o zero da funcao!\n", N);

        getchar();
        if (plot)
            gnuplot_close(h1);

        return p;
}

//return f(x) = ax + b
char* linePlotStringPoly(double x0, double y0, double x1, double y1)
{
            char* line;
            double* functDouble = (double*) malloc (sizeof(double) * 2);

            functDouble[1] = (y1 - y0)/ (x1 - x0); //a
            functDouble[0] = y1 - (functDouble[1]*x1); //b

            line = doubleToStringPoly(functDouble, 2);

            return line;
}


double* stringToDoublePoly(char* poly, int order)
{
        double* polyInDouble;

        int i;
        int j;
        int beginVector;

        polyInDouble = (double*) malloc (sizeof(double) * (order+1));

        i = beginVector = j = 0;

        while (poly[i] != '\0')
        {
                if (poly[i] == ',')
                {
                        poly[i] = '\0';

                        polyInDouble[j] = charToDouble(poly + beginVector);
                        i++; //ignora espaco vazio

                        beginVector = i + 1;
                        j++; //quantidade de virgulas
                }

                i++;
        }
```

```
        polyInDouble[j] = charToDouble(poly + beginVector);

        return polyInDouble;
}

char* doubleToStringPoly(double* poly, int order)
{
        int i;
        char aux[MAX_STRING];
        char* result;

        result = (char*) malloc (sizeof(char) * MAX_STRING * (order + 1));

        for(i = 0; i < order; i++)
        {
                sprintf(aux, "%f *(x**%d) + " , poly[i], i);
                strcat(result, aux);
        }

        sprintf(aux, "%f *(x**%d)" , poly[i], i);
        strcat(result, aux);

        return result;
}

double charToDouble(char* doubleInChar)
{
        int i = 0;
        int j = 1;
        int signal = 1;
        double integer = 0;
        double decimal = 0;

        if (doubleInChar == NULL)
                return 0;


        if (doubleInChar[0] == '-') //Se o numero for negativo
        {
                signal = -1;
                i = 1;
        }

        //Define a parte inteira
        while ((doubleInChar[i] != '\0') && (doubleInChar[i] != ',') && (doubleInChar[i] !=
            '.'))
        {
                integer = (integer * 10) + (doubleInChar[i] - 48);
                i++;
        }

                if (doubleInChar[i] != '\0')
                i++; //pula a virgula

        //Define a parte decimal
        while (doubleInChar[i] != '\0')
        {
                decimal += (doubleInChar[i] - 48) / pow(10, j);
                i++;
                j++;
        }
```

```c

    return signal*(integer + decimal);
}
```

Implementação 3.2: WithDefines/newtonRaphson.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#include "gnuplot_i.h"
#include "variables.h"

#define MAX_STRING 80
#define SLEEP_LGTH 2

double newtonRaphson(double p0, double TOL, int N, int plot);
double doubleAbs(double a);
char* linePlotStringPoly(double x0, double y0, double x1, double y1);
double charToDouble(char* doubleInChar);
char* doubleToStringPoly(double* poly, int order);

int main(int argc, char *argv[])
{

    double zero;
    double tol;
    double p0;
    int n;
    int plot;

    p0 = charToDouble(argv[1]);
    tol = charToDouble(argv[2]);
    n = (int) strtoul(argv[3], 0, 10);
    plot = (int) strtoul(argv[4], 0, 10);

    zero = newtonRaphson(p0, tol, n, plot);
    printf("\n\nf(x) = 0; x = %f\n", zero);


    return 0;
}

double doubleAbs(double a)
{
    return (a < 0) ? (-1*a) : a;
}

double newtonRaphson(double p0, double TOL, int N, int plot)
{
    int i = 0;
    double p = 0;
    double f = 0;
    double fLinha = 0;
    char* functLine;
    gnuplot_ctrl *h1;


    if (plot)
    {
      h1 = gnuplot_init();
        gnuplot_plot_slope(h1, 1.0, 0.0, "f(x)") ;
        gnuplot_plot_slope(h1, 2.0, 0.0, "f'(x)") ;
```

```

        gnuplot_setstyle(h1, "lines");
        gnuplot_cmd(h1, "set style line 1 lt 2 lw 15");

        gnuplot_resetplot(h1);
    gnuplot_plot_equation(h1, SFUNCTION(x), "f(x)");
      //sleep(SLEEP_LGTH);
    }

    while (i < N)
    {

        f = FUNCTION(p0);
        fLinha = DIFFUNCTION(p0);
        p = p0 - (f/fLinha);


        if (plot)
        {
            functLine = linePlotStringPoly(p0, f, p, 0.00);
            gnuplot_setstyle(h1, "lines");
            gnuplot_plot_equation(h1, functLine, "");
            sleep(SLEEP_LGTH);
        }

        if (doubleAbs(p - p0) < TOL)
        {
            if (plot)
            {
                gnuplot_resetplot(h1);
                gnuplot_cmd(h1, "set label '(%f,0)' at %f,%f point pointtype 2",p, p, 0.00);
            gnuplot_plot_equation(h1, SFUNCTION(x), "f(x)");
                getchar();
                gnuplot_close(h1);
            }
            printf("\nSolucao encontrada em %d iteracoes", i+1);
            return p;
        }

        printf("\n%f", p);

        p0 = p;
        i++;
    }
    printf("\nLimite de %d iterecoes atingido. Nao foi possivel determinar o zero da funcao!\n", N);

    getchar();
    if (plot)
        gnuplot_close(h1);

    return p;
}

//return f(x) = ax + b
char* linePlotStringPoly(double x0, double y0, double x1, double y1)
{
        char* line;
        double* functDouble = (double*) malloc (sizeof(double) * 2);

        functDouble[1] = (y1 - y0)/ (x1 - x0); //a
```

```
		functDouble[0] = y1 - (functDouble[1]*x1); //b

		line = doubleToStringPoly(functDouble, 2);

		return line;
}


char* doubleToStringPoly(double* poly, int order)
{
	int i;
	char aux[MAX_STRING];
	char* result;

	result = (char*) malloc (sizeof(char) * MAX_STRING * (order + 1));

	for(i = 0; i < order; i++)
	{
		sprintf(aux, "%f *(x**%d) + " , poly[i], i);
		strcat(result, aux);
	}

	sprintf(aux, "%f *(x**%d)" , poly[i], i);
	strcat(result, aux);

	return result;
}

double charToDouble(char* doubleInChar)
{
	int i = 0;
	int j = 1;
	int signal = 1;
	double integer = 0;
	double decimal = 0;

	if (doubleInChar == NULL)
		return 0;


	if (doubleInChar[0] == '-') //Se o numero for negativo
	{
		signal = -1;
		i = 1;
	}

	//Define a parte inteira
	while ((doubleInChar[i] != '\0') && (doubleInChar[i] != ',') && (doubleInChar[i] !=
		'.'))
	{
		integer = (integer * 10) + (doubleInChar[i] - 48);
		i++;
	}

		if (doubleInChar[i] != '\0')
		i++; //pula a virgula

	//Define a parte decimal
	while (doubleInChar[i] != '\0')
	{
		decimal += (doubleInChar[i] - 48) / pow(10, j);
		i++;
```

```
            j++;
        }

        return signal*(integer + decimal);
}
```

