Proposta de Cálculo de Parâmetros de Perfuração de Poços de Petróleo a partir de Coordenadas Espaciais com o Método da Bissecção*

Tiago C. A. Amorim[2]

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

Abstract

O Método de Mínima Curvatura é reconhecido como o mais aceito no cálculo de trajetória de poços de petróleo. A formulação para cálculo de coordenadas cartesianas a partir de parâmetros de perfuração é direta, e o cálculo inverso não tem formulação direta.

Neste trabalho é proposto um algoritmo para calcular parâmetros de perfuração a partir de coordenadas cartesianas. O algoritmo proposto tem a forma $g(x) = x$, e encontrar uma solução passa por um problema de encontrar a raiz de uma função.

Foi aplicado o Método da Bissecção para resolver o problema proposto. O testes realizados mostraram boa coerência entre os valores estimados com o processo iterativo e as respectivas respostas exatas.

Keywords: Método da Mínima Curvatura, Método da Bissecção

1. Introdução

O desenvolvimento de técnicas para construção de poços direcionais na indústria do petróleo iniciou nos anos 1920 [1]. A construção de poços direcionais pode ter diferentes objetivos, desde acessar acumulações que seriam difíceis de serem alcançadas com poços verticais (áreas montanhosas, acumulações abaixo de leitos de rios etc.), para aumento da produtividade (maior exposição da formação portadora de hidrocarbonetos) ou até para interceptar outros poços (poços de alívio em situações de *blowout*[1]).

O Método da Mínima Curvatura é largamente aceito como o método padrão para o cálculo de trajetória de poços [2]. Neste método a geometria do poço é descrita como uma série de arcos circulares e linhas retas. A transformação de parâmetros de perfuração (ΔS, θ,

*Relatório integrante 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)

[1]Um *blowout* é um evento indesejado, de produção descontrolada de um poço.

ϕ) em coordenadas cartesianas (ΔN, ΔE, ΔV) tem formulação explícita. A operação inversa, de coordenadas cartesianas em parâmetros de perfuração não tem formulação explícita.

No planejamento de novos poços de petróleo as coordenadas espaciais são conhecidas, e é necessário calcular os futuros parâmetros de perfuração. Os parâmetros de perfuração são utilizados para diferentes análises, como o de máximo DLS (*dogleg severity*), que é uma medida da curvatura de um poço entre dois pontos de medida, usualmente expressa em graus por metro.

Este relatório apresenta uma proposta de metodologia para cálculo dos parâmetros de perfuração a partir das coordenadas cartesianas de pontos ao longo da geometria do poço. A formulação foi derivada das fórmulas utilizadas no Método da Mínima Curvatura, e é implícita. Para resolver o problema foi aplicado o Método da Bissecção.

2. Metodologia

2.1. Método da Mínima Curvatura

Ao longo da perfuração de um poço de petróleo são realizadas medições do comprimento perfurado (comumente chamado de comprimento medido), inclinação (ângulo com relação à direção vertical) e azimute (ângulo entre a direção horizontal e o norte). A partir das coordenadas geográficas do ponto inicial do poço e deste conjunto de medições ao longo da trajetória, é possível calcular as coordenadas cartesianas (N, E, V) de qualquer posição do poço. A figura 1 apresenta um esquema dos parâmetros de perfuração de um poço direcional:

- ΔS: comprimento medido entre dois pontos ao longo da trajetória.
- θ: inclinação do poço no ponto atual.
- ϕ: azimute do poço no ponto atual.
- α: curvatura entre dois pontos ao longo da trajetória.

