

# 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<sup>2</sup>

<sup>a</sup>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*<sup>1</sup>).

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 ( $\Delta S, \theta$ ,

---

\*Relatório número 1 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](mailto:t100675@dac.unicamp.br) (Tiago C. A. Amorim)

<sup>1</sup>Um *blowout* é um evento indesejado, de produção descontrolada de um poço.

$\phi$ ) em coordenadas cartesianas ( $\Delta N$ ,  $\Delta E$ ,  $\Delta V$ ) 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 30 metros<sup>2</sup>.

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 azimuth (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:

- $\Delta S$ : comprimento medido entre dois pontos ao longo da trajetória.
- $\theta$ : inclinação do poço no ponto atual.
- $\phi$ : azimuth do poço no ponto atual.
- $\alpha$ : curvatura entre dois pontos ao longo da trajetória.

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]:

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

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

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

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

---

<sup>2</sup> $DLS[^\circ/30m] = \alpha/\Delta S \cdot 30$ , com  $\alpha$  em graus e  $\Delta S$  em metros.

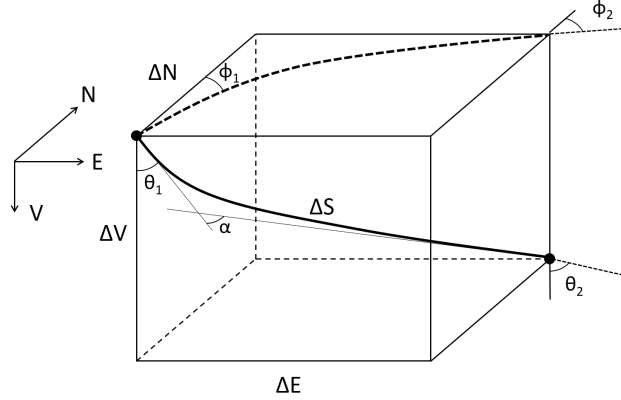


Figura 1: Parâmetros de perfuração entre dois pontos ao longo de um poço direcional.

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

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 parâmetros de perfuração entre dois pontos quaisquer são conhecidos os parâmetros de perfuração do ponto inicial<sup>3</sup> ( $\theta_1, \phi_1$ ) e as distâncias cartesianas entre os pontos ( $\Delta N, \Delta E, \Delta V$ ). O objetivo é calcular a direção do poço no 2º ponto ( $\theta_2$  e  $\phi_2$ ) e a distância medida entre os pontos ( $\Delta S$ ).

É possível inverter a equação 3 para obter uma expressão para  $\theta_2$ :

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

Dividindo a equação 1 pela equação 2 obtém-se duas expressões para  $\phi_2$ :

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

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

onde

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

<sup>3</sup>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  $\Delta Sf(\alpha)$ :

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

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  $\alpha$  e  $f(\alpha)$  durante o processo. O valor de  $\phi_2$  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  $\phi_2$  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 dividido 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<sup>4</sup>.

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

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

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*<sup>5</sup>:  $\frac{|x_i - x_{i-1}|}{|x_i|}$ .
- No início do código é verificado se  $|x_b - x_a| < \zeta$ , onde  $\zeta$  é calculado em função do critério de convergência estabelecido<sup>6</sup>. Se for *verdadeiro*, não é feito o *loop* do método.
- Se  $\text{senal}(f(x_a)) = \text{senal}(f(x_b))$  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  $\epsilon$  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 ( $f(x_a)$ ,  $f(x_b)$ ,  $f(x_{\text{medio}})$ ) e é retornado o valor de  $x$  com  $f(x)$  mais próximo de zero.

---

<sup>4</sup>Assumindo que o intervalo inicial fornecido também tem esta propriedade.

<sup>5</sup>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*!

<sup>6</sup>Definindo  $c_{\text{lim}}$  o limite de convergência, se for utilizado o critério de convergência *direto* então  $\zeta = c_{\text{lim}}$ . Se for utilizado o critério de convergência relativo então  $\zeta = c_{\text{lim}} \min |x_a|, |x_b|$  (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\_\_bisection()** 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 ( $sign(f(x_a)) = sign(f(x_b))$ ) 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 em função do problema a ser resolvido.

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 flutuante:

- **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*. Foi feita uma sensibilidade do critério de convergência ( $c_{lim}$ ) e foi possível trabalhar com limites de valores baixos sem aparente perda da qualidade da resposta por problemas com aritmética de máquina (Apêndice A.2).

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 bissecção tem garantia de convergência, mas não é garantido que o melhor resultado será o  $x_{medio}$  da última iteração.

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  $\Delta Sf(\alpha)$ . O Método da Bissecção se encaixa bem na resolução do problema proposto por trabalhar em uma faixa de valores pré-determinada. Devido a erros de aritmética de máquina, não é sempre possível garantir que  $sign(f(x_a)) \neq sign(f(x_b))$

quando a resposta está muito próxima do limite inferior. Desta forma, ao invés de gerar um erro quando  $\text{sinal}(f(x_a)) = \text{sinal}(f(x_b))$ , é gerada uma mensagem de alerta ao usuário 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 4º trecho do poço em 'S' descrito nos testes de `tests_minimum_curvature()` (Apêndice B). O valor exato de  $\theta_2$  é zero (o 5º 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  $f(\Delta Sf(\alpha)_{max})$ .

Os limites da região em que o problema proposto deve ser avaliado são definidos por um lado pelo menor valor possível de  $\Delta Sf(\alpha)$  discutido em 2.2. O valor máximo foi definido como  $[\Delta Sf(\alpha)]_{min} \cdot f(\alpha = 0.95\pi)$ . O limite superior é equivalente a aproximadamente um arco de  $180^\circ$ , o que não é algo factível em uma aplicação real, e que serve como um limite superior na região de busca.

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 (Apêndice A.1). 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 1).

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

onde

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

Tabela 1: Comparação entre o número de iterações previsto e realizado para diferentes testes.