Implementação 3.3: bissecao.c

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#include "gnuplot_i.h"

#define MAX_STRING 80
#define SLEEP_LGTH 2


double polyValue(double* poly, int order, double value);
double* polyDerivative(double* poly, int order);
double bissecao(double* poly, int polyOrder, double a, double b, double TOL, int N,
    int plot);
double doubleAbs(double a);
char* linePlotStringPoly(double x0, double y0, double x1, double y1);
double charToDouble(char* doubleInChar);
double* stringToDoublePoly(char* poly, int order);
char* doubleToStringPoly(double* poly, int order);

int main(int argc, char *argv[])
{

        double zero;
        double *poly;
        double tol;
        double a,b;
        int order;
        int n;
        int plot;

        order = (int) strtoul(argv[2], 0, 10);
        a = charToDouble(argv[3]);
        b = charToDouble(argv[4]);
        tol = charToDouble(argv[5]);
        n = (int) strtoul(argv[6], 0, 10);
        plot = (int) strtoul(argv[7], 0, 10);

        poly = stringToDoublePoly(argv[1], order);

        zero = bissecao(poly, order, a, b, tol, n, plot);
        printf("\n\nf(x) = 0; x = %f\n", zero);


        return 0;
}

double doubleAbs(double a)
{
        return (a < 0) ? (-1*a) : a;
}

double polyValue(double* poly, int order, double value)
{
        int i;
```

```
    double sum = 0;
    for (i = 0; i < order+1; i++)
        sum += poly[i]*pow(value,(double) i);
    return sum;
}

double* polyDerivative(double* poly, int order)
{
    int i;
    double* derivative = malloc(order*sizeof(double));
    for (i = 0; i < order; i++)
        derivative[i] = (i+1)*poly[i+1];
    derivative[order] = 0;
    return derivative;
}

double bissecao(double* poly, int polyOrder, double a, double b, double TOL, int N, int plot)
{
    int i = 0;
    double p = 0;
    double FA = 0;
    double FP = 0;
    gnuplot_ctrl *h1;

    if (plot)
    {
        h1 = gnuplot_init();
        gnuplot_plot_slope(h1, 1.0, 0.0, "f(x)") ;

        gnuplot_setstyle(h1, "lines");
        gnuplot_cmd(h1, "set style line 1 lt 2 lw 15");