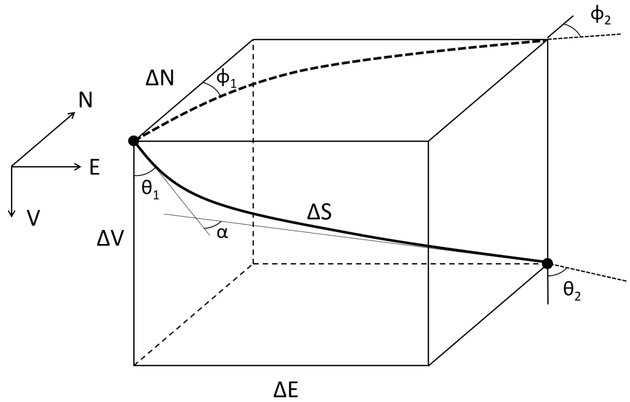

Figura 1: Parâmetros de perfuração entre dois pontos ao longo de um poço direcional.

As fórmulas que associam os parâmetros de perfuração e as coordenadas cartesianas de dois pontos ao longo de um poço direcional segundo o Método da Mínima Curvatura são [2]:

$$\Delta N = \frac{\Delta S}{2} f(\alpha)(\sin\theta_1 \cos\phi_1 + \sin\theta_2 \cos\phi_2) \tag{1}$$

$$\Delta E = \frac{\Delta S}{2} f(\alpha)(\sin\theta_1 \sin\phi_1 + \sin\theta_2 \sin\phi_2) \tag{2}$$

$$\Delta V = \frac{\Delta S}{2} f(\alpha)(\cos\theta_1 + \cos\theta_2) \tag{3}$$

$$\alpha = 2 \arcsin \sqrt{\sin^2 \frac{\theta_2 - \theta_1}{2} + \sin\theta_1 \sin\theta_2 \sin^2 \frac{\phi_2 - \phi_1}{2}} \tag{4}$$

$$f(\alpha) = \begin{cases} 1 + \frac{\alpha^2}{12}\{1 + \frac{\alpha^2}{10}[1 + \frac{\alpha^2}{168}(1 + \frac{31\alpha^2}{18})]\}, & \text{se } \alpha < 0,02 \\ \frac{2}{\alpha}\tan\frac{\alpha}{2}, & \text{c.c.} \end{cases} \tag{5}$$

A proposta de método para calcular os parâmetros de perfuração a partir das coordenadas cartesianas parte de manipulações das equações 1, 2 e 3. É assumido que para calcular os parâmtros de perfuração entre dois pontos quaisquer são conhecidos os parâmetros de perfuração do ponto inicial[2] (θ_1, ϕ_1) e as distâncias cartesianas entre os pontos (ΔN,ΔE, ΔV). O objetivo é calcular θ_1, ϕ_1 e ΔS.

É possível inverter a equação 3 para obter uma expressão para θ_2:

$$\cos\theta_2 = 2\frac{\Delta V}{\Delta S f(\alpha)} - \cos\theta_1 \tag{6}$$

Dividindo a equação 1 pela equação 2 obtém-se duas expressões para ϕ_2:

$$\sin\phi_2 = \frac{-\Delta N \Delta\Psi}{\Delta H^2}\left(\frac{\sin\theta_1}{\sin\theta_2}\right) + \Delta E \sqrt{\frac{1}{\Delta H^2} - \Delta\Psi^2 \left(\frac{\sin\theta_1}{\sin\theta_2}\right)^2} \tag{7}$$

$$\cos\phi_2 = \frac{\Delta E \Delta\Psi}{\Delta H^2}\left(\frac{\sin\theta_1}{\sin\theta_2}\right) + \Delta N \sqrt{\frac{1}{\Delta H^2} - \Delta\Psi^2 \left(\frac{\sin\theta_1}{\sin\theta_2}\right)^2} \tag{8}$$

onde

$$\begin{aligned} \Delta\Psi &= \Delta N \sin\phi_1 - \Delta E \cos\phi_1 \\ \Delta H^2 &= \Delta N^2 + \Delta E^2 \end{aligned}$$

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

Fazendo a soma dos quadrados das equações 1, 2 e 3 é possível obter uma expressão para $\Delta Sf(\alpha)$:

$$\Delta Sf(\alpha) = 2\sqrt{\frac{\Delta N^2 + \Delta E^2 + \Delta V^2}{A^2 + B^2 + C^2}} \tag{9}$$

onde

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

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

1. Assumir um valor inicial de $\Delta Sf(\alpha)$.
2. Calcular $\cos\theta_2$ com a equação 6.
3. Calcular $\sin\phi_2$ com a equação 7.
4. Calcular $\cos\phi_2$ com a equação 8.
5. Calcular $\Delta Sf(\alpha)$ com a equação 9.

