Aplicação do Método de Newton-Raphson para Resolver o Cálculo Inverso do Método da Mínima Curvatura*

Tiago C. A. Amorim[2]

[a]*Petrobras, Av. Henrique Valadares, 28, Rio de Janeiro, 20231-030, RJ, Brasil*

Abstract

A partir da formulação do Método da Mínima Curvatura, foi desenvolvido um algoritmo para fazer o cálculo de parâmetros de perfuração em função de coordenadas cartesianas. Este algoritmo tem formulação implícita, e pode ser resolvido utilizando métodos de busca de raiz de funções de uma variável. Neste relatório foi aplicado o Método de Newton-Raphson para resolver este problema. Com um ajuste ao método foi possível encontrar soluções para este problema com uma taxa de convergência duas vezes maior que aquela consguida com o Método da Bissecção.

Keywords: Método da Mínima Curvatura, Método de Newton-Raphson

---

1. Introdução

O Método da Mínima Curvatura possibilita o cálculo de cooordenadas cartesianas (ΔN, ΔE, ΔV) a partir de parâmetros perfuração (ΔS, θ, ϕ) assumindo que a trajetória de um poço direcional pode ser descrita como uma série de arcos [1]. Não existe formulação direta para o cálculo inverso, isto é, dos parâmetros de perfuração em função das coordenadas cartesianas.

No primeiro relatório desta série [2] foi apresentada uma proposta de metodologia para cálculo dos parâmetros de perfuração a partir das coordenadas cartesianas de pontos ao longo da geometria do poço. O algoritmo proposto tem formulação implícita e foi organizado como um problema do tipo $g(x) = x$. O problema foi resolvido com o Método da Bissecção, encontrando bons resultados.

Neste segundo relatório é aplicado o método de Newton-Raphson de para resolver o mesmo problema.

---

*Relatório número 2 como parte dos requisitos da disciplina IM253: Métodos Numéricos para Fenômenos de Transporte.

**Atualmente cursando doutorado no Departamento de Engenharia de Petróleo da Faculdade de Engenharia Mecânica da UNICAMP (Campinas/SP, Brasil).

Email address: `t100675@dac.unicamp.br` (Tiago C. A. Amorim)

2. Metodologia

2.1. Derivada do Algoritmo Proposto

A seguir são apresentadas as fórmulas que foram do algoritmo proposto para cálculo de parâmetros de perfuração em função de coordenadas cartesianas [2]. O algoritmo assume que para calcular os parâmtros de perfuração entre dois pontos quaisquer são conhecidos os parâmetros de perfuração do ponto inicial[1] (θ_1, ϕ_1) e as distâncias cartesianas entre os pontos (ΔN,ΔE, ΔV). O objetivo é calcular θ_2, ϕ_2 e ΔS:

$$\cos\theta_2 = 2\frac{\Delta V}{\Delta S f(\alpha)} - \cos\theta_1 \tag{1}$$

$$\sin\phi_2 = \frac{-\Delta N \Delta\Psi}{\Delta H^2}\left(\frac{\sin\theta_1}{\sin\theta_2}\right) + \frac{\Delta E}{\Delta H^2}\sqrt{\Delta H^2 - \Delta\Psi^2\left(\frac{\sin\theta_1}{\sin\theta_2}\right)^2} \tag{2}$$

$$\cos\phi_2 = \frac{\Delta E \Delta\Psi}{\Delta H^2}\left(\frac{\sin\theta_1}{\sin\theta_2}\right) + \frac{\Delta N}{\Delta H^2}\sqrt{\frac{1}{\Delta H^2} - \Delta\Psi^2\left(\frac{\sin\theta_1}{\sin\theta_2}\right)^2} \tag{3}$$

onde

$$\Delta\Psi = \Delta N \sin\phi_1 - \Delta E \cos\phi_1$$
$$\Delta H^2 = \Delta N^2 + \Delta E^2$$

$$\Delta S f(\alpha) = 2\sqrt{\frac{\Delta N^2 + \Delta E^2 + \Delta V^2}{A^2 + B^2 + C^2}} \tag{4}$$

onde

$$A = \sin\theta_1\cos\phi_1 + \sin\theta_2\cos\phi_2$$
$$B = \sin\theta_1\sin\phi_1 + \sin\theta_2\sin\phi_2$$
$$C = \cos\theta_1 + \cos\theta_2$$

Com as equações propostas é possível construir uma função do tipo $g(x) = x$:

1. Assumir um valor inicial de $\Delta S f(\alpha)$.
2. Calcular $\cos\theta_2$ com a equação 1.
3. Calcular $\sin\phi_2$ com a equação 2.
4. Calcular $\cos\phi_2$ com a equação 3.
5. Calcular $\Delta S f(\alpha)$ com a equação 4.

[1]Para o primeiro ponto da trajetória é assumido um poço na vertical: $\theta = 0$, $\phi = 0$.

Em [2] são discutidos detalhes adicionais sobre a implementação deste algoritmo.

Para utilizar o método de Newton-Raphson para resolver o problema proposto, será preciso encontrar a derivada da função com relação a x de $f(x) = g(x) - x$. Para facilitar a leitura, $\Delta S f(\alpha)$ é substituído por x na equação 4, e então é definida a função $f(x)$[2]:

$$f(x) = 2\sqrt{\frac{\Delta N^2 + \Delta E^2 + \Delta V^2}{A(x)^2 + B(x)^2 + C(x)^2}} - x \tag{5}$$

onde

$$\begin{aligned}
A(x) &= \sin\theta_1 \cos\phi_1 + \sin\theta_2(x)\cos\phi_2(x) \\
B(x) &= \sin\theta_1 \sin\phi_1 + \sin\theta_2(x)\sin\phi_2(x) \\
C(x) &= \cos\theta_1 + \cos\theta_2(x)
\end{aligned}$$

Derivando $f(x)$:

$$f(x)' = -\frac{2}{\Delta S^2}\left(\frac{\Delta N^2 + \Delta E^2 + \Delta V^2}{A(x)^2 + B(x)^2 + C(x)^2}\right)^{3/2} [A(x)A(x)' + B(x)B(x)' + C(x)C(x)'] - 1 \tag{6}$$

onde

$$\begin{aligned}
A(x)' &= \sin\theta_2(x)' \cos\phi_2(x) + \sin\theta_2(x)\cos\phi_2(x)' \\
B(x)' &= \sin\theta_2(x)' \sin\phi_2(x) + \sin\theta_2(x)\sin\phi_2(x)' \\
C(x)' &= \cos\theta_2(x)'
\end{aligned}$$

Os demais termos das equações são obtidos derivando as equações 1, 2 e 3:

$$\cos\theta_2(x)' = -\frac{2\Delta V}{x^2} \tag{7}$$

$$\sin\phi_2(x)' = \frac{\frac{\Delta\Psi \sin\theta_1}{\sin\theta_2(x)}}{\Delta H^2}\left(\Delta N + \Delta E \frac{\frac{\Delta\Psi \sin\theta_1}{\sin\theta_2(x)}}{\sqrt{\Delta H^2 - \left(\frac{\Delta\Psi \sin\theta_1}{\sin\theta_2(x)}\right)^2}}\right)\frac{\sin\theta_2(x)'}{\sin\theta_2(x)} \tag{8}$$

[2]Os termos que dependem de ΔS serão explicitados usando (x).

$$\cos\phi_2(x)' = \frac{\frac{\Delta\Psi\sin\theta_1}{\sin\theta_2(x)}}{\Delta H^2}\left(\Delta E - \Delta N\frac{\frac{\Delta\Psi\sin\theta_1}{\sin\theta_2(x)}}{\sqrt{\Delta H^2 - \left(\frac{\Delta\Psi\sin\theta_1}{\sin\theta_2(x)}\right)^2}}\right)\frac{\sin\theta_2(x)'}{\sin\theta_2(x)} \tag{9}$$

onde $\Delta\Psi$ e ΔH^2 seguem as mesmas definições das equações 2 e 3.

A definição de $\sin\theta_2(x)'$ pode ser desenvolvida a partir de $\cos^2\theta_2(x) + \sin^2\theta_2(x) = 1$:

$$\sin\theta_2(x)' = -\frac{\frac{2\Delta V}{x}(\cos\theta_1 - \frac{2\Delta V}{x})}{x\sqrt{1-\left(\cos\theta_1 - \frac{2\Delta V}{x}\right)^2}} \tag{10}$$

2.2. Método de Newton-Raphson

O código desenvolvido foi baseado no pseudo-código de [3]. O Método de Newton-Raphson faz uso da derivada da função de interesse. O método precisa apenas de um ponto inicial, e converge mais rápido que o Método da Bissecção, mas não tem garantia de encontrar uma raiz.

De modo simplificado o Método de Newton-Raphson pode ser descrito como:

1. Definir x_0.
2. Iniciar contagem de iterações com $i = 0$.
3. Calcular $f(x_i)$ e $f(x_i)'$.
4. Calcular $x_{i+1} = x_i - \frac{f(x_i)}{f(x_i)'}$.
5. Incrementar a contagem de iterações: $i = i + 1$.
6. Se não atingiu critério de convergência, retornar para passo 3.
7. Retornar x_{i+1}..