    }

    FA = polyValue(poly, polyOrder, a);

    while (i < N)
    {
        p = a + (b-a)/2;
        FP = polyValue(poly, polyOrder, p);

        if ((FP == 0) || ((b-a)/2 < TOL))
        {
            if (plot)
            {
                gnuplot_resetplot(h1);
                gnuplot_cmd(h1, "set label '(%f,0)' at %f,%f point pointtype 2",p, p, 0.00);
                gnuplot_plot_equation(h1, doubleToStringPoly(poly, polyOrder), "f(x)");
                getchar();
                gnuplot_close(h1);
            }
            printf("\nSolucao encontrada em %d iteracoes", i+1);
            return p;
        }

        if (plot)
            printf("\n%f", p);

        if (FA*FP > 0)
```

```
            {
                a = p;
                FA = FP;
            }
            else
                b = p;
            i++;
        }
        printf("\nLimite de %d iterecoes atingido. Nao foi possivel determinar o zero da
            funcao!\n", N);

        getchar();
        if (plot)
            gnuplot_close(h1);

        return p;
}

//return f(x) = ax + b
char* linePlotStringPoly(double x0, double y0, double x1, double y1)
{
            char* line;
            double* functDouble = (double*) malloc (sizeof(double) * 2);

            functDouble[1] = (y1 - y0)/ (x1 - x0); //a
            functDouble[0] = y1 - (functDouble[1]*x1); //b

            line = doubleToStringPoly(functDouble, 2);

            return line;
}


double* stringToDoublePoly(char* poly, int order)
{
        double* polyInDouble;

        int i;
        int j;
        int beginVector;

        polyInDouble = (double*) malloc (sizeof(double) * (order+1));

        i = beginVector = j = 0;

        while (poly[i] != '\0')
        {
            if (poly[i] == ',')
            {
                poly[i] = '\0';

                polyInDouble[j] = charToDouble(poly + beginVector);
                i++; //ignora espaco vazio

                beginVector = i + 1;
                j++; //quantidade de virgulas
            }

            i++;
        }

        polyInDouble[j] = charToDouble(poly + beginVector);
```

```

	return polyInDouble;
}

char* doubleToStringPoly(double* poly, int order)
{
	int i;
	char aux[MAX_STRING];
	char* result;

	result = (char*) malloc (sizeof(char) * MAX_STRING * (order + 1));

	for(i = 0; i < order; i++)
	{
		sprintf(aux, "%f *(x**%d) + ", poly[i], i);
		strcat(result, aux);
	}

	sprintf(aux, "%f *(x**%d)", poly[i], i);
	strcat(result, aux);

	return result;
}

double charToDouble(char* doubleInChar)
{
	int i = 0;
	int j = 1;
	int signal = 1;
	double integer = 0;
	double decimal = 0;

	if (doubleInChar == NULL)
		return 0;


	if (doubleInChar[0] == '-') //Se o numero for negativo
	{
		signal = -1;
		i = 1;
	}

	//Define a parte inteira
	while ((doubleInChar[i] != '\0') && (doubleInChar[i] != ',') && (doubleInChar[i] != '.'))
	{
		integer = (integer * 10) + (doubleInChar[i] - 48);
		i++;
	}

		if (doubleInChar[i] != '\0')
		i++; //pula a virgula

	//Define a parte decimal
	while (doubleInChar[i] != '\0')
	{
		decimal += (doubleInChar[i] - 48) / pow(10, j);
		i++;
		j++;
	}

	return signal*(integer + decimal);
```

```
}
```

Implementação 3.4: WithDefines/bissecao.c

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#include "gnuplot_i.h"

#define MAX_STRING 80
#define SLEEP_LGTH 2

#include "variables.h"

double bissecao(double a, double b, double TOL, int N, int plot);
double doubleAbs(double a);
double charToDouble(char* doubleInChar);

int main(int argc, char *argv[])
{

	double zero;
	double tol;
	double a,b;
	int n;
	int plot;

	a = charToDouble(argv[1]);
	b = charToDouble(argv[2]);
	tol = charToDouble(argv[3]);
	n = (int) strtoul(argv[4], 0, 10);
	plot = (int) strtoul(argv[5], 0, 10);

	zero = bissecao(a, b, tol, n, plot);
	printf("\n\nf(x) = 0; x = %f\n", zero);


	return 0;
}

double doubleAbs(double a)
{
	return (a < 0) ? (-1*a) : a;
}

double bissecao(double a, double b, double TOL, int N, int plot)
{
	int i = 0;
	double p = 0;
	double FA = 0;
	double FP = 0;
	gnuplot_ctrl *h1;

	if (plot)
	{
		h1 = gnuplot_init();
		gnuplot_plot_slope(h1, 1.0, 0.0, "f(x)") ;

		gnuplot_setstyle(h1, "lines");
		gnuplot_cmd(h1, "set style line 1 lt 2 lw 15");

	}
```

```
	FA = FUNCTION(a);

	while (i < N)
	{
		p = a + (b-a)/2;
		FP = FUNCTION(p);

		if ((FP == 0) || ((b-a)/2 < TOL))
		{
			if (plot)
			{
				gnuplot_resetplot(h1);
				gnuplot_cmd(h1, "set label '(%f,0)' at %f,%f point pointtype 2",p, p, 0.00);
			gnuplot_plot_equation(h1, SFUNCTION(p), "f(x)");
				getchar();
				gnuplot_close(h1);
			}
			printf("\nSolucao encontrada em %d iteracoes", i+1);
			return p;
		}

		printf("\n%f", p);

		if (FA*FP > 0)
		{
			a = p;
			FA = FP;
		}
		else
			b = p;
		i++;
	}
	printf("\nLimite de %d iterecoes atingido. Nao foi possivel determinar o zero da funcao!\n", N);

	getchar();
	if (plot)
		gnuplot_close(h1);

	return p;
}

double charToDouble(char* doubleInChar)
{
	int i = 0;
	int j = 1;
	int signal = 1;
	double integer = 0;
	double decimal = 0;

	if (doubleInChar == NULL)
		return 0;


	if (doubleInChar[0] == '-') //Se o numero for negativo
	{
		signal = -1;
		i = 1;
	}