Ao utilizar $\Delta Sf(\alpha)$ como parâmetro principal, evita-se calcular α e $f(\alpha)$ durante o processo. O valor de ϕ_2 só precisa ser calculado ao final, evitando usar arccos ou arcsin muitas vezes. Alguns cuidados adicionais precisam ser tomados ao utilizar este algoritmo:

- O valor mínimo de $\Delta Sf(\alpha)$ é uma linha reta entre os pontos:

$$\Delta Sf(\alpha) \geq \sqrt{\Delta N^2 + \Delta E^2 + \Delta V^2}$$

- $\Delta Sf(\alpha)$ tem um segundo limite inferior a ser atendido, definido pelos valores limite da equação 6 quando $\Delta V \neq 0$:

$$\Delta Sf(\alpha) \geq \begin{cases} \Delta V \frac{2}{\cos\theta_1 + 1}, & \text{se } \Delta V > 0 \\ \Delta V \frac{2}{\cos\theta_1 - 1}, & \text{se } \Delta V < 0 \end{cases}$$

- Se $\theta_2 = 0$, então ϕ_2 fica indefinido. Neste caso a recomendação é fazer $\phi_2 = \phi_1$.

- Se $\Delta N = \Delta E = 0$ então $|\phi_1 - \phi_2| = \pi$.

2.2. Método da Bissecção

Uma equação do tipo $g(x) = x$ pode ser resolvida buscando a raiz de $f(x) = x - g(x)$. Nesta primeira tentativa foi implementado o Método da Bissecção para buscar o resultado do problema proposto. O algoritmo foi baseado no pseudo-código descrito em [3]. O Método da Bissecção baseia-se no teorema do valor médio. O intervalo de busca pela raiz é sucessivamente divido em dois. O método tem garantia de que a raiz pertence ao intervalo ao manter os valores da função avaliada nos limites do intervalo com sinais opostos[3].

De modo simplificado o Método da Bissecção pode ser descrito como:

1. Definir x_a e x_b de modo que $sinal(f(x_a)) \neq sinal(f(x_b))$.
2. Calcular $f(x_a)$ e $f(x_b)$.
3. Calcular $x_{medio} = x_a + \frac{x_b - x_a}{2}$ e $f(x_{medio})$.
4. Se $sinal(f(x_{medio})) = sinal(f(x_a))$ então $x_a = x_{medio}$.
5. Se $sinal(f(x_{medio})) = sinal(f(x_b))$ então $x_b = x_{medio}$.
6. Se não atingiu critério de convergência, retornar para passo 2.
7. Retornar x_{medio}.

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

- Foram implementados dois métodos de cálculo do critério de convergência:
  - Critério *Direto*: $|x_i - x_{i-1}|$.
  - Critério *Relativo*[4]: $\frac{|x_i - x_{i-1}|}{|x_i|}$.
- No início do código é verificado se $|x_b - x_a| < \zeta$, onde ζ é calculado em função do critério de convergência estabelecido[5]. Se for *verdadeiro*, não é feito o *loop* do método.
- Se $sinal(f(x_a)) = sinal(f(x_b))$ o código apresenta uma mensagem de alerta e não é feito o *loop* do método. Optou-se por não gerar um erro, e mesmo neste caso é retornado um valor.
- 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*.
- Antes de sair da função, são comparados os três últimos resultados guardados ($f(x_a)$, $f(x_b)$, $f(x_{medio})$) e é retornado o valor de x com $f(x)$ mais próximo de zero.

