UNIVERSIDADE FEDERAL DO CEARÁ

Métodos Numéricos

Equipe:
ABNER DE LIMA ARAÚJO – Matrícula: 398067
EDEALCIA REGINA ALVES LUCIANO – Matrícula: 372164
FRANCISCO WANDSON GOES SANTOS – Matrícula: 398777
MARCOS VINICIUS GOMES DE SANTANA – Matrícula: 400685
MILTON CASSUL MIRANDA – Matrícula: 404877
THIAGO FRAXE CORREIA – Matrícula: 397796

1º Trabalho de Métodos Numéricos I - Raízes de Equações

Tema 4:
Uma determinada reação química produz uma quantidade c de CO2 medida em ppm (parte por milhão) dada pela equação polinomial
f(c) = a4c4 + a3c3 + a2c2 + a1c + a0. Se ξ é uma raiz de f(x) = 0 com multiplicidade p, dados xo e ε, para cada passo o método de Newton-Raphson é dado por xk+1 = xk – (pf(xk) / f ’(xk)) (k = 0,1,2...). De forma análoga pode-se introduzir um fator p no método da Secante para raízes múltiplas, obtendo então, xk+1 = xk – (pf(xk)(xk – xk-1))/(f(xk) – f(xk-1)) (k = 0,1,2,...).
Desenvolva um sistema para calcular a quantidade c de CO2 de uma determinada reação química dada. O sistema deve atender aos seguintes requisitos dados pelos itens abaixo:

Método de Newton para polinômios.

- Exemplo:
  $p4(x) = a4x^4 + a3x^3 + a2x^2 + a1x + a0$

- Forma dos parênteses encaixados:
  $p4(x) = (((a4x + a3)x + a2)x + a1)x + a0$

- Dado $x \in \mathbb{R}$, calculamos $p4(x)$ da seguinte forma:
  $b4 = a4$
  $b3 = a3 + b4x$
  $b2 = a2 + b3x$
  $b1 = a1 + b2x$
  $b0 = a0 + b1x$

Método de Newton para polinômios.

- Semelhantemente, a derivada $p4'(x)$ pode ser calculada:
  $p4'(x) = 4a4x^3 + 3a3x^2 + 2a2x + a1 \quad \longrightarrow \quad p4'(x) = b4x^3 + b3x^2 + b2x + b1$

- Forma dos parênteses encaixados:
  $p4'(x) = ((b4x + b3)x + b2)x + b1$

  $c4 = b4$
  $c3 = b3 + c4x$
  $c2 = b2 + c3x$
  $c1 = b1 + c2x$

Método de Newton para polinômios.

- Algoritmo

```
NewtonParaPolinomios(A, x, ε)
          para k ⟶ 1 até it_max faça
                    b = A[n]
                    c = b
                    para i ⟶ n - 1 até 1 faça
                              b = A[i] + bx        //Calcula p(x)
                              c = b + cx           //Calcula p’(x)
                    b = A[0] + bx
                    se |b|< ε então
                              retorna x
                    Δx = b / c
                    x = x - Δx
                    se |Δx| < ε então
                              retorna x
          erro “Não foi possível encontrar uma aproximação boa o suficiente”
```

Método de Newton para Multiplicidade

- Se a multiplicidade da raiz a ser procurada pelo método de Newton-Raphson for maior do que 1, a convergência do método deixa de ser quadrática e passa a ser **linear.**

- Para contornar esse problema e manter a convergência quadrática, usamos uma versão levemente alterada do método de Newton-Raphson:

$$\mathbf{x_{k+1} = x_k - m\, f'(x)/f(x)}$$

- No entanto, é necessário já saber, de antemão, a multiplicidade m da raiz para a qual a sequência $\{x_i\}$ converge.

Método de Newton para Multiplicidade

Exemplo:

Seja $P(x) = x^4 - 5x^3 + 6x^2 + 4x - 8$. Queremos encontrar a raiz de multiplicidade 3 desse polinômio.

Usando uma precisão $\varepsilon = 0{,}0001$ e 0 como nosso chute inicial, obtemos, aplicando o Método de Newton Para Multiplicidade e após 10 iterações, o número 2 como nossa aproximação para essa raiz. (Utilizamos o teste $|x_k - x_{k+1}| < \varepsilon$ como critério de parada).

Como $P(x)$ pode ser escrito alternativamente como $P(x) = (x-2)^3(x+1)$, confirmamos que a aproximação encontrada é a raiz de multiplicidade 3 de $P(x)$.