```

```
        //Define a parte inteira
        while ((doubleInChar[i] != '\0') && (doubleInChar[i] != ',') && (doubleInChar[i] !=
            '.'))
        {
                integer = (integer * 10) + (doubleInChar[i] - 48);
                i++;
        }

                if (doubleInChar[i] != '\0')
                i++; //pula a virgula

        //Define a parte decimal
        while (doubleInChar[i] != '\0')
        {
                decimal += (doubleInChar[i] - 48) / pow(10, j);
                i++;
                j++;
        }

        return signal*(integer + decimal);
}
```

Implementação 3.5: falsaPosicao.c

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#include "gnuplot_i.h"

#define MAX_STRING 80
#define SLEEP_LGTH 2


double polyValue(double* poly, int order, double value);
double falsaPosicao(double* poly, int polyOrder, double p0, double p1, double TOL,
    int N, int plot);
double doubleAbs(double a);
char* linePlotStringPoly(double x0, double y0, double x1, double y1);
double charToDouble(char* doubleInChar);
double* stringToDoublePoly(char* poly, int order);
char* doubleToStringPoly(double* poly, int order);

int main(int argc, char *argv[])
{

        double zero;
        double *poly;
        double tol;
        double p0;
        double p1;
        int order;
        int n;
        int plot;

        order = (int) strtoul(argv[2], 0, 10);
        p0 = charToDouble(argv[3]);
        p1 = charToDouble(argv[4]);
        tol = charToDouble(argv[5]);
        n = (int) strtoul(argv[6], 0, 10);
        plot = (int) strtoul(argv[7], 0, 10);

        poly = stringToDoublePoly(argv[1], order);
```

```

	zero = falsaPosicao(poly, order, p0, p1, tol, n, plot);
	printf("\n\nf(x) = 0; x = %f\n", zero);


	return 0;
}

double doubleAbs(double a)
{
	return (a < 0) ? (-1*a) : a;
}

double polyValue(double* poly, int order, double value)
{
	int i;
	double sum = 0;
	for (i = 0; i < order+1; i++)
		sum += poly[i]*pow(value,(double) i);
	return sum;
}


double falsaPosicao(double* poly, int polyOrder, double p0, double p1, double TOL,
    int N, int plot)
{
	int i;
	double q0;
	double q1;
	double p;
	double q;

	char* functLine;
	gnuplot_ctrl *h1;

	i = 1;
	q0 = polyValue(poly, polyOrder, p0);
	q1 = polyValue(poly, polyOrder, p1);

	if (plot)
	{
	    h1 = gnuplot_init();
		gnuplot_plot_slope(h1, 1.0, 0.0, "f(x)") ;
		gnuplot_plot_slope(h1, 2.0, 0.0, "f'(x)") ;

		gnuplot_setstyle(h1, "lines");
		gnuplot_cmd(h1, "set style line 1 lt 2 lw 15");

		gnuplot_resetplot(h1);
	gnuplot_plot_equation(h1, doubleToStringPoly(poly, polyOrder), "f(x)");
	}

	while (i < N)
	{

		p = p1 - (q1*(p1 - p0)/(q1- q0));
		//printf("\n\n%f %f %f %f %f", p, p0, q0, p1, q1);

		if (plot)
		{
			functLine = linePlotStringPoly(p0, q0, p1, q1);
			gnuplot_setstyle(h1, "lines");
```