Foram adicionados critérios adicionais ao algoritmo para controlar o processo iterativo:

- Foram implementados dois métodos de cálculo do critério de convergência[3]: *Direto* e *Relativo*.
- É feito um término prematuro do processo iterativo caso algum $|f(x)| < \epsilon$. O valor padrão de ϵ é 10^{-7} (variável `epsilon` no código). Este teste também é feito antes de entrar no *loop*.
- A cada iteração é verificado se a derivada da função está muito próxima de zero: $|f(x)'| < \epsilon$. Em caso positivo é gerada uma mensagem e é encerrado o *loop*.
- Como o método não garante convergência, ao longo das iterações é guardado o melhor resultado (x tal que $|f(x)|$ seja mínimo). Este é o resultado que é retornado.

[3]Ver descrição em [2].

3. Resultados

Para facilitar a análise da qualidade do código desenvolvido, foram criadas funções que realizam diversos testes onde a resposta exata é conhecida:

tests_newton_raphson() Testa o Método de Newton-Raphson em diferentes funções: linear, quadrática, exponencial e trigonométrica. Também foram apicados casos específicos para o algoritmo tratar: raiz no ponto inicial, ponto inicial muito próximo da raiz, ponto inicial em um ponto de mínimo da função (derivada nula), uso de critério de convergência relativo, função com derivada nula na raiz, função com vários mínimos e máximos locais.

tests_minimum_curvature() Testa as funções implementadas para cálculo de coordenadas cartesianas em função de parâmetros de perfuração (cálculo direto), e de parâmetros de perfuração em função de coordenadas cartesianas (cálculo iterativo).

A implementação do Método de Newton-Raphson teve uma alteração adicional em função dos resultados dos testes realizados. Nas fórmulas do problema proposto fica evidente que a variável principal ($\Delta Sf(\alpha)$) não pode ter um valor qualquer. Para evitar simplesmente parar o processo iterativo quando o próximo ponto é menor que o limite inferior, foi implementada uma verificação adicional no método: caso o novo valor de x estiver fora dos limites pré-estabelecidos, usará o valor do limite que foi violado. Desta forma é dada uma *chance* para o processo iterativo conseguir convergir para uma resposta melhor. Para os cálculos de parâmetros de perfuração foi utilizado um limite inferior de $\Delta Sf(\alpha)$ igual ao discutido em [2].

As funções polinomiais testadas convergiram muito rapidamente, mesmo as que tinham a derivada nula na raiz. O único teste em que o Método da Bissecção teve uma convergência mais rápida foi na função exponencial: $e^{x^2-4} - 1$ (Tabela 1). Esta função é especialmente difícil para o Método de Newton-Raphson porque tem uma mudança brusca da derivada à esquerda e à direita de cada raiz, tornando o método instável na região de derivada muito próxima de zero ($-2 < x < 2$) e com baixa taxa de convergência nas regiões de derivada muito alta ($x > 2$) ou muito baixa ($x < -2$).

Enquanto o Método da Bisseção para as diferentes funções testadas o erro reduziu em uma razão de aproximadamente 2 ([2]), o Método de Newton-Raphson teve comportamentos variados: convergência muito rápida para funções quadráticas e trigonométricas, convergência muito lenta para a função exponencial. Para o problema proposto a o erro caiu em uma razão de aproximadamente 4, ou seja, duas vezes mais rápido que com o Método da Bissecção (Ver Apêndice B.1).

Para algumas funções o Método de Newton-Raphson se mostrou instável, ficando muito dependente do ponto inicial para conseguir convergir. A função $\cos x + \sin x$ tem muitas raízes, e o resultado do Método de Newton-Raphson se mostrou muito dependente do ponto inicial. Para conseguir encontrar a raiz de uma determinada região foi preciso definir um ponto inicial *bem próximo* da raiz. Já a função $3x + x^2 \sin^2 \frac{x\pi}{10}$ mostrou muita instabilidade,

Tabela 1: Comparação entre o número de iterações necessários para um mesmo critério de convergência usando os Métodos da Bissecção e Newton-Raphson.

Função	Raiz	Bissecção			Newton-Raphson	
		x_a	x_b	Iter.	x_0	Iter.
Linear	0.3	0.	2.	11	0.	1
Quadrática	-0.1	-0.25	1.	11	0.25	4
Exponencial	2.	0.	10.	14	5.	25
Trigonométrica	$3\pi/4$	0.	5.	13	3.	3
1/4 círculo horizontal	20	14.1421	120.417	17	14.4743	1
Seção 2 do poço em S	131.652	130.526	1111.4	20	133.592	11
Poço 3D	10.3144	9.84918	83.8634	17	10.0805	4

em muitos casos não conseguiu convergir ao fazer pequenas mudanças no valor do ponto inicial.

Na aplicação do Método de Newton com o algoritmo desenvolvido para cálculo de parâmetros de perfuração os resultados foram muito bons. Nenhum dos testes teve problemas de convergência, e convergiram com um número de iterações menor que o do Método da Bissecção. O formato da função de construída favorece o uso do Método de Newton-Raphson, pois tem a forma próxima de uma reta (ver Apêndice A).

Para o problema proposto foram testados alguns valores distintos de ponto inicial, mas o impacto no resultado não é expressivo. Um valor inicial *melhor* irá resultar em uma ou duas iterações a menos. Inicializar no valor mínimo de $\Delta Sf(\alpha)$ não teve um resultado muito bom em alguns dos testes realizados, pois nesta região a derivada tem maior variação. No código foi utilizado por padrão um valor 10% maior que o mínimo.

4. Conclusão

O Método de Newton-Raphson se mostrou adequado para fazer o cálculo de parâmetros de perfuração em função de coordenadas cartesianas. Foi necessário adaptar o Método de Newton-Raphson para conseguir continuar o processo iterativo caso o novo valor de teste não esteja na região de validade da função. Com este ajuste todos os testes realizados alcançaram o critério de convergência com um número de iterações menor do que o que foi alcançado com o Método da Bissecção.

Referências

[1] A Compendium of Directional Calculations Based on the Minimum Curvature Method, Vol. All Days of SPE Annual Technical Conference and Exhibition. `arXiv:https://onepetro.org/SPEATCE/proceedings-pdf/03ATCE/All-03ATCE/SPE-84246-MS/2895686/spe-84246-ms.pdf`, `doi:10.2118/84246-MS`.
URL `https://doi.org/10.2118/84246-MS`

[2] T. C. A. Amorim, Proposta de cálculo de parâmetros de perfuração de poços de petróleo a partir de coordenadas espaciais com o método da bissecção, Relatório número 1 da disciplina IM253: Métodos Numéricos para Fenômenos de Transporte (08 2023).

[3] R. L. Burden, J. D. Faires, A. M. Burden, Análise numérica, Cengage Learning, 2016.

Apêndice A. Funções de Teste

Nos gráficos a seguir as curvas contínuas são as funções de teste e as tracejadas são as respectivas derivadas. A raiz (ou raízes) está denotada pelo quadrado azul. São apresentadas apenas algumas das funções de teste.

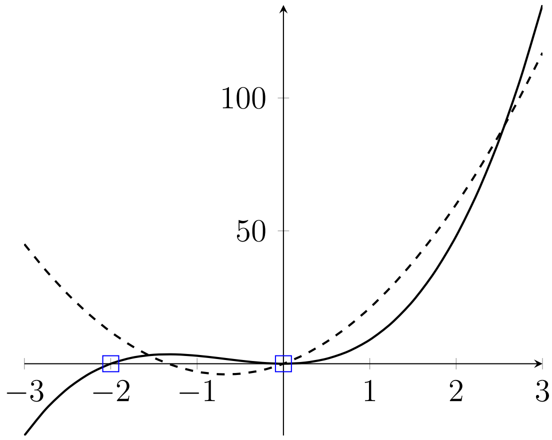

Figura A.1: Polinômio de 3º grau: $3x^3 + 2x^2$.

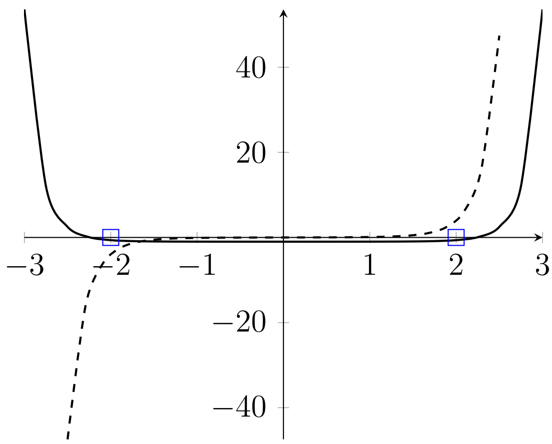

Figura A.2: Função exponencial: $e^{x^2-4} - 1$.