Função	$x_a$	$x_b$	Critério <i>Direto</i>		Critério <i>Relativo</i>	
			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

O Método da Bissecção gerou bons resultados na aplicação proposta. Em um dos testes proposto o método não conseguiu ser aplicado porque a resposta correto é igual ao limite inferior e, por erro de aritmética de máquina, o valor da função não é zero neste ponto. Para diversos testes a solução foi trivial, pois a resposta exata é igual ao limite inferior de  $\Delta Sf(\alpha)$ . Na Tabela 2 todos os testes reportados foram feitos utilizando como critério de parada  $|x_{i+1} - x_i| < c_{lim} = 1E - 3$ .

Tabela 2: Erro das respostas de interesse com relação ao valor exato para testes realizados com algoritmo proposto e o Método da Bissecção.

Teste	Iter.	$\Delta Sf(\alpha)$	$\theta_2$	$\phi_2$	$\Delta S$
Linha reta inclinada	0	0	0	1.11e-16	0
1/4 de círculo	7	2.52e-4	1.26e-5	0	1.26e-4
Seção 2 do poço em S	20	4.37e-5	2.52e-6	0	2.89e-05
Poço 3D	17	4.00e-5	1.41e-5	1.51e-5	3.07e-05

#### 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 Bissecção 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/A11-03ATCE/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

### Apêndice A.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 Bissecção. São apresentadas duas retas tracejadas onde a razão entre os erros de iterações sucessivas é 0.5, que é o que se espera do Método da Bissecção após um número considerável de iterações. Observa-se que os resultados numéricos não seguem exatamente uma linha reta, mas têm aproximadamente a mesma inclinação das retas tracejadas.

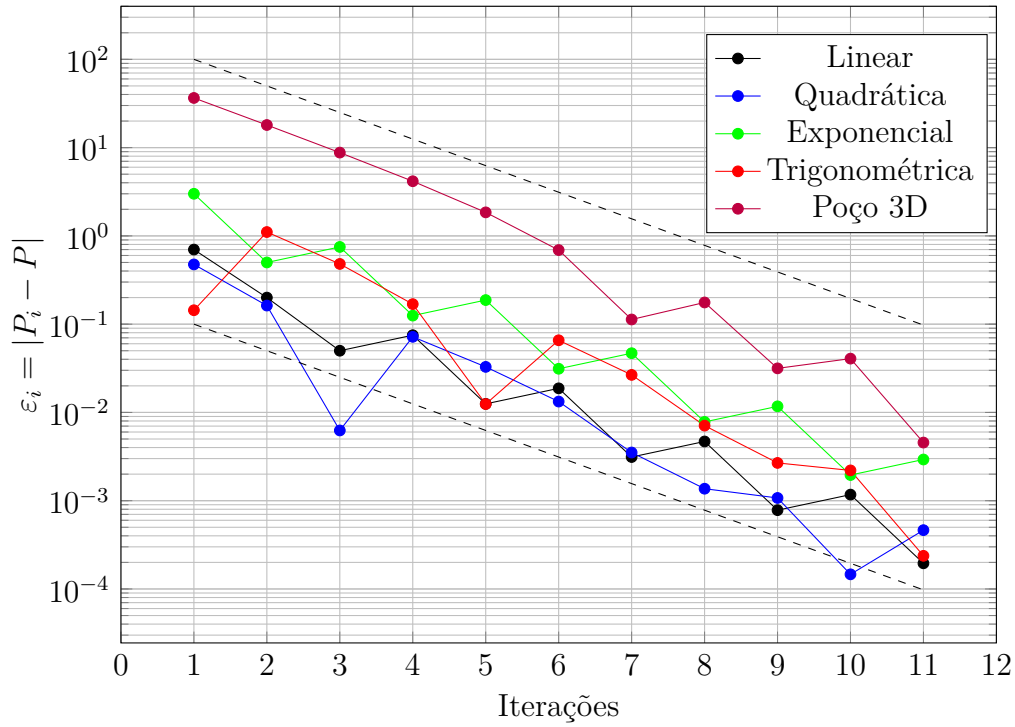


Figura A.2: Evolução do erro do Método da Bissecção com diferentes funções (linhas tracejadas representam  $\varepsilon_{i+1}/\varepsilon_i = 0.5$ ).

### Apêndice A.2. Erro em Função do Limite de Convergência

A figura a seguir mostra os erros (com relação ao valor exato) nas estimativas de diferentes parâmetros de perfuração. O algoritmo proposto foi aplicado a um trecho de poço com mudança em inclinação e direção, utilizando diferentes valores de limite de convergência ( $c_{lim}$ ).