```
				gnuplot_plot_equation(h1, functLine, "");
				sleep(SLEEP_LGTH);
			}

			if (doubleAbs(p - p1) < TOL)
			{
				if (plot)
				{
					gnuplot_resetplot(h1);
					gnuplot_cmd(h1, "set label '(%f,0)' at %f,%f point pointtype 2",p, p, 0.00);
				gnuplot_plot_equation(h1, doubleToStringPoly(poly, polyOrder), "f(x)");
					getchar();
					gnuplot_close(h1);
				}

				printf("\nSolucao encontrada em %d iteracoes", i+1);

				return p;
			}

			if (plot)
				printf("\n%f", p);

			q = polyValue(poly, polyOrder, p);

			if ((q * q1) < 0)
			{
				p0 = p1;
				q0 = q1;
			}

			p1 = p;
			q1 = q; //q1 = q

			i++;
		}
		printf("\nLimite de %d iterecoes atingido. Nao foi possivel determinar o zero da funcao!\n", N);

		getchar();
		if (plot)
			gnuplot_close(h1);

		return p;
}

//return f(x) = ax + b
char* linePlotStringPoly(double x0, double y0, double x1, double y1)
{
			char* line;
			double* functDouble = (double*) malloc (sizeof(double) * 2);

			functDouble[1] = (y1 - y0)/ (x1 - x0); //a
			functDouble[0] = y1 - (functDouble[1]*x1); //b

			line = doubleToStringPoly(functDouble, 2);

			return line;
}
```

```c

double* stringToDoublePoly(char* poly, int order)
{
	double* polyInDouble;

	int i;
	int j;
	int beginVector;

	polyInDouble = (double*) malloc (sizeof(double) * (order+1));

	i = beginVector = j = 0;

	while (poly[i] != '\0')
	{
		if (poly[i] == ',')
		{
			poly[i] = '\0';

			polyInDouble[j] = charToDouble(poly + beginVector);
			i++; //ignora espaco vazio

			beginVector = i + 1;
			j++; //quantidade de virgulas
		}

		i++;
	}

	polyInDouble[j] = charToDouble(poly + beginVector);

	return polyInDouble;
}

char* doubleToStringPoly(double* poly, int order)
{
	int i;
	char aux[MAX_STRING];
	char* result;

	result = (char*) malloc (sizeof(char) * MAX_STRING * (order + 1));

	for(i = 0; i < order; i++)
	{
		sprintf(aux, "%f *(x**%d) + " , poly[i], i);
		strcat(result, aux);
	}

	sprintf(aux, "%f *(x**%d)" , poly[i], i);
	strcat(result, aux);

	return result;
}

double charToDouble(char* doubleInChar)
{
	int i = 0;
	int j = 1;
	int signal = 1;
	double integer = 0;
	double decimal = 0;

```

```
	if (doubleInChar == NULL)
		return 0;


	if (doubleInChar[0] == '-') //Se o numero for negativo
	{
		signal = -1;
		i = 1;
	}

	//Define a parte inteira
	while ((doubleInChar[i] != '\0') && (doubleInChar[i] != ',') && (doubleInChar[i] != '.'))
	{
		integer = (integer * 10) + (doubleInChar[i] - 48);
		i++;
	}

		if (doubleInChar[i] != '\0')
		i++; //pula a virgula

	//Define a parte decimal
	while (doubleInChar[i] != '\0')
	{
		decimal += (doubleInChar[i] - 48) / pow(10, j);
		i++;
		j++;
	}