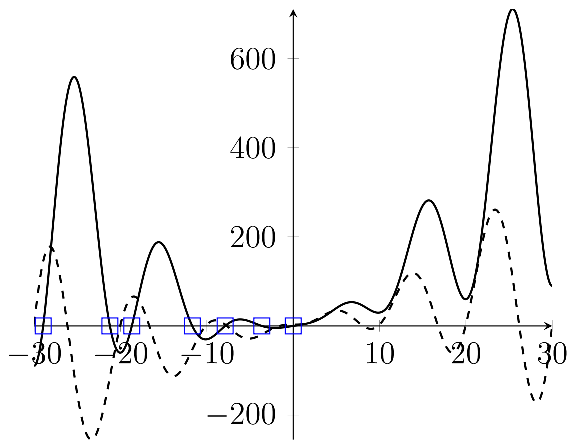

Figura A.3: Função trigonométrica: $3x + x^2 \sin^2 \frac{x\pi}{10}$.

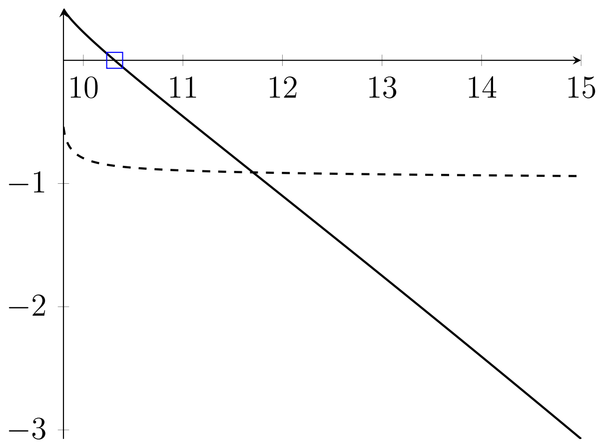

Figura A.4: Função de erro de $\Delta Sf(\alpha)$ para o caso Poço 3D.

Apêndice B. Gráficos de Diagnóstico

Apêndice B.1. Erro ao Longo da Iterações

A figura abaixo apresenta a evolução do erro, com relação à resposta exata, ao longo do processo iterativo de diferentes funções, utilizando o Método da Newton-Raphson. São apresentadas duas retas tracejadas onde a razão entre os erros de iterações sucessivas é 1/2 e 1/4.

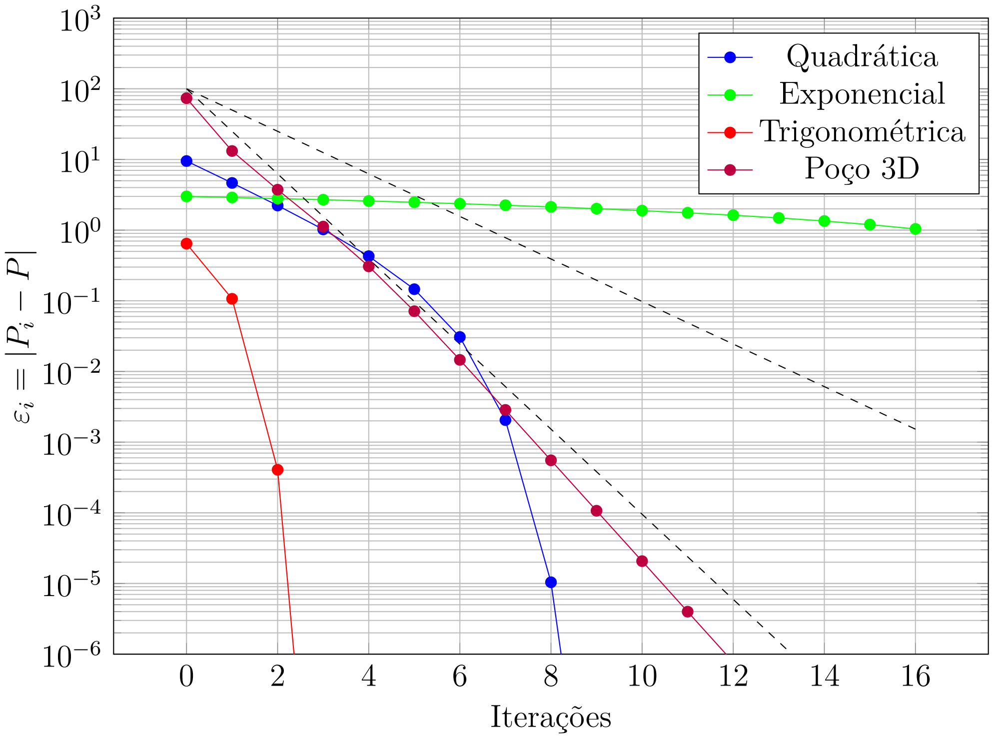

Figura B.5: Evolução do erro do Método de Newton-Raphson com diferentes funções (linhas tracejadas representam $\varepsilon_{i+1}/\varepsilon_i = 0.5$ e $\varepsilon_{i+1}/\varepsilon_i = 0.25$).

Apêndice B.2. Efeito do Ponto Inicial no Número de Iterações

A figura a seguir mostra o impacto da definição do ponto inicial em um dos testes realizados com o algoritmo de cálculo de parâmetros de perfuração em função de coordenadas cartesianas. Neste caso específico a resposta exata é aproximadamente 4.7% maior que $\Delta Sf(\alpha)_{min}$, e os pontos iniciais mais próximos convergem mais rápido. O principal resultado deste gráfico é mostrar que a definição do ponto inicial tem pouco impacto no número de iterações que o Método de Newton-Raphson precisará para alcançar o critério de convergência definido.

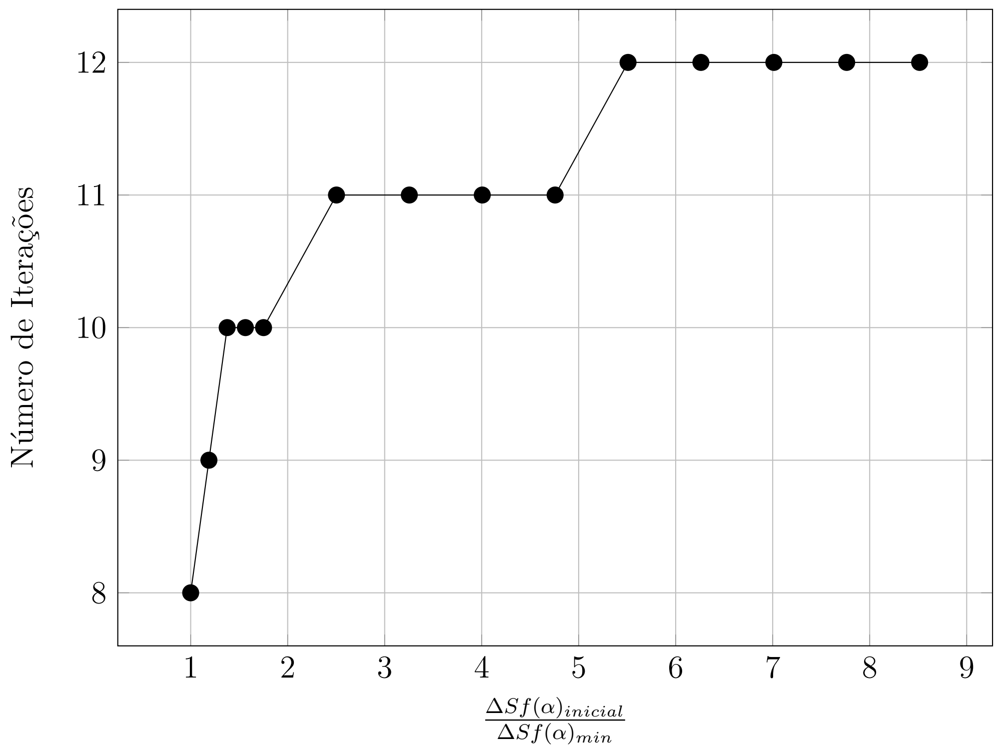

Figura B.6: Número de iterações em função do valor inicial da variável do problema.

Apêndice C. Código em C

O código de ambos métodos foi implementado em um único arquivo. O código é apresentado em duas partes neste documento para facilitar a leitura. O código pode ser encontrado em https://github.com/TiagoCAAmorim/numerical-methods.