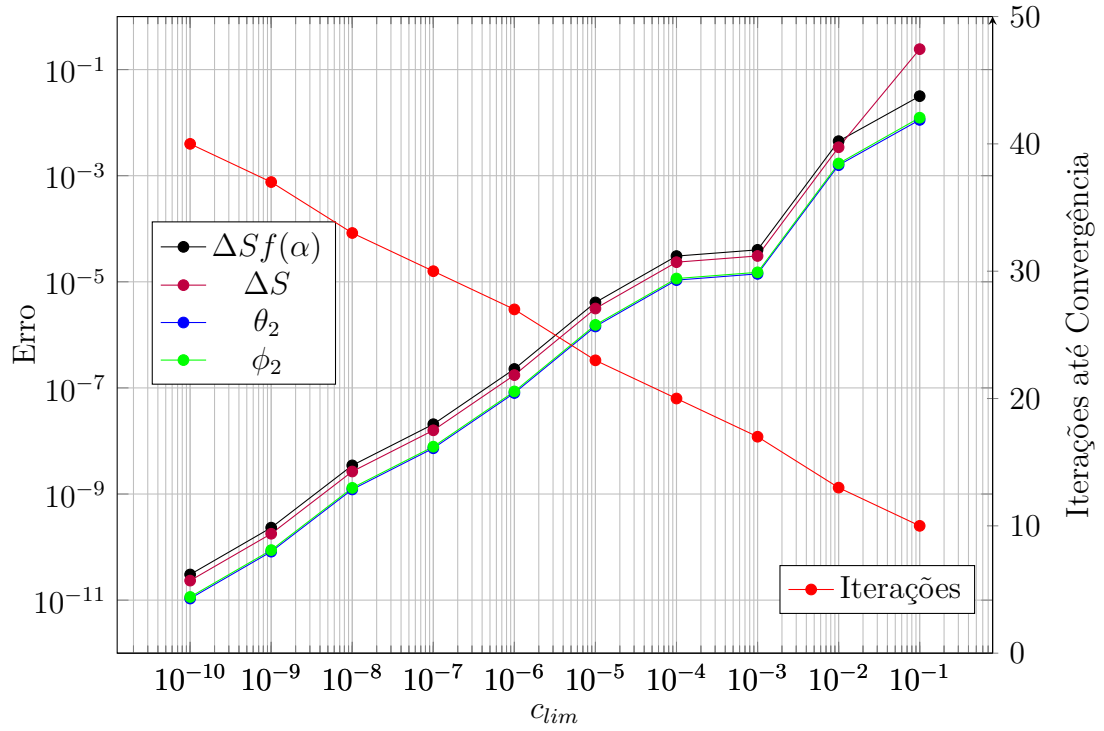


Figura A.3: Erro das variáveis de interesse em função do limite de convergência utilizado.

## Apêndice B. Esquema do Poço em S

Um dos exemplos testados com o algoritmo proposto para calcular parâmetros de perfuração em função de coordenadas cartesianas é um poço em S. Uma representação esquemática da trajetória do poço (sem escala) é apresentada abaixo.

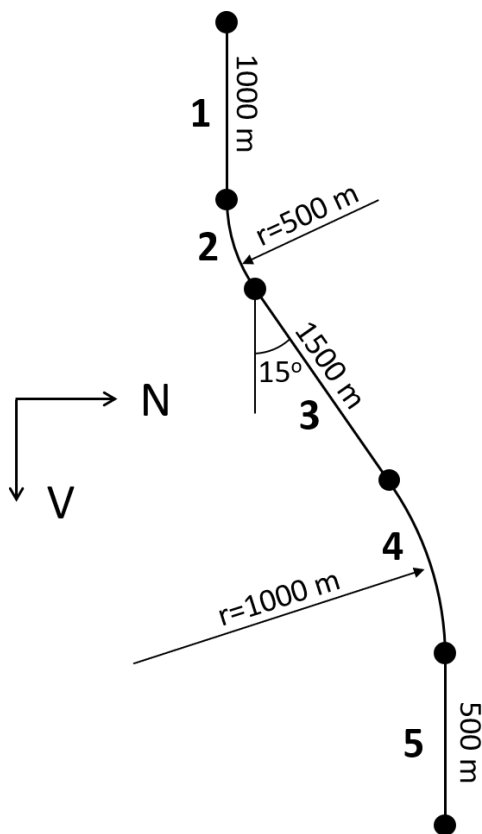


Figura B.4: Esquema do poço em 'S' descrito nos testes de `tests_minimum_curvature()`.

## 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*

```
1  /*
2      Implementation of the bisection method to find root of 1D functions
3  */
4
5  #include <stdio.h>
6  #include <stdbool.h>
7  #include <math.h>
8  #include <assert.h>
9
10 const double epsilon = 1E-7;
11 const double pi = 3.14159265358979323846;
12
13 const int default_iterations = 100;
14 const double default_convergence = 1E-4;
15
16 const bool stop_on_error = true;
17
18 bool check_error(bool ok_condition, char *error_message){
19     if( !ok_condition){
20         printf(error_message);
21         if( stop_on_error){
22             assert(false);
23         }
24     }
25     return ok_condition;
26 }
27
28 double min(double a, double b){
29     return a<b? a: b;
30 }
31
32 double max(double a, double b){
33     return a>b? a: b;
34 }
35
36 bool is_root(double fx, double x){
37     return fabs(fx) < epsilon;
38 }
39
40 double calculate_convergence(double x_current, double x_previous, bool
    relative_convergence){
41     double reference = 1;
42     if( relative_convergence)
```

```

43     if( fabs(x_current) > epsilon){
44         reference = x_current;
45     } else if( fabs(x_previous) > epsilon) {
46         reference = x_previous;
47     } else{
48         return 0.;
49     }
50     return fabs( (x_current - x_previous) / reference );
51 }
52
53 void print_error(char *message, double true_value, double calculated_value,
54 double error_limit, bool relative_convergence){
55     double relative_error;
56     relative_error = calculate_convergence(true_value, calculated_value,
57     relative_convergence);
58     printf("%s: True=%g Calculated=%g Error=%g", message, true_value,
59     calculated_value, relative_error);
60     if( relative_error > error_limit){
61         printf("    <= ##### Attention #####");
62     }
63     printf("\n");
64 }
65
66 double return_closest_to_root(double x_a, double fx_a, double x_b, double
67     fx_b){
68     if( fabs(fx_b) < fabs(fx_a)){
69         return x_b;
70     } else{
71         return x_a;
72     }
73 }
74
75 double return_closest_to_root_3pt(double x_a, double fx_a, double x_b, double
76     fx_b, double x_c, double fx_c){
77     fx_a = fabs(fx_a);
78     fx_b = fabs(fx_b);
79     fx_c = fabs(fx_c);
80     if( min(fx_a, fx_b) < fx_c){
81         if( fx_b < fx_a){
82             return x_b;
83         } else{
84             return x_a;
85         }
86     } else{
87         return x_c;
88     }
89 }
90
91 double find_root_bissection_debug(double (*func)(double), double x_a, double
92     x_b, double convergence_tol, bool relative_convergence, int max_iterations
93     , bool debug, double x_root) {