	return signal*(integer + decimal);
}
```

Implementação 3.6: WithDefines/falsaPosicao.c

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#include "gnuplot_i.h"

#include "variables.h"

#define MAX_STRING 80
#define SLEEP_LGTH 2

double falsaPosicao(double p0, double p1, double TOL, int N, int plot);
double doubleAbs(double a);
double charToDouble(char* doubleInChar);
char* linePlotStringPoly(double x0, double y0, double x1, double y1);
char* doubleToStringPoly(double* poly, int order);

int main(int argc, char *argv[])
{

	 double zero;
	 double tol;
	 double p0;
	 double p1;
	 int n;
	 int plot;

	 p0 = charToDouble(argv[1]);
	 p1 = charToDouble(argv[2]);
```

```
        tol = charToDouble(argv[3]);
        n = (int) strtoul(argv[4], 0, 10);
        plot = (int) strtoul(argv[5], 0, 10);

        zero = falsaPosicao(p0, p1, tol, n, plot);
        printf("\n\nf(x) = 0; x = %f\n", zero);


        return 0;
}

double doubleAbs(double a)
{
        return (a < 0) ? (-1*a) : a;
}


double falsaPosicao(double p0, double p1, double TOL, int N, int plot)
{
        int i;
        double q0;
        double q1;
        double p;
        double q;

        char* functLine;
        gnuplot_ctrl *h1;

        i = 1;
        q0 = FUNCTION(p0);
        q1 = FUNCTION(p1);

        if (plot)
        {
            h1 = gnuplot_init();
                gnuplot_plot_slope(h1, 1.0, 0.0, "f(x)") ;
                gnuplot_plot_slope(h1, 2.0, 0.0, "f'(x)") ;

                gnuplot_setstyle(h1, "lines");
                gnuplot_cmd(h1, "set style line 1 lt 2 lw 15");

                gnuplot_resetplot(h1);
        gnuplot_plot_equation(h1, SFUNCTION(x), "f(x)");
        }

        while (i < N)
        {

                p = p1 - (q1*(p1 - p0)/(q1- q0));

                if (plot)
                {
                        functLine = linePlotStringPoly(p0, q0, p1, q1);;
                        gnuplot_setstyle(h1, "lines");
                        gnuplot_plot_equation(h1, functLine, "");
                        sleep(SLEEP_LGTH);
                }

                if (doubleAbs(p - p1) < TOL)
                {
                        if (plot)
                        {
```

```
                gnuplot_resetplot(h1);
                gnuplot_cmd(h1, "set label '(%f,0)' at %f,%f point pointtype 2",p, p, 0.00);
            gnuplot_plot_equation(h1, SFUNCTION(x), "f(x)");
                getchar();
                gnuplot_close(h1);
            }

            printf("\nSolucao encontrada em %d iteracoes", i+1);

            return p;
        }
        printf("\n%f", p);

        q = FUNCTION(p);

        if ((q * q1) < 0)
        {
            p0 = p1;
            q0 = q1;
        }

        p1 = p;
        q1 = q; //q1 = q

        i++;
    }
    printf("\nLimite de %d iterecoes atingido. Nao foi possivel determinar o zero da funcao!\n", N);

    getchar();
    if (plot)
        gnuplot_close(h1);

    return p;
}

//return f(x) = ax + b
char* linePlotStringPoly(double x0, double y0, double x1, double y1)
{
        char* line;
        double* functDouble = (double*) malloc (sizeof(double) * 2);

        functDouble[1] = (y1 - y0)/ (x1 - x0); //a
        functDouble[0] = y1 - (functDouble[1]*x1); //b

        line = doubleToStringPoly(functDouble, 2);

        return line;
}

char* doubleToStringPoly(double* poly, int order)
{
    int i;
    char aux[MAX_STRING];
    char* result;

    result = (char*) malloc (sizeof(char) * MAX_STRING * (order + 1));

    for(i = 0; i < order; i++)
    {
        sprintf(aux, "%f *(x**%d) + " , poly[i], i);
```

```c
                    strcat(result, aux);
          }

          sprintf(aux, "%f *(x**%d)" , poly[i], i);
          strcat(result, aux);

          return result;
}

double charToDouble(char* doubleInChar)
{
        int i = 0;
        int j = 1;
        int signal = 1;
        double integer = 0;
        double decimal = 0;

        if (doubleInChar == NULL)
                return 0;


        if (doubleInChar[0] == '-') //Se o numero for negativo
        {
                signal = -1;
                i = 1;
        }

        //Define a parte inteira
        while ((doubleInChar[i] != '\0') && (doubleInChar[i] != ',') && (doubleInChar[i] !=
            '.'))
        {
                integer = (integer * 10) + (doubleInChar[i] - 48);
                i++;
        }

                if (doubleInChar[i] != '\0')
                i++; //pula a virgula

        //Define a parte decimal
        while (doubleInChar[i] != '\0')
        {
                decimal += (doubleInChar[i] - 48) / pow(10, j);
                i++;
                j++;
        }

        return signal*(integer + decimal);
}
```

Implementação 3.7: secante.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#include "gnuplot_i.h"

#define MAX_STRING 80
#define SLEEP_LGTH 2


double polyValue(double* poly, int order, double value);
```

```
double secante(double* poly, int polyOrder, double p0, double p1, double TOL, int N,
        int plot);
double doubleAbs(double a);
char* linePlotStringPoly(double x0, double y0, double x1, double y1);
double charToDouble(char* doubleInChar);
double* stringToDoublePoly(char* poly, int order);
char* doubleToStringPoly(double* poly, int order);

int main(int argc, char *argv[])
{

        double zero;
        double *poly;
        double tol;
        double p0;
        double p1;
        int order;
        int n;
        int plot;

        order = (int) strtoul(argv[2], 0, 10);
        p0 = charToDouble(argv[3]);
        p1 = charToDouble(argv[4]);
        tol = charToDouble(argv[5]);
        n = (int) strtoul(argv[6], 0, 10);
        plot = (int) strtoul(argv[7], 0, 10);

        poly = stringToDoublePoly(argv[1], order);

        zero = secante(poly, order, p0, p1, tol, n, plot);
        printf("\n\nf(x) = 0; x = %f\n", zero);


        return 0;
}

double doubleAbs(double a)
{
        return (a < 0) ? (-1*a) : a;
}

double polyValue(double* poly, int order, double value)
{
        int i;
        double sum = 0;
        for (i = 0; i < order+1; i++)
                sum += poly[i]*pow(value,(double) i);
        return sum;
}


double secante(double* poly, int polyOrder, double p0, double p1, double TOL, int N,
        int plot)
{
        int i;
        double q0;
        double q1;
        double p;

        char* functLine;
        gnuplot_ctrl *h1;

```

```
    i = 1;
    q0 = polyValue(poly, polyOrder, p0);
    q1 = polyValue(poly, polyOrder, p1);

    if (plot)
    {
        h1 = gnuplot_init();
            gnuplot_plot_slope(h1, 1.0, 0.0, "f(x)") ;
            gnuplot_plot_slope(h1, 2.0, 0.0, "f'(x)") ;