Apêndice C.1. Método da Bissecção

```c
/*
    Implementation of the bissection method to find root of 1D functions
*/

#include <stdio.h>
#include <stdbool.h>
#include <math.h>
#include <assert.h>

const double epsilon = 1E-7;
const double pi = 3.14159265358979323846;

const int default_iterations = 100;
const double default_convergence = 1E-4;

const bool stop_on_error = true;
```

```

bool check_error(bool ok_condition, char *error_message){
    if( !ok_condition){
        printf(error_message);
        if( stop_on_error){
            assert(false);
        }
    }
    return ok_condition;
}

int imin(int a, int b){
    return a<b? a: b;
}

int imax(int a, int b){
    return a>b? a: b;
}

double min(double a, double b){
    return a<b? a: b;
}

double max(double a, double b){
    return a>b? a: b;
}

bool is_root(double fx, double x){
    return fabs(fx) < epsilon;
}

double calculate_convergence(double x_current, double x_previous, bool
   relative_convergence){
    double reference = 1;
    if( relative_convergence)
        if( fabs(x_current) > epsilon){
            reference = x_current;
        } else if( fabs(x_previous) > epsilon) {
            reference = x_previous;
        } else{
            return 0.;
        }
    return fabs( (x_current - x_previous) / reference );
}

void print_error(char *message, double true_value, double calculated_value,
   double error_limit, bool relative_convergence){
    double relative_error;
    relative_error = calculate_convergence(true_value, calculated_value,
   relative_convergence);
```

```
    printf("%s: True=%g Calculated=%g Error=%g", message, true_value,
    calculated_value, relative_error);
    if( relative_error > error_limit){
        printf("  <= ######### Attention ###########");
    }
    printf("\n");
}

double return_closest_to_root(double x_a, double fx_a, double x_b, double
    fx_b){
    if( fabs(fx_b) < fabs(fx_a)){
        return x_b;
    } else{
        return x_a;
    }
}

double return_closest_to_root_3pt(double x_a, double fx_a, double x_b, double
     fx_b, double x_c, double fx_c){
    fx_a = fabs(fx_a);
    fx_b = fabs(fx_b);
    fx_c = fabs(fx_c);
    if( min(fx_a, fx_b) < fx_c){
        if( fx_b < fx_a){
            return x_b;
        } else{
            return x_a;
        }
    } else{
        return x_c;
    }
}

double find_root_bissection_debug(double (*func)(double), double x_a, double
    x_b, double convergence_tol, bool relative_convergence, int max_iterations
    , bool debug, double x_root) {
    double fx_a, fx_b, fx_mean;
    double x_mean, x_mean_previous;
    double convergence;
    enum type_exit_function {no_exit, root, converged, sign_error};
    enum type_exit_function exit_function = no_exit;
    bool print_true_error;
    int i=0;

    if( x_b < x_a){
        x_mean = x_a;
        x_a = x_b;
        x_b = x_mean;
    }

    fx_a = func(x_a);
```

```
    fx_b = func(x_b);
    x_mean = 1e20;
    fx_mean = 1e20;

    convergence = calculate_convergence(min(x_a, x_b), max(x_a, x_b), relative_convergence);
    if( is_root(fx_a, x_a) || is_root(fx_b, x_b)){
        exit_function = root;
    };

    if( exit_function == no_exit){
        if( convergence < convergence_tol ){
            if( debug){
                printf("Initial limits are closer than convergence criteria: |%g - %g| = %g < %g.\n",x_b,x_a,fabs(x_a - x_b), convergence_tol);
            }
            exit_function = converged;
        }
    }

    if( exit_function == no_exit){
        if( signbit(fx_a) == signbit(fx_b) ){
            printf("Function has same sign in limits: f(%g) = %g f(%g) = %g.\n",x_a,fx_a,x_b,fx_b);
            if( debug){
                double x_trial;
                printf("Sample of function results in the provided domain:\n");
                for(int i=0; i<=20; i++){
                    x_trial = x_a + (x_b - x_a)*i/20;
                    printf("  f(%g) = %g\n", x_trial, func(x_trial));
                }
            }
            exit_function = sign_error;
        }
    }

    if( !exit_function){
        print_true_error = (x_root >= x_a) && (x_root <= x_b);
        x_mean_previous = x_a;
        if( debug){
            if( print_true_error){
                printf("%3s   %-28s\t%-28s\t%-28s\t%-11s\t%-11s\n","#", "Lower bound", "Upper bound", "Mean point", "Convergence","|x - root|");
            } else{
                printf("%3s   %-28s\t%-28s\t%-28s\t%-11s\n","#", "Lower bound", "Upper bound", "Mean point", "Convergence");
            }
        }

        for(i=1 ; i<=max_iterations ; i++){
```

```
            x_mean = x_a + (x_b - x_a)/2;
            fx_mean = func(x_mean);
            convergence = calculate_convergence(x_mean, x_mean_previous, relative_convergence);
            x_mean_previous = x_mean;

            if( debug){
                if( print_true_error){
                    printf("%3i.  f(%11g) = %11g\tf(%11g) = %11g\tf(%11g) = %11g\t%11g\t%11g\n",i, x_a, fx_a, x_b, fx_b, x_mean, fx_mean, convergence, fabs(x_mean - x_root));
                } else{
                    printf("%3i.  f(%11g) = %11g\tf(%11g) = %11g\tf(%11g) = %11g\t%11g\n",i, x_a, fx_a, x_b, fx_b, x_mean, fx_mean, convergence);
                }
            }

            if( convergence < convergence_tol){
                exit_function = converged;
                break;
            }

            if( is_root(fx_mean, x_mean)){
                exit_function = root;
                break;
            }

            if( signbit(fx_a) == signbit(fx_mean)){
                x_a = x_mean;
                fx_a = fx_mean;
            } else{
                x_b = x_mean;
                fx_b = fx_mean;
            }
        }
    }
    if( debug){
        switch(exit_function)
        {
            case root: printf("Found |f(x)| < %g after %i iterations.\n", epsilon, i); break;
            case converged: printf("Reached convergence after %i iteration(s): %g < %g.\n", i, convergence, convergence_tol); break;
            case sign_error: printf("Cannot continue. Returning result closest to zero amongst f(x_a) and f(x_b).\n"); break;
            case no_exit: printf("Convergence was not reached after %i iteration(s): %g.\n", i-1, convergence); break;
            default: printf("Unkown exit criteria.\n");
        }
    }
    return return_closest_to_root_3pt(x_a, fx_a, x_b, fx_b, x_mean, fx_mean);
```

```
}

double find_root_bissection(double (*func)(double), double x_a, double x_b){
    return find_root_bissection_debug(func, x_a, x_b, default_convergence, true, default_iterations, false, -1e99);
}

int estimate_bissection_iterations(double x_a, double x_b, double x_root, double convergence_tol, bool relative_convergence){
    double x_reference = 1;
    double n;

    if( relative_convergence){
        if(fabs(x_root)<epsilon || x_root < min(x_a,x_b) || x_root > max(x_a, x_b)){
            x_reference = min(fabs(x_a), fabs(x_b));
            if( x_reference < epsilon)
                x_reference = max(fabs(x_a), fabs(x_b));
            if( x_reference < epsilon)
                return 0;
        } else{
            x_reference = x_root;
        }
    }
    n = log(fabs(x_b - x_a) / fabs(x_reference) / convergence_tol)/log(2);
    return imax(0, round(n+0.5));
}

void test_bissection(double (*func)(double), double x_root, double x_a, double x_b, double convergence_tol, bool relative_convergence, int max_iterations, bool debug, char *message){
    printf("\nTest Bissection Method: %s\n", message);
    int iterations = estimate_bissection_iterations(x_a, x_b, x_root, convergence_tol, false);
    printf("Estimated number of iterations: %i\n", iterations);
    double x_root_bissection = find_root_bissection_debug(func, x_a, x_b, convergence_tol, relative_convergence, max_iterations, debug, x_root);
    print_error(" => Root", x_root, x_root_bissection, convergence_tol, relative_convergence);
}
```