---

[3] Assumindo que o intervalo inicial fornecido também tem esta propriedade.

[4] Caso $|x_i| < \epsilon$, é utilizado $|x_{i-1}|$ no denominador. E se também $|x_{i-1}| < \epsilon$ o valor da convergência é considerado *zero*!

[5] Definindo c_{lim} o limite de convergência, se for utilizado o critério de convergência *direto* então $\zeta = c_{lim}$. Se for utilizado o critério de convergência relativo então $\zeta = c_{lim} \min |x_a|, |x_b|$ (a avaliação do menor valor segue as mesmas regras do cálculo do critério de convergência, ignorando qualquer $|x| < \epsilon$ e retorna *zero* caso ambos sejam valores pequenos).

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_bissection() Testa o Método da Bissecção em diferentes funções: linear, quadrática, exponencial e com sen/cos. Também foram apicados casos específicos para o algoritmo tratar: raiz em um dos limites, uso de critério de convergência relativo, saída do *loop* sem atingir o critério de convergência, má definição do intervalo inicial ($sinal(f(x_a)) = sinal(f(x_b))$) e intervalo inicial muito pequeno ($|x_b - x_a| < \zeta$).

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).

Algumas definições que foram feitas no código que implementa o Método da Bissecção são resultado dos testes realizados.

A definição de um critério de parada prematura se mostrou importante para evitar que o método continue buscando uma raiz quando já encontrou uma solução *aceitável*. A definição deste limite $|f(x)| < \epsilon = 10^{-7}$ foi empírica e deve ser revista para problemas com valores usuais de $f(x)$ com ordem de grandeza diferente da que foi utilizada nos testes ($\approx 10^1$). Um exemplo é o da função $f(x) = -3x + 0.9$, que tem raiz em 0.3. A função implementada retorna $f(0.3) \approx 1.11022 \cdot 10^{-16}$, de modo que é preciso levar em conta erros de aritmética de máquina na definição do critério de parada prematura.

Todas as funções foram definidas para trabalhar com números do tipo `double`. Inicialmente as funções estavam definidas para trabalhar com `float`, mas estes mostraram não conseguirem trabalhar com valores de convergência muito baixos. A avaliação da função $f(x) = -3x + 0.9$ na sua raiz foi testada usando diferentes tipos de números de ponto flutante:

- `float`: $f(0.3) \approx -3.57628 \cdot 10^{-8}$
- `double`: $f(0.3) \approx 1.11022 \cdot 10^{-16}$
- `long double`: $f(0.3) \approx 5.35872 \cdot 10^{-312}$

Mesmo que o `long double` consiga o melhor resultado, considerou-se que trabalhar com `double` já é *suficiente*.

É sempre feita uma verificação ao final do código para avaliar qual é o valor entre x_a, x_b e x_{medio} que minimiza $|f(x)|$. O método da bisecção tem garantia de convergência, mas não é garantido que o melhor resultado será o x_{medio} da última iteração. Um exemplo prático deste efeito é visto na Tabela 3, onde o melhor resultado é alcançado na 8^a iteração, mas o critério de convergência[6] só é alcançado na 11^a iteração.

[6] A coluna com os valores da convergência foi omitida por falta de espaço na página

Tabela 1: Busca pela raiz de $5x^2 + 3x - 0.25$ no intervalo $[-0.25; 1]$ pelo Método da Bissecção.