```

```

87     double fx_a, fx_b, fx_mean;
88     double x_mean, x_mean_previous;
89     double convergence;
90     enum type_exit_function {no_exit, root, converged, sign_error};
91     enum type_exit_function exit_function = no_exit;
92     bool print_true_error;
93     int i=0;
94
95     if( x_b < x_a){
96         x_mean = x_a;
97         x_a = x_b;
98         x_b = x_mean;
99     }
100
101     fx_a = func(x_a);
102     fx_b = func(x_b);
103     x_mean = 1e20;
104     fx_mean = 1e20;
105
106     convergence = calculate_convergence(min(x_a, x_b), max(x_a, x_b),
relative_convergence);
107     if( is_root(fx_a, x_a) || is_root(fx_b, x_b)){
108         exit_function = root;
109     };
110
111     if( exit_function == no_exit){
112         if( convergence < convergence_tol ){
113             if( debug){
114                 printf("Initial limits are closer than convergence criteria:
115                 |%g - %g| = %g < %g.\n",x_b,x_a,fabs(x_a - x_b), convergence_tol);
116             }
117             exit_function = converged;
118         }
119     }
120
121     if( exit_function == no_exit){
122         if( signbit(fx_a) == signbit(fx_b) ){
123             printf("Function has same sign in limits: f(%g) = %g f(%g) = %g.\n
124             n",x_a,fx_a,x_b,fx_b);
125             if( debug){
126                 double x_trial;
127                 printf("Sample of function results in the provided domain:\n"
128                 );
129                 for(int i=0; i<=20; i++){
130                     x_trial = x_a + (x_b - x_a)*i/20;
131                     printf("   f(%g) = %g\n", x_trial, func(x_trial));
132                 }
133                 exit_function = sign_error;
134             }
135         }
136     }

```

```

134
135     if( !exit_function){
136         print_true_error = (x_root >= x_a) && (x_root <= x_b);
137         x_mean_previous = x_a;
138         if( debug){
139             if( print_true_error){
140                 printf("%3s    %-28s\t%-28s\t%-28s\t%-11s\t%-11s\n", "#", "
Lower bound", "Upper bound", "Mean point", "Convergence", "|x - root|");
141             } else{
142                 printf("%3s    %-28s\t%-28s\t%-28s\t%-11s\n", "#", "Lower bound
", "Upper bound", "Mean point", "Convergence");
143             }
144         }
145
146         for(i=1 ; i<=max_iterations ; i++){
147             x_mean = x_a + (x_b - x_a)/2;
148             fx_mean = func(x_mean);
149             convergence = calculate_convergence(x_mean, x_mean_previous,
relative_convergence);
150             x_mean_previous = x_mean;
151
152             if( debug){
153                 if( print_true_error){
154                     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));
155                 } else{
156                     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);
157                 }
158             }
159
160             if( convergence < convergence_tol){
161                 exit_function = converged;
162                 break;
163             }
164
165             if( is_root(fx_mean, x_mean)){
166                 exit_function = root;
167                 break;
168             }
169
170             if( signbit(fx_a) == signbit(fx_mean)){
171                 x_a = x_mean;
172                 fx_a = fx_mean;
173             } else{
174                 x_b = x_mean;
175                 fx_b = fx_mean;
176             }
177         }
178     }

```

```

179     if( debug){
180         switch(exit_function)
181         {
182             case root: printf("Found |f(x)| < %g after %i iterations.\n",
epsilon, i); break;
183             case converged: printf("Reached convergence after %i iteration(s)
: %g < %g.\n", i, convergence, convergence_tol); break;
184             case sign_error: printf("Cannot continue. Returning result
closest to zero among f(x_a) and f(x_b).\n"); break;
185             case no_exit: printf("Convergence was not reached after %i
iteration(s): %g.\n", i-1, convergence); break;
186             default: printf("Unkown exit criteria.\n");
187         }
188     }
189     return return_closest_to_root_3pt(x_a, fx_a, x_b, fx_b, x_mean, fx_mean);
190 }
191
192 double find_root_bissection(double (*func)(double), double x_a, double x_b){
193     return find_root_bissection_debug(func, x_a, x_b, default_convergence,
true, default_iterations, false, -1e99);
194 }
195
196 int estimate_bissection_iterations(double x_a, double x_b, double x_root,
double convergence_tol, bool relative_convergence){
197     double x_reference = 1;
198     double n;
199
200     if( relative_convergence){
201         if(fabs(x_root)<epsilon || x_root < min(x_a,x_b) || x_root > max(x_a,
x_b)){
202             x_reference = min(fabs(x_a), fabs(x_b));
203             if( x_reference < epsilon)
204                 x_reference = max(fabs(x_a), fabs(x_b));
205             if( x_reference < epsilon)
206                 return 0;
207         } else{
208             x_reference = x_root;
209         }
210     }
211     n = log(fabs(x_b - x_a) / fabs(x_reference) / convergence_tol)/log(2);
212     return max(0, round(n+0.5));
213 }
214
215 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){
216     printf("\nTest Bissection Method: %s\n", message);
217     int iterations = estimate_bissection_iterations(x_a, x_b, x_root,
convergence_tol, false);
218     printf("Estimated number of iterations: %i\n", iterations);