Apêndice C.2. Método de Newton-Raphson

```
double find_root_newton_raphson_debug(double (*func)(double), double (*func_prime)(double), double x_init, double x_min, double x_max, double convergence_tol, bool relative_convergence, int max_iterations, bool debug, double x_root, bool print_true_error) {
    double x_current, x_previous, x_best;
    double fx_current, fx_previous, fx_best;
    double fpx_current, fpx_previous;
    double convergence;
    enum type_exit_function {no_exit, root, converged, small_derivative};
```

```c
    enum type_exit_function exit_function = no_exit;
    int i=0;

    x_current = x_init;
    fx_current = func(x_current);
    fpx_current = func_prime(x_current);
    x_best = x_current;
    fx_best = fabs(fx_current);

    if( is_root(fx_current, x_current)){
        exit_function = root;
    }

    if( exit_function == no_exit){

        if( debug){
            if( print_true_error){
                printf("%3s  %-11s\t%-11s\t%-11s\t%-11s\t%-11s\n","#", "x", "f(x)", "f'(x)", "Convergence","|x - root|");
                printf("%3i  %11g\t%11g\t%11g\t%11s\t%11g\n",i, x_current, fx_current, fpx_current, "", fabs(x_current - x_root));
            } else{
                printf("%3s %-11s\t%-11s\t%-11s\t%-11s\n","#", "x", "f(x)", "f'(x)", "Convergence");
                printf("%3i  %11g\t%11g\t%11g\t%11s\n",i, x_current, fx_current, fpx_current, "");
            }
        }

        for(i=1 ; i<=max_iterations ; i++){
            x_previous = x_current;
            fx_previous = fx_current;
            fpx_previous = fpx_current;

            if( fabs(fpx_previous) < epsilon){
                if( debug){
                    printf("1st derivate is too small. Halted iterative process.");
                }
                exit_function = small_derivative;
                break;
            }
            x_current = x_previous - fx_previous / fpx_previous;

            if( x_current < x_min){
                if( debug){
                    printf("x is below lower limit: %g < %g.\n", x_current, x_min);
                }
                x_current = x_min;
            }
```

```
            if( x_current > x_max){
                if( debug){
                    printf("x is above upper limit: %g < %g.\n", x_current,
x_max);
                }
                x_current = x_max;
            }
            x_current = min(x_current, x_max);

            fx_current = func(x_current);
            fpx_current = func_prime(x_current);

            if( fabs(fx_current) < fx_best){
                x_best = x_current;
                fx_best = fabs(fx_current);
            }

            convergence = calculate_convergence(x_current, x_previous,
relative_convergence);

            if( debug){
                if( print_true_error){
                    printf("%3i  %11g\t%11g\t%11g\t%11g\t%11g\n",i, x_current
, fx_current, fpx_current, convergence, fabs(x_current - x_root));
                } else{
                    printf("%3i  %11g\t%11g\t%11g\t%11g\n",i, x_current,
fx_current, fpx_current, convergence);
                }
            }

            if( convergence < convergence_tol){
                exit_function = converged;
                break;
            }

            if( is_root(fx_current, x_current)){
                exit_function = root;
                break;
            }
        }
    }
    if( debug){
        switch(exit_function)
        {
            case root: printf("Found |f(x)| < %g after %i iterations.\n",
epsilon, i); break;
            case converged: printf("Reached convergence after %i iteration(s)
: %g < %g.\n", i, convergence, convergence_tol); break;
            case small_derivative: printf("Cannot continue. Returning best
result result after %i iterations.\n", i); break;
            case no_exit: printf("Convergence was not reached after %i
```

```
iteration(s): %g.\n", i-1, convergence); break;
            default: printf("Unkown exit criteria.\n");
        }
    }
    return x_best;
}

double find_root_newton_raphson(double (*func)(double), double (*func_prime)(
    double), double x_init){
    return find_root_newton_raphson_debug(func, func_prime, x_init, -1E99, 1
    E99, default_convergence, true, default_iterations, false, -1e99, false);
}

void test_newton_raphson(double (*func)(double), double (*func_prime)(double)
    , double x_root, double x_init, double x_min, double x_max, double
    convergence_tol, bool relative_convergence, int max_iterations, bool debug
    , char *message){
    printf("\nTest Newton-Raphson Method: %s\n", message);
    double x_root_NR = find_root_newton_raphson_debug(func, func_prime,
    x_init, x_min, x_max, convergence_tol, relative_convergence,
    max_iterations, debug, x_root, true);
    print_error(" => Root", x_root, x_root_NR, convergence_tol,
    relative_convergence);
}

// Root at x=0.3
double f_linear(double x){
    return -x*3 + 0.9;
}
double fp_linear(double x){
    return -3;
}

// Root at x=0.3
double f_linear2(double x){
    return -x*3E5 + 0.9E5;
}
double fp_linear2(double x){
    return -3E5;
}

// Roots at x=-0.5 and -0.1
double f_quadratic(double x){
    return (5*x + 3)*x + 0.25; // = 5*x*x + 3*x + 0.25;
}
double fp_quadratic(double x){
    return 10*x + 3;
}

// Root at x=0.0
double f_x2(double x){
```

```
    return 3*x*(3*x + 4);
}
double fp_x2(double x){
    return 6*(3*x + 2);
}

// Root at x=0.0
double f_x3(double x){
    return 3*x*x*(x + 2);
}
double fp_x3(double x){
    return 3*x*(3*x + 4);
}

// Root at x= +-2
double f_exponential(double x){
    return exp(x*x-4) - 1;
}
double fp_exponential(double x){
    return 2*x*exp(x*x-4);
}

// Root at x= pi/2 (+ n*2pi)
double f_cos(double x){
    return cos(x);
}
double fp_cos(double x){
    return -sin(x);
}

// Root at x= 3/4*pi (+ n*2pi)
double f_trigonometric(double x){
    return cos(x) + sin(x);
}
double fp_trigonometric(double x){
    return -sin(x) + cos(x);
}

// Root at x= 0
double f_trigonometric2(double x){
    return 3*x + x*x*sin(x*pi/10)*sin(x*pi/10);
}
double fp_trigonometric2(double x){
    return 3 + pi*x*x/5*cos(x*pi/10)*sin(x*pi/10) + 2*x*pi*sin(x*pi/10)*sin(x
   *pi/10);
}

void tests_newton_raphson(){
    bool relative_convergence = false;
    int max_iterations = 50;
    bool debug = true;
```

```

 test_newton_raphson(f_linear, fp_linear, 0.3, 0, -1E99, 1E99, 0.001, relative_convergence, max_iterations, debug, "Linear function");
 test_newton_raphson(f_linear, fp_linear, 0.3, 0, -1E99, 1E99, 0.001, true, max_iterations, debug, "Linear function with relative convergence");
 test_newton_raphson(f_linear, fp_linear, 0.3, 0.3, -1E99, 1E99, 0.001, relative_convergence, max_iterations, debug, "Linear function with root in x_init");

 test_newton_raphson(f_linear2, fp_linear2, 0.3, 0.3-epsilon/10, -1E99, 1E99, 0.001, relative_convergence, max_iterations, debug, "Linear function with x_init very close to root");

 test_newton_raphson(f_quadratic, fp_quadratic, -0.1, 0.25, -1E99, 1E99, 0.001, relative_convergence, max_iterations, debug, "Quadratic function, root#1");
 test_newton_raphson(f_quadratic, fp_quadratic, -0.5, -10., -1E99, 1E99, 0.001, relative_convergence, max_iterations, debug, "Quadratic function, root#2");
 test_newton_raphson(f_quadratic, fp_quadratic, -0.5, -0.3, -1E99, 1E99, 0.001, relative_convergence, max_iterations, debug, "Quadratic function, x_init = minimum");

 test_newton_raphson(f_x2, fp_x2, 0.0, 10., -1E99, 1E99, 0.001, relative_convergence, max_iterations, debug, "2nd deegre polinomial with minimum = root");
 test_newton_raphson(f_x3, fp_x3, 0.0, 10., -1E99, 1E99, 0.001, relative_convergence, max_iterations, debug, "3rd deegre polinomial with f'(root)=0");

 test_newton_raphson(f_exponential, fp_exponential, 2., 5, -1E99, 1E99, 0.001, relative_convergence, max_iterations, debug, "Exponential function, root #1");
 test_newton_raphson(f_exponential, fp_exponential, 2., 1., -1E99, 1E99, 0.001, relative_convergence, max_iterations, debug, "Exponential function, root #1");
 test_newton_raphson(f_exponential, fp_exponential, -2., -5, -1E99, 1E99, 0.001, relative_convergence, max_iterations, debug, "Exponential function, root #2");
 test_newton_raphson(f_exponential, fp_exponential, 2, 0, -1E99, 1E99, 0.001, relative_convergence, max_iterations, debug, "Exponential function, x_init = minimum");

 test_newton_raphson(f_cos, fp_cos, pi/2, 10*epsilon, -1E99, 1E99, 0.001, relative_convergence, max_iterations, debug, "Cossine function, x_init close to maximum");
 test_newton_raphson(f_cos, fp_cos, pi/2, 0, -1E99, 1E99, 0.001, relative_convergence, max_iterations, debug, "Cossine function, x_init = maximum");

 test_newton_raphson(f_trigonometric, fp_trigonometric, 3./4*pi, 3., -1E99
```

```
    , 1E99, 0.001, relative_convergence, max_iterations, debug, "Trigonometric
     function with multiple roots");

    test_newton_raphson(f_trigonometric2, fp_trigonometric2, 0, 10., -1E99, 1
    E99, 0.001, relative_convergence, max_iterations, debug, "Trigonometric
    function with multiple minima and 'very good' x_init");
    test_newton_raphson(f_trigonometric2, fp_trigonometric2, 0, 12, -1E99, 1
    E99, 0.001, relative_convergence, max_iterations, debug, "Trigonometric
    function with multiple minima and 'bad' x_init");
    test_newton_raphson(f_trigonometric2, fp_trigonometric2, 0, 9, -1E99, 1
    E99, 0.001, relative_convergence, max_iterations, debug, "Trigonometric
    function with multiple minima and 'good' x_init");
}
```

Apêndice C.3. Método da Mínima Curvatura

```
// Minimum Curvature Method
double angle_subtraction(double a2, double a1){
    double dif = a2 - a1;
    if( dif < -pi){
        dif = dif + 2*pi;
    } else if( dif > pi){
        dif = dif - 2*pi;
    }
    return dif;
}

double alfa(double theta1, double phi1, double theta2, double phi2){
    double x, y;
    x = sin( angle_subtraction(theta2, theta1)/2 );
    x *= x;
    y = sin( angle_subtraction(phi2, phi1)/2 );
    y *= y;
    y *= sin(theta1)*sin(theta2);
    x += y;
    x = sqrt(x);
    return 2* asin(x);
}

double f_alfa(double a){
    if( a<0.02){
        double x;
        double a2 = a*a;
        x = 1 + 32*a2/18;
        x = 1 + a2/168*x;
        x = 1+ a2/10*x;
        return 1+ a2/12*x;
    } else{
        return 2/a * tan(a/2);
    }
}