Int.	x_a	$f(x_a)$	x_b	$f(x_b)$	x_{medio}	$f(x_{medio})$
1	-0.25	-0.1875	1	8.25	0.375	2.07812
2	-0.25	-0.1875	0.375	2.07812	0.0625	0.457031
3	-0.25	-0.1875	0.0625	0.457031	-0.09375	0.0126953
4	-0.25	-0.1875	-0.09375	0.0126953	-0.171875	-0.11792
5	-0.171875	-0.11792	-0.09375	0.0126953	-0.132812	-0.0602417
6	-0.132812	-0.0602417	-0.09375	0.0126953	-0.113281	-0.0256805
7	-0.113281	-0.0256805	-0.09375	0.0126953	-0.103516	-0.00696945
8	-0.103516	-0.00696945	-0.09375	0.0126953	-0.0986328	0.00274372
9	-0.103516	-0.00696945	-0.0986328	0.00274372	-0.101074	-0.00214267
10	-0.101074	-0.00214267	-0.0986328	0.00274372	-0.0998535	0.000293076
11	-0.101074	-0.00214267	-0.0998535	0.000293076	-0.100464	-0.000926659

A metodologia proposta para o cálculo de parâmetros de perfuração com o Método da Mínima Curvatura tem algumas particularidades, como a sua indefinição quando é testado com um valor muito baixo de $\Delta Sf(\alpha)$. A definição do intervalo de busca é direta, mas, devido a erros de aritmética de máquina, não é possível garantir que $sinal(f(x_a)) \neq sinal(f(x_b))$ quando a resposta está muito próxima de um dos limites. Desta forma, ao invés de gerar uma mensagem de erro quando $sinal(f(x_a)) = sinal(f(x_b))$, é gerada uma mensagem e o código retorna o valor de x (igual a x_a ou x_b) que minimizar $|f(x)|$.

Este problema ficou evidente no cálculo do 4^o trecho do poço em 'S' descrito nos testes de `tests_minimum_curvature()` (Figura Apêndice B). O valor exato de θ_2 é zero (o 5^0 trecho é reto e vertical). Neste caso, o valor mínimo de $\Delta Sf(\alpha)$ é limitado pela equação 6, e é igual à resposta exata. Neste ponto o algoritmo calcula $f(263.305) \approx -2.58273 \cdot 10^{-7}$, que não atinge o critério de parada prematura, mas tem o mesmo sinal de $f(\Delta Sf(\alpha)_{max})$.

Foi implementada uma função para estimar o número de iterações necessárias para alcançar o critério de convergência definido (equação 10). Quando o critério de convergência (c_{lim}) é *direto*, a estimativa se mostrou exata (exceto nos casos em que há parada prematura), o que indica que o algoritmo implantado converge com a taxa que era esperada (ver Figura Apêndice A). Buscou-se realizar a mesma estimativa para o caso de uso do critério de convergência *relativo*, e neste caso as estimativas não exatas, mas tem boa previsão (ver Tabela 3).

$$n_{int} = \left\lceil \frac{\log \frac{|x_b - x_a|}{|x_{referencia}|} \frac{1}{c_{lim}}}{\log 2} \right\rceil \quad (10)$$

onde

$$x_{referencia} = \begin{cases} 1, & \text{se Critério de convergência direto} \\ \min(|x_a|, |x_b|), & \text{c.c.} \end{cases}$$

Tabela 2: Comparação entre o número de iterações previsto e realizado para diferentes testes.

Função	x_a	x_b	Critério *Direto*		Critério *Relativo*	
			Previsão	Realizado	Previsão	Realizado
Linear	0.	2.	11	11	11	13
Quadrática	-0.25	1.	11	11	11	14
Exponencial	0.	10.	14	14	14	13
Trigonométrica	0.	5.	13	13	13	12
1/4 círculo horizontal	14.1421	120.417	17	17	13	13
Seção 2 do poço em S	130.526	1111.4	20	20	13	13
Poço 3D	9.84918	83.8634	17	17	13	13

4. Conclusão

O algoritmo proposto para o cálculo de parâmetros de perfuração em função das coordenadas cartesianas de pontos ao longo do se mostrou eficaz. Foram feitos ajustes ao código implementado de forma a evitar o uso de funções trigonométricas ao longo do processo iterativo. O Método da Bisseção se mostrou adequado para uso com o algoritmo proposto. As funções propostas não são válidas para quaisquer valores de entrada, e a delimitação de uma região de busca foi importante para garantir a convergência do problema.