```



```

219     double x_root_bissection = find_root_bissection_debug(func, x_a, x_b,
220     convergence_tol, relative_convergence, max_iterations, debug, x_root);
221     print_error(" => Root", x_root, x_root_bissection, convergence_tol,
222     relative_convergence);
223 }
224 // Root at x=0.3
225 double f_linear(double x){
226     return -x*3 + 0.9;
227 }
228 // Root at x=0.3
229 double f_linear2(double x){
230     return -x*3E5 + 0.9E5;
231 }
232 // Roots at x=-0.5 and -0.1
233 double f_quadratic(double x){
234     return (5*x + 3)*x + 0.25; // = 5*x*x + 3*x + 0.25;
235 }
236 // Root at x= +-2
237 double f_exponential(double x){
238     return exp(x*x-4) - 1;
239 }
240 // Root at x= pi/2 (+ n*2pi)
241 double f_cos(double x){
242     return cos(x);
243 }
244 // Root at x= 3/4*pi (+ n*2pi)
245 double f_trigonometric(double x){
246     return cos(x) + sin(x);
247 }
248 void tests_bissection(){
249     bool relative_convergence = false;
250     int max_iterations = 50;
251     bool debug = true;
252
253     test_bissection(f_linear, 0.3, 0, 2, 0.001, relative_convergence,
254     max_iterations, debug, "Linear function");
255     test_bissection(f_linear, 0.3, 2, 0, 0.001, relative_convergence,
256     max_iterations, debug, "Linear function with inverted limits");
257     test_bissection(f_linear, 0.3, 0, 2, 0.001, true, max_iterations, debug,
258     "Linear function with relative convergence");
259     test_bissection(f_linear, 0.3, 0, 1, 0.001, relative_convergence, 5,
260     debug, "Linear function with insufficient iterations");
261     test_bissection(f_linear, 0.3, 0., 0.4, 0.001, relative_convergence,
262     max_iterations, debug, "Linear function with early exit");

```

```

263     test_bissection(f_linear, 0.3, 0.3, 1, 0.001, relative_convergence,
max_iterations, debug, "Linear function with root in x_a");
264     test_bissection(f_linear, 0.3, 0, 0.3, 0.001, relative_convergence,
max_iterations, debug, "Linear function with root in x_b");
265     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");
266     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");
267
268     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");
269     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");
270
271     test_bissection(f_quadratic, -0.1, -0.25, 1, 0.001, relative_convergence
, max_iterations, debug, "Quadratic function, root#1");
272     test_bissection(f_quadratic, -0.5, -0.25, -1, 0.001, relative_convergence
, max_iterations, debug, "Quadratic function, root#2");
273     test_bissection(f_exponential, 2., 0, 10, 0.001, relative_convergence,
max_iterations, debug, "Exponential function");
274     test_bissection(f_cos, pi/2, 0, 2, 0.001, relative_convergence,
max_iterations, debug, "Cossine function");
275     test_bissection(f_trigonometric, 3./4*pi, 0, 5, 0.001,
relative_convergence, max_iterations, debug, "Trigonometric function");
276 }

```

## Apêndice C.2. Método da Mínima Curvatura

```

1  // Minimum Curvature Method
2  double angle_subtraction(double a2, double a1){
3      double dif = a2 - a1;
4      if( dif < -pi){
5          dif = dif + 2*pi;
6      } else if( dif > pi){
7          dif = dif - 2*pi;
8      }
9      return dif;
10 }
11
12 double alfa(double theta1, double phi1, double theta2, double phi2){
13     double x, y;
14     x = sin( angle_subtraction(theta2, theta1)/2 );
15     x *= x;
16     y = sin( angle_subtraction(phi2, phi1)/2 );
17     y *= y;
18     y *= sin(theta1)*sin(theta2);
19     x += y;
20     x = sqrt(x);
21     return 2* asin(x);
22 }

```

```

23
24 double f_alfa(double a){
25     if( a<0.02){
26         double x;
27         double a2 = a*a;
28         x = 1 + 32*a2/18;
29         x = 1 + a2/168*x;
30         x = 1+ a2/10*x;
31         return 1+ a2/12*x;
32     } else{
33         return 2/a * tan(a/2);
34     }
35 }
36
37 double deltaN_with_deltaSfa(double deltaSfa, double theta1, double phi1,
38     double theta2, double phi2){
39     return deltaSfa/2*(sin(theta1)*cos(phi1) + sin(theta2)*cos(phi2));
40 }
41
42 double deltaN_with_fa(double deltaS, double fa, double theta1, double phi1,
43     double theta2, double phi2){
44     return deltaN_with_deltaSfa(deltaS*fa, theta1, phi1, theta2, phi2);
45 }
46
47 double deltaN(double deltaS, double theta1, double phi1, double theta2,
48     double phi2){
49     double fa = f_alfa( alfa(theta1, phi1, theta2, phi2) );
50     return deltaN_with_fa(deltaS, fa, theta1, phi1, theta2, phi2);
51 }
52
53 double deltaE_with_deltaSfa(double deltaSfa, double theta1, double phi1,
54     double theta2, double phi2){
55     return deltaSfa/2*(sin(theta1)*sin(phi1) + sin(theta2)*sin(phi2));
56 }
57
58 double deltaE_with_fa(double deltaS, double fa, double theta1, double phi1,
59     double theta2, double phi2){
60     return deltaE_with_deltaSfa(deltaS*fa, theta1, phi1, theta2, phi2);
61 }
62
63 double deltaE(double deltaS, double theta1, double phi1, double theta2,
64     double phi2){
65     double fa = f_alfa( alfa(theta1, phi1, theta2, phi2) );
66     return deltaE_with_fa(deltaS, fa, theta1, phi1, theta2, phi2);
67 }
68
69 double deltaV_with_deltaSfa(double deltaSfa, double theta1, double theta2){
70     return deltaSfa/2*(cos(theta1) + cos(theta2));
71 }
72