```

```
double deltaN_with_deltaSfa(double deltaSfa, double theta1, double phi1, double theta2, double phi2){
    return deltaSfa/2*(sin(theta1)*cos(phi1) + sin(theta2)*cos(phi2));
}

double deltaN_with_fa(double deltaS, double fa, double theta1, double phi1, double theta2, double phi2){
    return deltaN_with_deltaSfa(deltaS*fa, theta1, phi1, theta2, phi2);
}

double deltaN(double deltaS, double theta1, double phi1, double theta2, double phi2){
    double fa = f_alfa( alfa(theta1, phi1, theta2, phi2) );
    return deltaN_with_fa(deltaS, fa, theta1, phi1, theta2, phi2);
}

double deltaE_with_deltaSfa(double deltaSfa, double theta1, double phi1, double theta2, double phi2){
    return deltaSfa/2*(sin(theta1)*sin(phi1) + sin(theta2)*sin(phi2));
}

double deltaE_with_fa(double deltaS, double fa, double theta1, double phi1, double theta2, double phi2){
    return deltaE_with_deltaSfa(deltaS*fa, theta1, phi1, theta2, phi2);
}

double deltaE(double deltaS, double theta1, double phi1, double theta2, double phi2){
    double fa = f_alfa( alfa(theta1, phi1, theta2, phi2) );
    return deltaE_with_fa(deltaS, fa, theta1, phi1, theta2, phi2);
}


double deltaV_with_deltaSfa(double deltaSfa, double theta1, double theta2){
    return deltaSfa/2*(cos(theta1) + cos(theta2));
}

double deltaV_with_fa(double deltaS, double fa, double theta1, double theta2){
    return deltaV_with_deltaSfa(deltaS*fa, theta1, theta2);
}

double deltaV(double deltaS, double theta1, double phi1, double theta2, double phi2){
    double fa = f_alfa( alfa(theta1, phi1, theta2, phi2) );
    return deltaV_with_fa(deltaS, fa, theta1, theta2);
}

struct tangle{
    double cos, sin, rad;
};
```

```

double calculate_rad(struct tangle angle, bool debug){
    check_error( fabs(angle.cos) <= 1., "## Failed |cos(angle)| <= 1.\n");
    check_error( fabs(angle.sin) <= 1., "## Failed |sin(angle)| <= 1.\n");

    angle.cos = min(1,max(angle.cos,-1));
    angle.sin = min(1,max(angle.sin,-1));

    double a_cos = acos(angle.cos);
    double a_sin = asin(angle.sin);

    if( angle.sin < 0){
        a_cos = 2*pi - a_cos;
        if( angle.cos > 0){
            a_sin = 2*pi + a_sin;
        }
    }
    if( angle.cos < 0){
        a_sin = pi - a_sin;
    }

    double a = (a_sin + a_cos)/2;
    if( debug){
        double convergence = calculate_convergence(a, a_cos, true);
        printf("Convergence error in angle calculation (%4g.pi rad): %g\n", a/pi, convergence);
    }
    return a;
}

struct tangle calculate_theta2(double deltaV, double deltaSfa, double cos_theta1){
    struct tangle theta2;
    theta2.cos = 2*deltaV / deltaSfa - cos_theta1;
    if( check_error( fabs(theta2.cos) <= 1., "## Failed |cos(theta2)| <= 1.\n")){
        theta2.sin = sqrt(1 - theta2.cos * theta2.cos);
    } else{
        theta2.sin = 0;
    }
    return theta2;
}

struct tangle calculate_theta2_prime(double deltaV, double deltaSfa, double cos_theta1){
    struct tangle theta2;
    double s1 = - 2 * deltaV / deltaSfa;
    theta2.cos = s1 / deltaSfa;
    double s2 = (cos_theta1 + s1);
    check_error( s2*s2 <= 1., "## Failed cos(theta2)' calculation.\n");
    theta2.sin = s1 / deltaSfa * s2 / sqrt(1-s2*s2);
```

```
    return theta2;
}

struct tangle calculate_phi2_deltaE_zero(double deltaN, double sin_theta1, double cos_phi1, double sin_phi1, double sin_theta2){
    struct tangle phi2;
    if( fabs(sin_theta2)<epsilon && fabs(sin_theta1)<epsilon){
        phi2.sin = -sin_phi1;
    } else{
        if( check_error( fabs(sin_theta2) > epsilon, "## Failed sin(theta2) !=0.\n")){
            phi2.sin = - sin_theta1 / sin_theta2 * sin_phi1;
        } else{
            phi2.sin = -sin_phi1;
        }
    }
    if( check_error( fabs(phi2.sin) <= 1, "## Failed |sin(phi2)| <= 1.\n")){
        phi2.cos = sqrt(1 - phi2.sin*phi2.sin);
        if( cos_phi1 < 0){
            phi2.cos = -phi2.cos;
        }
    } else{
        phi2.cos = 0;
    }
    return phi2;
}

struct tangle calculate_phi2_deltaE_zero_prime(double deltaN, double sin_theta1, double cos_phi1, double sin_phi1, double sin_theta2, double sin_theta2_prime){
    struct tangle phi2;
    struct tangle phi2_no_prime;

    phi2_no_prime = calculate_phi2_deltaE_zero(deltaN, sin_theta1, cos_phi1, sin_phi1, sin_theta2);
    if( fabs(sin_theta2)<epsilon && fabs(sin_theta1)<epsilon){
        phi2.sin = 0;
    } else{
        if( check_error( fabs(sin_theta2) > epsilon, "## Failed sin(theta2) !=0.\n")){
            phi2.sin = - sin_theta1 / sin_theta2 * sin_phi1 * sin_theta2_prime / sin_theta2;
        } else{
            phi2.sin = 0;
        }
    }
    if( check_error( fabs(phi2.sin) <= 1, "## Failed |sin(phi2)| <= 1.\n")){
        phi2.cos = sqrt(1 - phi2.sin*phi2.sin);
        if( fabs(phi2_no_prime.sin) < epsilon){
            phi2.cos = 0;
        } else{
```

```
            phi2.cos = - phi2.sin * phi2_no_prime.sin / sqrt(1 - phi2_no_prime.sin*phi2_no_prime.sin);
        }
        if( cos_phi1 < 0){
            phi2.cos = -phi2.cos;
        }
    } else{
        phi2.cos = 0;
    }
    return phi2;
}

struct tangle calculate_phi2_deltaN_zero(double deltaE, double sin_theta1, double cos_phi1, double sin_phi1, double sin_theta2){
    struct tangle phi2;
    if( fabs(sin_theta2)<epsilon && fabs(sin_theta1)<epsilon){
        phi2.cos = -cos_phi1;
    } else{
        check_error( fabs(sin_theta2) > epsilon, "## Failed sin(theta2) != 0.\n");
        phi2.cos = - sin_theta1 / sin_theta2 * cos_phi1;
    }
    phi2.sin = sqrt(1 - phi2.cos*phi2.cos);
    if( sin_phi1 > 0){
        phi2.sin = -phi2.sin;
    }
    return phi2;
}

struct tangle calculate_phi2_deltaN_zero_prime(double deltaE, double sin_theta1, double cos_phi1, double sin_phi1, double sin_theta2, double sin_theta2_prime){
    struct tangle phi2;
    struct tangle phi2_no_prime;

    phi2_no_prime = calculate_phi2_deltaN_zero(deltaE, sin_theta1, cos_phi1, sin_phi1, sin_theta2);
    if( fabs(sin_theta2)<epsilon && fabs(sin_theta1)<epsilon){
        phi2.cos = 0;
    } else{
        check_error( fabs(sin_theta2) > epsilon, "## Failed sin(theta2) != 0.\n");
        phi2.cos = - sin_theta1 / sin_theta2 * cos_phi1 * sin_theta2_prime / sin_theta2;
    }
    check_error( fabs(phi2_no_prime.cos) <= 1, "## Failed |cos(phi2)| <= 1.\n");
    if( fabs(phi2_no_prime.cos) < epsilon){
        phi2.sin = 0;
    } else{
        phi2.sin = - phi2.cos * phi2_no_prime.cos / sqrt(1 - phi2_no_prime.
```

```
    cos*phi2_no_prime.cos);
    }
    if( sin_phi1 > 0){
        phi2.sin = -phi2.sin;
    }
    return phi2;
}

struct tangle calculate_phi2(double deltaE, double deltaN, double sin_theta1, double cos_phi1, double sin_phi1, double sin_theta2){
    struct tangle phi2;

    if( fabs(sin_theta2) < epsilon){
        phi2.cos = cos_phi1;
        phi2.sin = sin_phi1;
    } else if( fabs(deltaE) < epsilon && fabs(deltaN) < epsilon){
        phi2.cos = -cos_phi1;
        phi2.sin = -sin_phi1;
    } else if( fabs(deltaE) < epsilon){
        phi2 = calculate_phi2_deltaE_zero(deltaN, sin_theta1, cos_phi1, sin_phi1, sin_theta2);
    } else if( fabs(deltaN) < epsilon){
        phi2 = calculate_phi2_deltaN_zero(deltaE, sin_theta1, cos_phi1, sin_phi1, sin_theta2);
     } else{
        double deltaEpsilon = deltaN * sin_phi1 - deltaE * cos_phi1;
        double sin_theta1_sin_theta2 = sin_theta1 / sin_theta2;
        double deltaH2 = deltaE*deltaE + deltaN*deltaN;
        double deltaBeta2 = deltaH2 - deltaEpsilon * deltaEpsilon * sin_theta1_sin_theta2 * sin_theta1_sin_theta2;

        check_error( deltaBeta2 >= 0, "## Failed deltaBeta2 >= 0.\n");
        deltaBeta2 = max(0, deltaBeta2);
        double deltaBeta = sqrt(deltaBeta2);

        phi2.sin = (-deltaN * deltaEpsilon * sin_theta1_sin_theta2 + deltaE * deltaBeta) / deltaH2;
        phi2.cos = ( deltaE * deltaEpsilon * sin_theta1_sin_theta2 + deltaN * deltaBeta) / deltaH2;
    }
    return phi2;
}

struct tangle calculate_phi2_prime(double deltaE, double deltaN, double sin_theta1, double cos_phi1, double sin_phi1, double sin_theta2, double sin_theta2_prime){
    struct tangle phi2;

    if( fabs(sin_theta2) < epsilon){
        phi2.cos = 0;
        phi2.sin = 0;
```

```
    } else if( fabs(deltaE) < epsilon && fabs(deltaN) < epsilon){
        phi2.cos = 0;
        phi2.sin = 0;
    } else if( fabs(deltaE) < epsilon){
        phi2 = calculate_phi2_deltaE_zero_prime(deltaN, sin_theta1, cos_phi1,
     sin_phi1, sin_theta2, sin_theta2_prime);
    } else if( fabs(deltaN) < epsilon){
        phi2 = calculate_phi2_deltaN_zero_prime(deltaE, sin_theta1, cos_phi1,
     sin_phi1, sin_theta2, sin_theta2_prime);
     } else{
        double deltaEpsilon = deltaN * sin_phi1 - deltaE * cos_phi1;
        double sin_theta1_sin_theta2 = sin_theta1 / sin_theta2;
        double deltaH2 = deltaE*deltaE + deltaN*deltaN;
        double deltaBeta2 = deltaH2 - deltaEpsilon * deltaEpsilon *
    sin_theta1_sin_theta2 * sin_theta1_sin_theta2;

        check_error( deltaBeta2 >= 0, "## Failed deltaBeta2 >= 0.\n");
        deltaBeta2 = max(0, deltaBeta2);
        double deltaBeta = sqrt(deltaBeta2);

        double sin_theta1_sin_theta2_prime = - sin_theta1_sin_theta2 /
    sin_theta2 * sin_theta2_prime;

        phi2.sin = -(deltaN + deltaE * deltaEpsilon * sin_theta1_sin_theta2 /