Referências

[1] I. A. of Drilling Contractors (IADC), IADC Drilling Manual, International Association of Drilling Contractors (IADC), 2015, prévia do livro em `https://iadc.org/wp-content/uploads/2015/08/preview-dd.pdf` (accessado em 28/08/2023).

[2] 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`

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

Apêndice A. Gráficos Diagnóstico

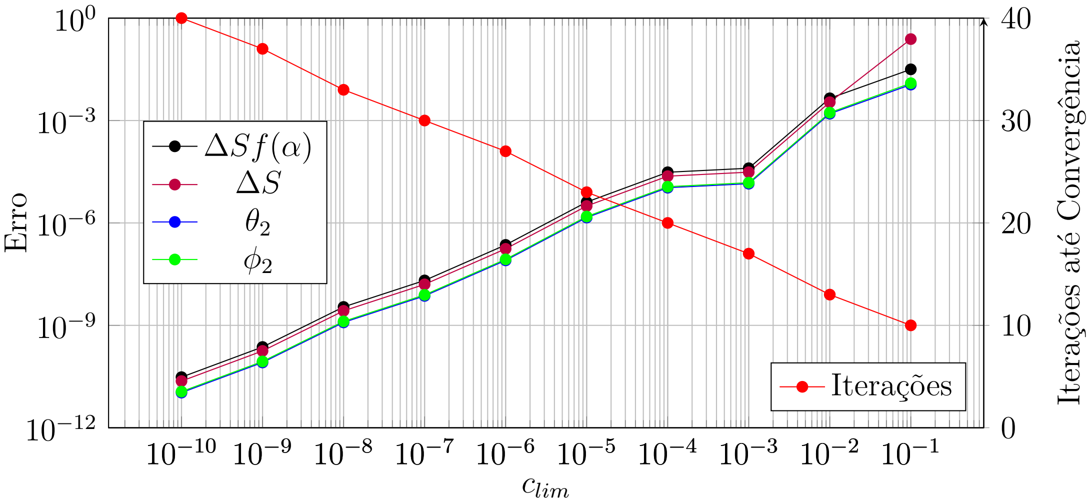

Figura A.2: Erro das variáveis de interesse em função do limite de convergência.

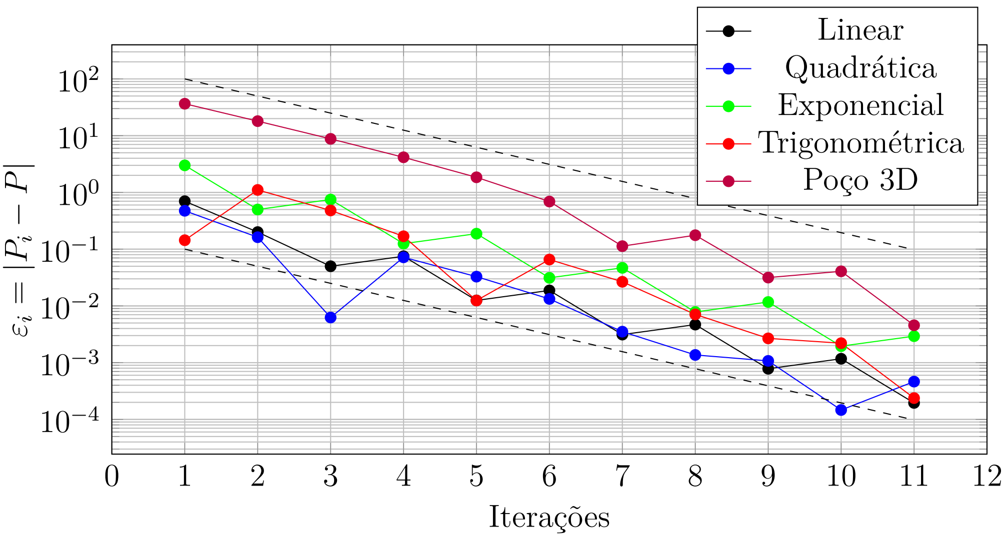

Figura A.3: Evolução do erro do Método da Bissecção com diferentes funções (linhas tracejadas representam $\varepsilon_{i+1}/\varepsilon_i = 0.5$).