```

```

68 double deltaV_with_fa(double deltaS, double fa, double theta1, double theta2)
69 {
70     return deltaV_with_deltaSfa(deltaS*fa, theta1, theta2);
71 }
72 double deltaV(double deltaS, double theta1, double phi1, double theta2,
73 double phi2){
74     double fa = f_alfa( alfa(theta1, phi1, theta2, phi2) );
75     return deltaV_with_fa(deltaS, fa, theta1, theta2);
76 }
77 struct tangle{
78     double cos, sin, rad;
79 };
80
81 double calculate_rad(struct tangle angle, bool debug){
82     check_error( fabs(angle.cos) <= 1., "## Failed |cos(angle)| <= 1.\n");
83     check_error( fabs(angle.sin) <= 1., "## Failed |sin(angle)| <= 1.\n");
84
85     angle.cos = min(1,max(angle.cos,-1));
86     angle.sin = min(1,max(angle.sin,-1));
87
88     double a_cos = acos(angle.cos);
89     double a_sin = asin(angle.sin);
90
91     if( angle.sin < 0){
92         a_cos = 2*pi - a_cos;
93         if( angle.cos > 0){
94             a_sin = 2*pi + a_sin;
95         }
96     }
97     if( angle.cos < 0){
98         a_sin = pi - a_sin;
99     }
100
101     double a = (a_sin + a_cos)/2;
102     if( debug){
103         double convergence = calculate_convergence(a, a_cos, true);
104         printf("Convergence error in angle calculation (%4g.pi rad): %g\n", a
105 /pi, convergence);
106     }
107     return a;
108 }
109 struct tangle calculate_theta2(double deltaV, double deltaSfa, double
110 cos_theta1){
111     struct tangle theta2;
112     theta2.cos = 2*deltaV / deltaSfa - cos_theta1;
113     if( check_error( fabs(theta2.cos) <= 1., "## Failed |cos(theta2)| <= 1.\n
114 ")){
115         theta2.sin = sqrt(1 - theta2.cos * theta2.cos);

```

```

114     } else{
115         theta2.sin = 0;
116     }
117     return theta2;
118 }
119
120 struct tangle calculate_phi2_deltaE_zero(double deltaN, double sin_theta1,
double cos_phi1, double sin_phi1, double sin_theta2){
121     struct tangle phi2;
122     if( fabs(sin_theta2)<epsilon && fabs(sin_theta1)<epsilon){
123         phi2.sin = -sin_phi1;
124     } else{
125         if( check_error( fabs(sin_theta2) > epsilon, "## Failed sin(theta2)
!=0.\n")){
126             phi2.sin = - sin_theta1 / sin_theta2 * sin_phi1;
127         } else{
128             phi2.sin = -sin_phi1;
129         }
130     }
131     if( check_error( fabs(phi2.sin) <= 1, "## Failed |sin(phi2)| <= 1.\n")){
132         phi2.cos = sqrt(1 - phi2.sin*phi2.sin);
133         if( cos_phi1 < 0){
134             phi2.cos = -phi2.cos;
135         }
136     } else{
137         phi2.cos = 0;
138     }
139     return phi2;
140 }
141
142 struct tangle calculate_phi2_deltaN_zero(double deltaE, double sin_theta1,
double cos_phi1, double sin_phi1, double sin_theta2){
143     struct tangle phi2;
144     if( fabs(sin_theta2)<epsilon && fabs(sin_theta1)<epsilon){
145         phi2.cos = -cos_phi1;
146     } else{
147         check_error( fabs(sin_theta2) > epsilon, "## Failed sin(theta2) !=
0.\n");
148         phi2.cos = - sin_theta1 / sin_theta2 * cos_phi1;
149     }
150     check_error( fabs(phi2.cos) <= 1, "## Failed |cos(phi2)| <= 1.\n");
151     phi2.sin = sqrt(1 - phi2.cos*phi2.cos);
152     if( sin_phi1 > 0){
153         phi2.sin = -phi2.sin;
154     }
155     return phi2;
156 }
157
158 struct tangle calculate_phi2(double deltaE, double deltaN, double sin_theta1,
double cos_phi1, double sin_phi1, double sin_theta2){
159     struct tangle phi2;

```

```

160
161     if( fabs(sin_theta2) < epsilon){
162         phi2.cos = cos_phi1;
163         phi2.sin = sin_phi1;
164     } else if( fabs(deltaE) < epsilon && fabs(deltaN) < epsilon){
165         phi2.cos = -cos_phi1;
166         phi2.sin = -sin_phi1;
167     } else if( fabs(deltaE) < epsilon){
168         phi2 = calculate_phi2_deltaE_zero(deltaN, sin_theta1, cos_phi1,
169 sin_phi1, sin_theta2);
170     } else if( fabs(deltaN) < epsilon){
171         phi2 = calculate_phi2_deltaN_zero(deltaE, sin_theta1, cos_phi1,
172 sin_phi1, sin_theta2);
173     } else{
174         double deltaEpsilon = deltaN * sin_phi1 - deltaE * cos_phi1;
175         double sin_theta1_sin_theta2 = sin_theta1 / sin_theta2;
176         double deltaH2 = deltaE*deltaE + deltaN*deltaN;
177         double deltaBeta2 = deltaH2 - deltaEpsilon * deltaEpsilon *
178 sin_theta1_sin_theta2 * sin_theta1_sin_theta2;
179
180         check_error( deltaBeta2 >= 0, "## Failed deltaBeta2 >= 0.\n");
181         deltaBeta2 = max(0, deltaBeta2);
182         double deltaBeta = sqrt(deltaBeta2);
183
184         phi2.sin = (-deltaN * deltaEpsilon * sin_theta1_sin_theta2 + deltaE *
185 deltaBeta) / deltaH2;
186         phi2.cos = ( deltaE * deltaEpsilon * sin_theta1_sin_theta2 + deltaN *
187 deltaBeta) / deltaH2;
188     }
189     return phi2;
190 }
191
192 double calculate_deltaSfa(double deltaE, double deltaN, double deltaV, double
193 cos_theta1, double sin_theta1, double cos_phi1, double sin_phi1, double
194 cos_theta2, double sin_theta2, double cos_phi2, double sin_phi2){
195     double aE = sin_theta1 * sin_phi1 + sin_theta2 * sin_phi2;
196     double aN = sin_theta1 * cos_phi1 + sin_theta2 * cos_phi2;
197     double aV = cos_theta1 + cos_theta2;
198     return 2 * sqrt( (deltaE*deltaE + deltaN*deltaN + deltaV*deltaV) / (aE*aE
199 + aN*aN + aV*aV) );
200 }
201
202 double calculate_deltaSfa_aproximate(double deltaE, double deltaN, double
203 deltaV, double cos_theta1, double sin_theta1, double cos_phi1, double
204 sin_phi1, double deltaSfa){
205     struct tangle theta2 = calculate_theta2(deltaV, deltaSfa, cos_theta1);
206     struct tangle phi2 = calculate_phi2(deltaE, deltaN, sin_theta1, cos_phi1,
207 sin_phi1, theta2.sin);
208     return calculate_deltaSfa(deltaE, deltaN, deltaV, cos_theta1, sin_theta1,
209 cos_phi1, sin_phi1, theta2.cos, theta2.sin, phi2.cos, phi2.sin);
210 }