     deltaBeta) * deltaEpsilon / deltaH2 * sin_theta1_sin_theta2_prime;
        phi2.cos = -(deltaE - deltaN * deltaEpsilon * sin_theta1_sin_theta2 /
     deltaBeta) * deltaEpsilon / deltaH2 * sin_theta1_sin_theta2_prime;
    }
    return phi2;
}

double calculate_deltaSfa(double deltaE, double deltaN, double deltaV, double
     cos_theta1, double sin_theta1, double cos_phi1, double sin_phi1, double
    cos_theta2, double sin_theta2, double cos_phi2, double sin_phi2){
    double aE = sin_theta1 * sin_phi1 + sin_theta2 * sin_phi2;
    double aN = sin_theta1 * cos_phi1 + sin_theta2 * cos_phi2;
    double aV = cos_theta1 + cos_theta2;
    return 2 * sqrt( (deltaE*deltaE + deltaN*deltaN + deltaV*deltaV) / (aE*aE
     + aN*aN + aV*aV) );
}

double calculate_deltaSfa_prime(double deltaE, double deltaN, double deltaV,
    double cos_theta1, double sin_theta1, double cos_phi1, double sin_phi1,
    double cos_theta2, double sin_theta2, double cos_phi2, double sin_phi2,
    double cos_theta2_prime, double sin_theta2_prime, double cos_phi2_prime,
    double sin_phi2_prime){
    double aE = sin_theta1 * sin_phi1 + sin_theta2 * sin_phi2;
    double aN = sin_theta1 * cos_phi1 + sin_theta2 * cos_phi2;
    double aV = cos_theta1 + cos_theta2;

    double aE_prime = sin_theta2 * sin_phi2_prime + sin_theta2_prime *
```

```
    sin_phi2;
    double aN_prime = sin_theta2 * cos_phi2_prime + sin_theta2_prime * cos_phi2;
    double aV_prime = cos_theta2_prime;

    double dS2 = deltaE*deltaE + deltaN*deltaN + deltaV*deltaV;
    double dA2 = aE*aE + aN*aN + aV*aV;
    return - 2 * pow(dS2/dA2, 3/2) / dS2 * (aE*aE_prime + aN*aN_prime + aV*aV_prime);
}

double calculate_deltaSfa_aproximate(double deltaE, double deltaN, double deltaV, double cos_theta1, double sin_theta1, double cos_phi1, double sin_phi1, double deltaSfa){
    struct tangle theta2 = calculate_theta2(deltaV, deltaSfa, cos_theta1);
    struct tangle phi2 = calculate_phi2(deltaE, deltaN, sin_theta1, cos_phi1, sin_phi1, theta2.sin);
    return calculate_deltaSfa(deltaE, deltaN, deltaV, cos_theta1, sin_theta1, cos_phi1, sin_phi1, theta2.cos, theta2.sin, phi2.cos, phi2.sin);
}

double calculate_deltaSfa_aproximate_prime(double deltaE, double deltaN, double deltaV, double cos_theta1, double sin_theta1, double cos_phi1, double sin_phi1, double deltaSfa){
    struct tangle theta2 = calculate_theta2(deltaV, deltaSfa, cos_theta1);
    struct tangle theta2_prime = calculate_theta2_prime(deltaV, deltaSfa, cos_theta1);
    struct tangle phi2 = calculate_phi2(deltaE, deltaN, sin_theta1, cos_phi1, sin_phi1, theta2.sin);
    struct tangle phi2_prime = calculate_phi2_prime(deltaE, deltaN, sin_theta1, cos_phi1, sin_phi1, theta2.sin, theta2_prime.sin);
    return calculate_deltaSfa_prime(deltaE, deltaN, deltaV, cos_theta1, sin_theta1, cos_phi1, sin_phi1, theta2.cos, theta2.sin, phi2.cos, phi2.sin, theta2_prime.cos, theta2_prime.sin, phi2_prime.cos, phi2_prime.sin);
}

double calculate_deltaSfa_error(double deltaE, double deltaN, double deltaV, double cos_theta1, double sin_theta1, double cos_phi1, double sin_phi1, double deltaSfa){
    double deltaSfa_calc = calculate_deltaSfa_aproximate( deltaE, deltaN, deltaV, cos_theta1, sin_theta1, cos_phi1, sin_phi1, deltaSfa);
    return deltaSfa_calc - deltaSfa;
}

double calculate_deltaSfa_error_prime(double deltaE, double deltaN, double deltaV, double cos_theta1, double sin_theta1, double cos_phi1, double sin_phi1, double deltaSfa){
    double deltaSfa_calc_prime = calculate_deltaSfa_aproximate_prime( deltaE, deltaN, deltaV, cos_theta1, sin_theta1, cos_phi1, sin_phi1, deltaSfa);
    return deltaSfa_calc_prime - 1;
}
```

```
double dE, dN, dV, cos_theta1, sin_theta1, cos_phi1, sin_phi1;
void define_well_path_data(double deltaE, double deltaN, double deltaV, double theta1, double phi1){
    dE = deltaE;
    dN = deltaN;
    dV = deltaV;
    dE = deltaE;
    cos_theta1 = cos(theta1);
    sin_theta1 = sin(theta1);
    cos_phi1 = cos(phi1);
    sin_phi1 = sin(phi1);
}
double calculate_defined_deltaSfa_error(double deltaSfa){
    return calculate_deltaSfa_error(dE, dN, dV, cos_theta1, sin_theta1, cos_phi1, sin_phi1, deltaSfa);
}
double calculate_defined_deltaSfa_error_prime(double deltaSfa){
    return calculate_deltaSfa_error_prime(dE, dN, dV, cos_theta1, sin_theta1, cos_phi1, sin_phi1, deltaSfa);
}

void test_MCM_formulas(char *message, double deltaS, double theta1, double phi1, double theta2, double phi2, double true_alfa, double true_deltaE, double true_deltaN, double true_deltaV, bool report_alfa_dEdNdV, bool relative_convergence, double convergence_limit){

    printf("\n%s\n", message);

    double a = alfa(theta1, phi1, theta2, phi2);
    double dE = deltaE(deltaS, theta1, phi1, theta2, phi2);
    double dN = deltaN(deltaS, theta1, phi1, theta2, phi2);
    double dV = deltaV(deltaS, theta1, phi1, theta2, phi2);

    if( report_alfa_dEdNdV){
        print_error("  Alfa",true_alfa,a, convergence_limit, relative_convergence);
        print_error("  Delta E", true_deltaE,dE, convergence_limit, relative_convergence);
        print_error("  Delta N", true_deltaN,dN, convergence_limit, relative_convergence);
        print_error("  Delta V", true_deltaV,dV, convergence_limit, relative_convergence);
    } else{
        printf("  Calculated displacement: dE=%g dN=%g dV=%g\n", dE, dN, dV);
        printf("  Calculated alfa=%g\n",a);
        printf("  Calculated f(alfa)=%g\n",f_alfa(a));
    }

    double deltaSfa = deltaS * f_alfa(a);
    struct tangle theta2_ = calculate_theta2(dV, deltaSfa, cos(theta1));
```

```
    print_error("  theta2", theta2, calculate_rad(theta2_, false), convergence_limit, relative_convergence);

    struct tangle phi2_ = calculate_phi2(dE, dN, sin(theta1), cos(phi1), sin(phi1), sin(theta2));
    print_error("  phi2", phi2, calculate_rad(phi2_, false), convergence_limit, relative_convergence);

    double deltaSfa_calc = calculate_deltaSfa(dE, dN, dV, cos(theta1), sin(theta1), cos(phi1), sin(phi1), cos(theta2), sin(theta2), cos(phi2), sin(phi2));
    double dS = deltaSfa_calc / f_alfa(a);
    print_error("  Delta S", deltaS, dS, convergence_limit, relative_convergence);

    printf("Calculate theta2, phi2 and dS from dE, dN, dV, theta1 and phi1.\n");
    define_well_path_data( dE, dN, dV, theta1, phi1);
    double deltaSfa_min = sqrt(dE*dE + dN*dN + dV*dV);
    double deltaSfa_max = deltaSfa_min * f_alfa(0.95*pi);
    if( dV > 0){
        deltaSfa_min = max(deltaSfa_min, 2*dV/(cos_theta1+1) );
    } else if( dV < 0){
        deltaSfa_min = max(deltaSfa_min, 2*dV/(cos_theta1-1) );
    }

    deltaSfa_calc = find_root_newton_raphson_debug(calculate_defined_deltaSfa_error, calculate_defined_deltaSfa_error_prime, deltaSfa_min*1.1, deltaSfa_min, deltaSfa_max, convergence_limit, relative_convergence, 100, true, deltaSfa, true);

    print_error("  Delta S x f(alfa)", deltaSfa, deltaSfa_calc, convergence_limit, relative_convergence);
    theta2_ = calculate_theta2(dV, deltaSfa_calc, cos(theta1));
    print_error("  theta2", theta2, calculate_rad(theta2_, false), convergence_limit, relative_convergence);
    phi2_ = calculate_phi2(dE, dN, sin(theta1), cos(phi1), sin(phi1), theta2_.sin);
    print_error("  phi2", phi2, calculate_rad(phi2_, false), convergence_limit, relative_convergence);
    a = alfa(theta1, phi1, calculate_rad(theta2_, false), calculate_rad(phi2_, false));
    dS = deltaSfa_calc / f_alfa(a);
    print_error("  Delta S", deltaS, dS, convergence_limit, relative_convergence);
}

void tests_minimum_curvature(){
    double theta1, phi1, theta2, phi2;
    double a;
    double dS, dE, dN, dV;
```

```
    char message[100];

    bool relative_convergence = false;
    double convergence_limit = 0.001;

    printf("\n#### Minimum Curvature Method Tests ####\n");

    dS=10;
    theta1=0;    phi1=0.88*pi;
    theta2=0;    phi2=0.88*pi;
    a=0.;
    dE=0., dN=0., dV=dS;
    test_MCM_formulas("Vertical well", dS, theta1, phi1, theta2, phi2, a, dE,
     dN, dV, true, relative_convergence, convergence_limit);

    dS=10;
    theta1=pi/4;    phi1=pi/6;
    theta2=pi/4;    phi2=pi/6;
    a=0.;
    dE=dS*sin(pi/4)*sin(pi/6);
    dN=dS*sin(pi/4)*cos(pi/6);
    dV=dS*cos(pi/4);
    test_MCM_formulas("Slant straight well", dS, theta1, phi1, theta2, phi2,
   a, dE, dN, dV, true, relative_convergence, convergence_limit);

    dS=10*pi/2;
    theta1=pi/2;    phi1=3*pi/2;
    theta2=pi/2;    phi2=0.;
    a=pi/2;
    dE=-10;    dN=10;    dV=0.;
    test_MCM_formulas("1/4 circle horizontal well", dS, theta1, phi1, theta2,
     phi2, a, dE, dN, dV, true, relative_convergence, convergence_limit);

    dS=10*pi/2;
    theta1=pi/2;    phi1=pi/4;
    theta2=pi/2;    phi2=7*pi/4.;
    a=pi/2;
    dE=0.;    dN=10*sqrt(2);    dV=0.;
    test_MCM_formulas("1/4 circle deltaE=0 horizontal well", dS, theta1, phi1
   , theta2, phi2, a, dE, dN, dV, true, relative_convergence,
   convergence_limit);

    dS=10*pi/2;
    theta1=0;        phi1=0.1871*pi;
    theta2=pi/2;     phi2=0.;
    a=pi/2;
    dE=0.;    dN=10.;    dV=10.;
    test_MCM_formulas("1/4 circle alligned north vertical to horizontal well"
   , dS, theta1, phi1, theta2, phi2, a, dE, dN, dV, true,
   relative_convergence, convergence_limit);