Apêndice B. Esquema do Poço em S

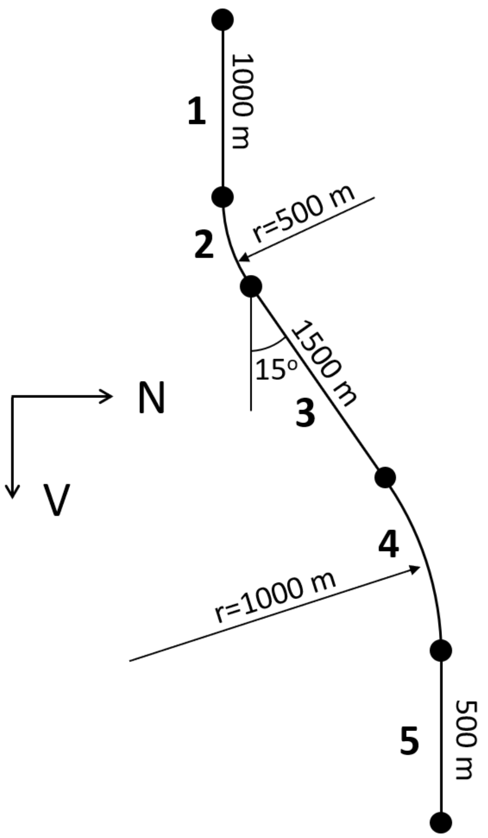

Figura B.4: Esquema do poço em 'S' descrito nos testes de tests_minimum_curvature().

Apêndice C. Código em C

```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-17;
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;
}

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, int i, bool debug){
    if( fabs(fx) < epsilon){
        if( debug){
            printf("Early exit after %i iterations: f(%g) = %g.\n", i, x, fx);
        }
        return true;
    }
    return false;
}

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("  <= ######### Atention ###########");
    }
    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;
    double exit_function = false;
    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);
    exit_function = is_root(fx_a, x_a, 0, debug) || is_root(fx_b, x_b, 0,
```

```
debug);

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

     if( !exit_function){
         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);
             printf("Returning result closest to zero amongst f(x_a) and f(x_b
).\n");
             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 = true;
         }
     }

     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;