```

```

199
200 double calculate_deltaSfa_error(double deltaE, double deltaN, double deltaV,
    double cos_theta1, double sin_theta1, double cos_phi1, double sin_phi1,
    double deltaSfa){
201     double deltaSfa_calc = calculate_deltaSfa_aproximate( deltaE, deltaN,
    deltaV, cos_theta1, sin_theta1, cos_phi1, sin_phi1, deltaSfa);
202     return deltaSfa_calc - deltaSfa;
203 }
204
205 double dE, dN, dV, cos_theta1, sin_theta1, cos_phi1, sin_phi1;
206 void define_well_path_data(double deltaE, double deltaN, double deltaV,
    double theta1, double phi1){
207     dE = deltaE;
208     dN = deltaN;
209     dV = deltaV;
210     dE = deltaE;
211     cos_theta1 = cos(theta1);
212     sin_theta1 = sin(theta1);
213     cos_phi1 = cos(phi1);
214     sin_phi1 = sin(phi1);
215 }
216 double calculate_defined_deltaSfa_error(double deltaSfa){
217     return calculate_deltaSfa_error(dE, dN, dV, cos_theta1, sin_theta1,
    cos_phi1, sin_phi1, deltaSfa);
218 }
219
220 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){
221
222     printf("\n%s\n", message);
223
224     double a = alfa(theta1, phi1, theta2, phi2);
225     double dE = deltaE(deltaS, theta1, phi1, theta2, phi2);
226     double dN = deltaN(deltaS, theta1, phi1, theta2, phi2);
227     double dV = deltaV(deltaS, theta1, phi1, theta2, phi2);
228
229     if( report_alfa_dEdNdV){
230         print_error(" Alfa",true_alfa,a, convergence_limit,
    relative_convergence);
231         print_error(" Delta E", true_deltaE,dE, convergence_limit,
    relative_convergence);
232         print_error(" Delta N", true_deltaN,dN, convergence_limit,
    relative_convergence);
233         print_error(" Delta V", true_deltaV,dV, convergence_limit,
    relative_convergence);
234     } else{
235         printf(" Calculated displacement: dE=%g dN=%g dV=%g\n", dE, dN, dV);
236         printf(" Calculated alfa=%g\n",a);
237         printf(" Calculated f(alfa)=%g\n",f_alfa(a));

```

```

238 }
239
240 double deltaSfa = deltaS * f_alfa(a);
241 struct tangle theta2_ = calculate_theta2(dV, deltaSfa, cos(theta1));
242 print_error("  theta2", theta2, calculate_rad(theta2_, false),
convergence_limit, relative_convergence);
243
244 struct tangle phi2_ = calculate_phi2(dE, dN, sin(theta1), cos(phi1), sin(
phi1), sin(theta2));
245 print_error("  phi2", phi2, calculate_rad(phi2_, false),
convergence_limit, relative_convergence);
246
247 double deltaSfa_calc = calculate_deltaSfa(dE, dN, dV, cos(theta1), sin(
theta1), cos(phi1), sin(phi1), cos(theta2), sin(theta2), cos(phi2), sin(
phi2));
248 double dS = deltaSfa_calc / f_alfa(a);
249 print_error("  Delta S", deltaS, dS, convergence_limit,
relative_convergence);
250
251 printf("Calculate theta2, phi2 and dS from dE, dN, dV, theta1 and phi1.\n
");
252 define_well_path_data( dE, dN, dV, theta1, phi1);
253 double deltaSfa_min = sqrt(dE*dE + dN*dN + dV*dV);
254 double deltaSfa_max = deltaSfa_min * f_alfa(0.95*pi);
255 if( dV > 0){
256     deltaSfa_min = max(deltaSfa_min, 2*dV/(cos_theta1+1) );
257 } else if( dV < 0){
258     deltaSfa_min = max(deltaSfa_min, 2*dV/(cos_theta1-1) );
259 }
260
261 int estimated_iterations = estimate_bisection_iterations(deltaSfa_min,
deltaSfa_max, -9999, convergence_limit, relative_convergence);
262 printf("Estimated number of iterations: %i\n", estimated_iterations);
263 deltaSfa_calc = find_root_bisection_debug(
calculate_defined_deltaSfa_error, deltaSfa_min, deltaSfa_max,
convergence_limit, relative_convergence, 100, true, deltaSfa);
264
265 print_error("  Delta S x f(alfa)", deltaSfa, deltaSfa_calc,
convergence_limit, relative_convergence);
266 theta2_ = calculate_theta2(dV, deltaSfa_calc, cos(theta1));
267 print_error("  theta2", theta2, calculate_rad(theta2_, false),
convergence_limit, relative_convergence);
268 phi2_ = calculate_phi2(dE, dN, sin(theta1), cos(phi1), sin(phi1), theta2_
.sin);
269 print_error("  phi2", phi2, calculate_rad(phi2_, false),
convergence_limit, relative_convergence);
270 a = alfa(theta1, phi1, calculate_rad(theta2_, false), calculate_rad(phi2_
, false));
271 dS = deltaSfa_calc / f_alfa(a);
272 print_error("  Delta S", deltaS, dS, convergence_limit,
relative_convergence);