```

```
    dS=10*pi/2;
    theta1=pi/6;         phi1=0.;
    theta2=theta1+pi/2;  phi2=0.;
    a=pi/2;
    dE=0.;    dN=10.*(cos(pi/6)+sin(pi/6));    dV=10.*(cos(pi/6)-sin(pi/6));
    test_MCM_formulas("1/4 circle alligned north well 'going up'", dS, theta1
   , phi1, theta2, phi2, a, dE, dN, dV, true, relative_convergence,
   convergence_limit);

    dS=10*pi/2;
    theta1=pi/4;         phi1=0.;
    theta2=theta1+pi/2;  phi2=0.;
    a=pi/2;
    dE=0.;    dN=10.*(cos(theta1)+sin(theta1));    dV=10.*(cos(theta1)-sin(
   theta1));
    test_MCM_formulas("1/4 circle alligned north well 'going up' #2", dS,
   theta1, phi1, theta2, phi2, a, dE, dN, dV, true, relative_convergence,
   convergence_limit);

    dS=10*pi/2;
    theta1=pi/3;         phi1=0.;
    theta2=theta1+pi/2;  phi2=0.;
    a=pi/2;
    dE=0.;    dN=10.*(cos(theta1)+sin(theta1));    dV=10.*(cos(theta1)-sin(
   theta1));
    test_MCM_formulas("1/4 circle alligned north well 'going up' #3", dS,
   theta1, phi1, theta2, phi2, a, dE, dN, dV, true, relative_convergence,
   convergence_limit);

    dS=10*pi/2;
    theta1=0;        phi1=0.1871*pi;
    theta2=pi/2;     phi2=pi/6;
    a=pi/2;
    dE=10.*sin(pi/6);    dN=10.*cos(pi/6);    dV=10.;
    test_MCM_formulas("1/4 circle 30o north vertical to horizontal well", dS,
    theta1, phi1, theta2, phi2, a, dE, dN, dV, true, relative_convergence,
   convergence_limit);

    double array_dS[5]={1000., 500*pi/12, 1500., 1000*pi/12, 500.};
    double array_theta[6]={0., 0., pi/12, pi/12, 0., 0.};
    double array_phi[6]={0., 0., 0., 0., 0., 0.};
    double array_a[5]={0., pi/12, 0, pi/12, 0.};
    double array_dE[5]={0., 0., 0., 0., 0.};
    double array_dN[5]={0., 500.*(1-cos(pi/12)), 1500.*sin(pi/12), 1000.*(1-
   cos(pi/12)), 0.};
    double array_dV[5]={1000., 500.*sin(pi/12), 1500.*cos(pi/12), 1000.*sin(
   pi/12), 500.};
    printf("\n'S-shaped' well\n");
    for( int i=0; i<5; i++){
        sprintf(message, "Section #%i",i+1);
        test_MCM_formulas(message, array_dS[i], array_theta[i], array_phi[i],
```

```
    array_theta[i+1], array_phi[i+1], array_a[i], array_dE[i], array_dN[i],
    array_dV[i], true, relative_convergence, convergence_limit);
    }

    dS=10.;
    theta1=pi/10;       phi1=pi/4;
    theta2=pi/6;        phi2=7*pi/4;
    a=0.;
    dE=0.;    dN=0.;    dV=0.;

    double c;
    for( int i = 1; i<= 5; i++){
        c = pow(10, -i);
        sprintf(message, "'3D' well with convergence = %g", c);
        test_MCM_formulas(message, dS, theta1, phi1, theta2, phi2, a, dE, dN,
    dV, false, relative_convergence, c);
    }

}

void print_fx_fpx(double (*func)(double), double (*func_prime)(double), int
    points, double x_min, double x_max){
    for(int i=0; i<=points; i++){
        double x = x_min + (x_max - x_min) * i / points;
        printf("%g  %g  %g\n", x, func(x), func_prime(x));
    }
}

int main(){
    tests_newton_raphson();
    tests_minimum_curvature();
    return 0;
}
```