```

```c
            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){
                break;
            }

            if( is_root(fx_mean, x_mean, i, debug)){
                exit_function = true;
                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){
        if( exit_function || convergence < convergence_tol){
            printf("Reached convergence after %i iteration(s): %g.\n", i, convergence);
        } else {
            printf("Convergence was not reached after %i iteration(s): %g.\n", i, convergence);
        }
    }
    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 max(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);
}

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

float f_linear_float(float x){
    return -x*3 + 0.9;
}

long double f_linear_long_double(long double x){
    return -x*3 + 0.9;
}

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

// 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;
}
```

```

// Roots at x=-0.5 and -0.1
double f_quadratic2(double x){
    return 5*x*x + 3*x + 0.25;
}

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

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

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

int tests_bissection(){
    bool relative_convergence = false;
    int max_iterations = 50;
    bool debug = true;

    test_bissection(f_linear, 0.3, 0, 2, 0.001, relative_convergence, max_iterations, debug, "Linear function");
    test_bissection(f_linear, 0.3, 2, 0, 0.001, relative_convergence, max_iterations, debug, "Linear function with inverted limits");
    test_bissection(f_linear, 0.3, 0, 2, 0.001, true, max_iterations, debug, "Linear function with relative convergence");
    test_bissection(f_linear, 0.3, 0, 1, 0.001, relative_convergence, 5, debug, "Linear function with insufficient iterations");
    test_bissection(f_linear, 0.3, 0., 0.4, 0.001, relative_convergence, max_iterations, debug, "Linear function with early exit");
    test_bissection(f_linear, 0.3, 0.3, 1, 0.001, relative_convergence, max_iterations, debug, "Linear function with root in x_a");
    test_bissection(f_linear, 0.3, 0, 0.3, 0.001, relative_convergence, max_iterations, debug, "Linear function with root in x_b");
    test_bissection(f_linear, 0.3, 1, 2.3, 0.001, relative_convergence, max_iterations, debug, "Linear function with error in [x_a,x_b] #1");
    test_bissection(f_linear, 0.3, -2, 0., 0.001, relative_convergence, max_iterations, debug, "Linear function with error in [x_a,x_b] #2");

    test_bissection(f_linear, 0.3, 0.3-epsilon/3, 0.3+epsilon/3, 0.001, relative_convergence, max_iterations, debug, "Linear function with domain very close to root");
    test_bissection(f_linear2, 0.3, 0.3-epsilon/3, 0.3+epsilon/3, 0.001, relative_convergence, max_iterations, debug, "Linear function #2 with 'small' domain");
```

```
    test_bissection(f_quadratic, -0.1, -0.25,  1, 0.001, relative_convergence
    , max_iterations, debug, "Quadratic function, root#1");
    test_bissection(f_quadratic, -0.5, -0.25, -1, 0.001, relative_convergence
    , max_iterations, debug, "Quadratic function, root#2");
    test_bissection(f_exponential, 2., 0, 10, 0.001, relative_convergence,
    max_iterations, debug, "Exponential function");
    test_bissection(f_cos, pi/2, 0, 2, 0.001, relative_convergence,
    max_iterations, debug, "Cossine function");
    test_bissection(f_trigonometric, 3./4*pi, 0, 5, 0.001,
    relative_convergence, max_iterations, debug, "Trigonometric function");
}


// 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;
};

```

```c
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_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_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.sin*phi2.sin);
    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 deltaEpsilon_sin_theta1 = deltaEpsilon * sin_theta1;
```

```
        double deltaH2 = deltaE*deltaE + deltaN*deltaN;
        double deltaBeta2 = deltaH2 * sin_theta2*sin_theta2 - deltaEpsilon_sin_theta1*deltaEpsilon_sin_theta1;

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

        double deltaH2_sin_theta2 = deltaH2 * sin_theta2;
        if( !check_error( fabs(deltaH2_sin_theta2) > epsilon, "## Failed deltaH2_sin_theta2 != 0.\n")){
            deltaH2_sin_theta2 = deltaH2*0.001;
        }

        phi2.sin = (-deltaN * deltaEpsilon_sin_theta1 + deltaE*deltaBeta) / deltaH2_sin_theta2;
        phi2.cos = ( deltaE * deltaEpsilon_sin_theta1 + deltaN*deltaBeta) / deltaH2_sin_theta2;
    }
    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_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_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 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);
}

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) );
    }

    int estimated_iterations = estimate_bissection_iterations(deltaSfa_min, deltaSfa_max, -9999, convergence_limit, relative_convergence);
    printf("Estimated number of iterations: %i\n", estimated_iterations);
    deltaSfa_calc = find_root_bissection_debug(calculate_defined_deltaSfa_error, deltaSfa_min, deltaSfa_max, convergence_limit, relative_convergence, 100, true, deltaSfa);

    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;
    struct tangle theta2_, phi2_;
    double theta2_calc, phi2_calc;
    double dSfa, dS_calc;
    char message[100];
```

```

    bool relative_convergence = false;
    bool convergence_limit = 0.001;

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

    printf("\nAngle calculation\n");
    for(int i=0; i<=10; i++){
        a = 0.2*i*pi;
        theta2_.cos = cos(a);
        theta2_.sin = sin(a);
        theta2_.rad = calculate_rad(theta2_, true);
        print_error(" Angle calculation",a,theta2_.rad, 0.001, false);
    }

    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<= 10; 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);
    }
}

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