```



```

273 }
274
275 void tests_minimum_curvature(){
276     double theta1, phi1, theta2, phi2;
277     double a;
278     double dS, dE, dN, dV;
279     struct tangle theta2_, phi2_;
280     double theta2_calc, phi2_calc;
281     double dSfa, dS_calc;
282     char message[100];
283
284     bool relative_convergence = false;
285     double convergence_limit = 0.001;
286
287     printf("\n#### Minimum Curvature Method Tests ####\n");
288
289     dS=10;
290     theta1=0;    phi1=0.88*pi;
291     theta2=0;    phi2=0.88*pi;
292     a=0.;
293     dE=0., dN=0., dV=dS;
294     test_MCM_formulas("Vertical well", dS, theta1, phi1, theta2, phi2, a, dE,
295                       dN, dV, true, relative_convergence, convergence_limit);
296
297     dS=10;
298     theta1=pi/4;    phi1=pi/6;
299     theta2=pi/4;    phi2=pi/6;
300     a=0.;
301     dE=dS*sin(pi/4)*sin(pi/6);
302     dN=dS*sin(pi/4)*cos(pi/6);
303     dV=dS*cos(pi/4);
304     test_MCM_formulas("Slant straight well", dS, theta1, phi1, theta2, phi2,
305                       a, dE, dN, dV, true, relative_convergence, convergence_limit);
306
307     dS=10*pi/2;
308     theta1=pi/2;    phi1=3*pi/2;
309     theta2=pi/2;    phi2=0.;
310     a=pi/2;
311     dE=-10;    dN=10;    dV=0.;
312     test_MCM_formulas("1/4 circle horizontal well", dS, theta1, phi1, theta2,
313                       phi2, a, dE, dN, dV, true, relative_convergence, convergence_limit);
314
315     dS=10*pi/2;
316     theta1=pi/2;    phi1=pi/4;
317     theta2=pi/2;    phi2=7*pi/4.;
318     a=pi/2;
319     dE=0.;    dN=10*sqrt(2);    dV=0.;
320     test_MCM_formulas("1/4 circle deltaE=0 horizontal well", dS, theta1, phi1,
321                       theta2, phi2, a, dE, dN, dV, true, relative_convergence,
322                       convergence_limit);
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338

```

```

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

325
326     dS=10*pi/2;
327     theta1=pi/6;          phi1=0.;
328     theta2=theta1+pi/2;   phi2=0.;
329     a=pi/2;
330     dE=0.;      dN=10.*(cos(pi/6)+sin(pi/6));      dV=10.*(cos(pi/6)-sin(pi/6));
331     test_MCM_formulas("1/4 circle aligned north well 'going up'", dS, theta1
, phi1, theta2, phi2, a, dE, dN, dV, true, relative_convergence,
convergence_limit);

332
333     dS=10*pi/2;
334     theta1=pi/4;          phi1=0.;
335     theta2=theta1+pi/2;   phi2=0.;
336     a=pi/2;
337     dE=0.;      dN=10.*(cos(theta1)+sin(theta1));      dV=10.*(cos(theta1)-sin(
theta1));
338     test_MCM_formulas("1/4 circle aligned north well 'going up' #2", dS,
theta1, phi1, theta2, phi2, a, dE, dN, dV, true, relative_convergence,
convergence_limit);

339
340     dS=10*pi/2;
341     theta1=pi/3;          phi1=0.;
342     theta2=theta1+pi/2;   phi2=0.;
343     a=pi/2;
344     dE=0.;      dN=10.*(cos(theta1)+sin(theta1));      dV=10.*(cos(theta1)-sin(
theta1));
345     test_MCM_formulas("1/4 circle aligned north well 'going up' #3", dS,
theta1, phi1, theta2, phi2, a, dE, dN, dV, true, relative_convergence,
convergence_limit);

346
347     dS=10*pi/2;
348     theta1=0;          phi1=0.1871*pi;
349     theta2=pi/2;       phi2=pi/6;
350     a=pi/2;
351     dE=10.*sin(pi/6);      dN=10.*cos(pi/6);      dV=10.;
352     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);

353
354     double array_dS[5]={1000., 500*pi/12, 1500., 1000*pi/12, 500.};
355     double array_theta[6]={0., 0., pi/12, pi/12, 0., 0.};
356     double array_phi[6]={0., 0., 0., 0., 0., 0.};
357     double array_a[5]={0., pi/12, 0, pi/12, 0.};

```

```

358     double array_dE[5]={0., 0., 0., 0., 0.};
359     double array_dN[5]={0., 500.*(1-cos(pi/12)), 1500.*sin(pi/12), 1000.*(1-
cos(pi/12)), 0.};
360     double array_dV[5]={1000., 500.*sin(pi/12), 1500.*cos(pi/12), 1000.*sin(
pi/12), 500.};
361     printf("\n'S-shaped' well\n");
362     for( int i=0; i<5; i++){
363         sprintf(message, "Section #%i",i+1);
364         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);
365     }
366
367     dS=10.;
368     theta1=pi/10;      phi1=pi/4;
369     theta2=pi/6;      phi2=7*pi/4;
370     a=0.;
371     dE=0.;    dN=0.;    dV=0.;
372
373     double c;
374     for( int i = 1; i<= 5; i++){
375         c = pow(10, -i);
376         sprintf(message, "'3D' well with convergence = %g", c);
377         test_MCM_formulas(message, dS, theta1, phi1, theta2, phi2, a, dE, dN,
dV, false, relative_convergence, c);
378     }
379 }
380
381 int main(){
382     tests_bissection();
383     tests_minimum_curvature();
384     return 0;
385 }

```