            gnuplot_setstyle(h1, "lines");
            gnuplot_cmd(h1, "set style line 1 lt 2 lw 15");

            gnuplot_resetplot(h1);
    gnuplot_plot_equation(h1, doubleToStringPoly(poly, polyOrder), "f(x)");
    }

    while (i < N)
    {

            p = p1 - (q1*(p1 - p0)/(q1- q0));

            if (plot)
            {
                functLine = linePlotStringPoly(p0, q0, p1, q1);
                gnuplot_setstyle(h1, "lines");
                gnuplot_plot_equation(h1, functLine, "");
                sleep(SLEEP_LGTH);
            }

            if (doubleAbs(p - p1) < TOL)
            {
                if (plot)
                {
                    gnuplot_resetplot(h1);
                    gnuplot_cmd(h1, "set label '(%f,0)' at %f,%f point pointtype 2",p, p, 0.00);
                gnuplot_plot_equation(h1, doubleToStringPoly(poly, polyOrder), "f(x)");
                    getchar();
                    gnuplot_close(h1);
                }

                printf("\nSolucao encontrada em %d iteracoes", i+1);

                return p;
            }

            if (plot)
                printf("\n%f", p);

            p0 = p1;
            q0 = q1;
            p1 = p;
            q1 = polyValue(poly, polyOrder, p); //q1 = f(p)

            i++;
    }
    printf("\nLimite de %d iterecoes atingido. Nao foi possivel determinar o zero da funcao!\n", N);

    getchar();
```

```c
	if (plot)
		gnuplot_close(h1);

	return p;
}

//return f(x) = ax + b
char* linePlotStringPoly(double x0, double y0, double x1, double y1)
{
		char* line;
		double* functDouble = (double*) malloc (sizeof(double) * 2);

		functDouble[1] = (y1 - y0)/ (x1 - x0); //a
		functDouble[0] = y1 - (functDouble[1]*x1); //b

		line = doubleToStringPoly(functDouble, 2);

		return line;
}


double* stringToDoublePoly(char* poly, int order)
{
	double* polyInDouble;

	int i;
	int j;
	int beginVector;

	polyInDouble = (double*) malloc (sizeof(double) * (order+1));

	i = beginVector = j = 0;

	while (poly[i] != '\0')
	{
		if (poly[i] == ',')
		{
			poly[i] = '\0';

			polyInDouble[j] = charToDouble(poly + beginVector);
			i++; //ignora espaco vazio

			beginVector = i + 1;
			j++; //quantidade de virgulas
		}

		i++;
	}

	polyInDouble[j] = charToDouble(poly + beginVector);

	return polyInDouble;
}

char* doubleToStringPoly(double* poly, int order)
{
	int i;
	char aux[MAX_STRING];
	char* result;

	result = (char*) malloc (sizeof(char) * MAX_STRING * (order + 1));

```

```c
        for(i = 0; i < order; i++)
        {
                sprintf(aux, "%f *(x**%d) + ", poly[i], i);
                strcat(result, aux);
        }

        sprintf(aux, "%f *(x**%d)", poly[i], i);
        strcat(result, aux);

        return result;
}

double charToDouble(char* doubleInChar)
{
        int i = 0;
        int j = 1;
        int signal = 1;
        double integer = 0;
        double decimal = 0;

        if (doubleInChar == NULL)
                return 0;


        if (doubleInChar[0] == '-') //Se o numero for negativo
        {
                signal = -1;
                i = 1;
        }

        //Define a parte inteira
        while ((doubleInChar[i] != '\0') && (doubleInChar[i] != ',') && (doubleInChar[i] != '.'))
        {
                integer = (integer * 10) + (doubleInChar[i] - 48);
                i++;
        }

                if (doubleInChar[i] != '\0')
                i++; //pula a virgula

        //Define a parte decimal
        while (doubleInChar[i] != '\0')
        {
                decimal += (doubleInChar[i] - 48) / pow(10, j);
                i++;
                j++;
        }

        return signal*(integer + decimal);
}
```

Implementação 3.8: WithDefines/secante.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#include "gnuplot_i.h"
#include "variables.h"

#define MAX_STRING 80
#define SLEEP_LGTH 2
```

```
double secante(double p0, double p1, double TOL, int N, int plot);
double doubleAbs(double a);
char* linePlotStringPoly(double x0, double y0, double x1, double y1);
double charToDouble(char* doubleInChar);
char* doubleToStringPoly(double* poly, int order);

int main(int argc, char *argv[])
{

        double zero;
        double tol;
        double p0;
        double p1;
        int n;
        int plot;

        p0 = charToDouble(argv[1]);
        p1 = charToDouble(argv[2]);
        tol = charToDouble(argv[3]);
        n = (int) strtoul(argv[4], 0, 10);
        plot = (int) strtoul(argv[5], 0, 10);

        zero = secante(p0, p1, tol, n, plot);
        printf("\n\nf(x) = 0; x = %f\n", zero);


        return 0;
}

double doubleAbs(double a)
{
        return (a < 0) ? (-1*a) : a;
}


double secante(double p0, double p1, double TOL, int N, int plot)
{
        int i;
        double q0;
        double q1;
        double p;

        char* functLine;
        gnuplot_ctrl *h1;

        i = 1;
        q0 = FUNCTION(p0);
        q1 = FUNCTION(p1);

        if (plot)
        {
            h1 = gnuplot_init();
                gnuplot_plot_slope(h1, 1.0, 0.0, "f(x)") ;
                gnuplot_plot_slope(h1, 2.0, 0.0, "f'(x)") ;

                gnuplot_setstyle(h1, "lines");
                gnuplot_cmd(h1, "set style line 1 lt 2 lw 15");

                gnuplot_resetplot(h1);
        gnuplot_plot_equation(h1, SFUNCTION(x), "f(x)");
```

```
	}

	while (i < N)
	{

		p = p1 - (q1*(p1 - p0)/(q1- q0));

		if (plot)
		{
			functLine = linePlotStringPoly(p0, q0, p1, q1);
			gnuplot_setstyle(h1, "lines");
			gnuplot_plot_equation(h1, functLine, "");
			sleep(SLEEP_LGTH);
		}

		if (doubleAbs(p - p1) < TOL)
		{
			if (plot)
			{
				gnuplot_resetplot(h1);
				gnuplot_cmd(h1, "set label '(%f,0)' at %f,%f point pointtype 2",p, p, 0.00);
			gnuplot_plot_equation(h1, SFUNCTION(x), "f(x)");
				getchar();
				gnuplot_close(h1);
			}

			printf("\nSolucao encontrada em %d iteracoes", i+1);

			return p;
		}

		p0 = p1;
		q0 = q1;
		p1 = p;
		q1 = FUNCTION(p); //q1 = f(p)

		i++;
	}
	printf("\nLimite de %d iterecoes atingido. Nao foi possivel determinar o zero da funcao!\n", N);

	getchar();
	if (plot)
		gnuplot_close(h1);

	return p;
}

//return f(x) = ax + b
char* linePlotStringPoly(double x0, double y0, double x1, double y1)
{
		char* line;
		double* functDouble = (double*) malloc (sizeof(double) * 2);

		functDouble[1] = (y1 - y0)/ (x1 - x0); //a
		functDouble[0] = y1 - (functDouble[1]*x1); //b

		line = doubleToStringPoly(functDouble, 2);

		return line;
}
```

```
double* stringToDoublePoly(char* poly, int order)
{
	double* polyInDouble;

	int i;
	int j;
	int beginVector;

	polyInDouble = (double*) malloc (sizeof(double) * (order+1));

	i = beginVector = j = 0;

	while (poly[i] != '\0')
	{
		if (poly[i] == ',')
		{
			poly[i] = '\0';

			polyInDouble[j] = charToDouble(poly + beginVector);
			i++; //ignora espaco vazio

			beginVector = i + 1;
			j++; //quantidade de virgulas
		}

		i++;
	}

	polyInDouble[j] = charToDouble(poly + beginVector);

	return polyInDouble;
}

char* doubleToStringPoly(double* poly, int order)
{
	int i;
	char aux[MAX_STRING];
	char* result;

	result = (char*) malloc (sizeof(char) * MAX_STRING * (order + 1));

	for(i = 0; i < order; i++)
	{
		sprintf(aux, "%f *(x**%d) + " , poly[i], i);
		strcat(result, aux);
	}

	sprintf(aux, "%f *(x**%d)" , poly[i], i);
	strcat(result, aux);

	return result;
}

double charToDouble(char* doubleInChar)
{
	int i = 0;
	int j = 1;
	int signal = 1;
	double integer = 0;
	double decimal = 0;
```

```
        if (doubleInChar == NULL)
                return 0;


        if (doubleInChar[0] == '−') //Se o numero for negativo
        {
                signal = −1;
                i = 1;
        }

        //Define a parte inteira
        while ((doubleInChar[i] != '\0') && (doubleInChar[i] != ',') && (doubleInChar[i] !=
                '.'))
        {
                integer = (integer ∗ 10) + (doubleInChar[i] − 48);
                i++;
        }

                if (doubleInChar[i] != '\0')
                i++; //pula a virgula

        //Define a parte decimal
        while (doubleInChar[i] != '\0')
        {
                decimal += (doubleInChar[i] − 48) / pow(10, j);
                i++;
                j++;
        }

        return signal∗(integer + decimal);
}
```

Referências Bibliográficas

[1] Richard L. Burden and J. Douglas Faires. Análise Numérica. Cengage, tradução da 8ª edição norte-americana edition, 2008.

[2] GNU Plot. Documentation of gnuplot 4.2, An Interactive Plotting Program, Setembro 2009. Disponível em <http://www.gnuplot.info/docs_4.2/gnuplot.html>.