Implementação Método Newton Multiplicidade

- Utiliza o método calcular da classe Polinômio para calcular o valor do polinômio no valor x especificado.
- Utiliza método gerarDerivada para obter a derivada do polinômio. Esse método tem complexidade linear. Desvantagem: mais lento, por um fator constante, que aplicação direta do Método de Horner.

```
Resultado calcularRaizNewtonMultiplicidade(unsigned short int p, double precisao)
{
    //Preparando o objeto Resultado
    Resultado resultado;
    stringstream polinomio;
    polinomio << *this;
    resultado.setPolinomio(polinomio.str());
    resultado.setMetodo("Método de Newton para Multiplicidade");

    unsigned short int numIter = 0;
    Polinomio derivada = this->gerarDerivada();

    //Fazendo isolamento para obter chute inicial
    double* intervalo = (double*) calloc(2, sizeof(double));
    this->getIntervalo(intervalo);
    double currentValue = intervalo[0];
    free(intervalo);
    resultado.setChuteInicial(currentValue);
    double nextValue;
    while (numIter < 1000)
    {

        nextValue = currentValue - p*this->calcular(currentValue)/derivada.calcular(currentValue);
        if (abs(nextValue - currentValue) < precisao){
            resultado.setRaiz(nextValue);
            resultado.setNumIter(numIter + 1);
            resultado.setError(false);
                    raizesResultado[1] = resultado.getRaiz();
            return resultado;

        }
        currentValue = nextValue;
        numIter++;
    }
    resultado.setRaiz(0.0);
    resultado.setNumIter(numIter + 1);
    resultado.setError(true);
        raizesResultado[1] = resultado.getRaiz();
    return resultado;

}
```

Método da Secante para Multiplicidade

- Método de Newton:

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}$$

- Grande desvantagem:

Obter f’(x) e calcular seu valor a cada iteração (pode ser difícil).

Método da Secante para Multiplicidade

- Solução:

Aproximar f'(x) pelo quociente das diferenças:

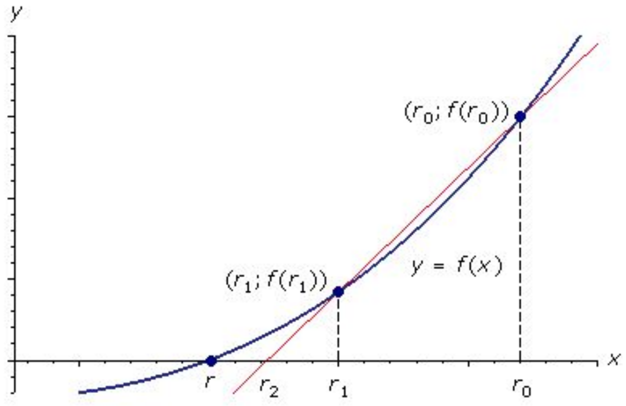
$$f'(x_k) \approx \frac{f(x_k) - f(x_{k-1})}{x_k - x_{k-1}}$$

Aplicando a aproximação ao método de Newton obtemos que:

$$\varphi(x_k) = \frac{x_{k-1}f(x_k) - x_k f(x_{k-1})}{f(x_k) - f(x_{k-1})}$$

Método da Secante para Multiplicidade

- Solução:



Através de duas aproximações x_{k-1} e x_k encontramos x_{k+1}, que é calculado como sendo a interseção da reta secante que passa pelo ponto $(x_{k-1}, f(x_{k-1}))$ e por $(x_k, f(x_k))$ com o eixo das abscissas.

- Ou seja, na primeira iteração são necessárias duas aproximações para $\boldsymbol{\xi}$, x_0 e x_1.

Método da Secante para Multiplicidade

- Algoritmo:

```
Algoritmo: Secante
Entrada: x0, x1, ε1, ε2, iterMax
Saída: raiz
  se abs(f(x0)) < ε1 então raiz ← x0; Fim.
  se abs(f(x1)) < ε1 ou abs(x1−x0) < ε2 então raiz ← x1; Fim.
  k ← 1
  repita
    x2 ← x1 − f(x1)/(f(x1) − f(x0)) * (x1−x0)
    escreva k, x2, f(x2)
    se abs(f(x2)) < ε1 ou abs(x2−x1) < ε2 ou k ≥ iterMax então
      raiz ← x2; Fim.
    fim se
    x0 ← x1
    x1 ← x2
    k ← k+1
  fim repita
fim algoritmo
```

Método da Secante para Multiplicidade

- Código:

```cpp
Resultado calcularRaizSecanteMultiplicidade(double precisao,int multiplicidade)
{
    double xk_prox, xk_anterior, xk_atual, *intervalo;
    int numIter=1;
    Resultado retorno;

    intervalo = (double*) calloc(2, sizeof(double));
    this->getIntervalo(intervalo);

    stringstream polinomio;
    polinomio << *this;
    retorno.setPolinomio(polinomio.str());
    retorno.setMetodo("Metodo da Secante para multiplicidade");

    xk_anterior = intervalo[0] + 0.5;
    xk_atual = xk_anterior + 0.2;

    retorno.setChuteInicial(xk_atual);

    if(abs(this->calcular(xk_anterior)) < precisao)
    {
        retorno.setNumIter(numIter);
        retorno.setRaiz(xk_anterior);
        free(intervalo);
        return retorno;
    }

    if( (abs(this->calcular(xk_atual)) < precisao) ||
        (abs(xk_atual - xk_anterior)   < precisao) )
    {
        retorno.setNumIter(numIter);
        retorno.setRaiz(xk_atual);
        free(intervalo);
        return retorno;
    }

    while(numIter < 1000)
    {
        xk_prox = xk_atual - ( (multiplicidade*this->calcular(xk_atual)*(xk_atual - xk_anterior))/(this->calcular(xk_atual) - this->calcular(xk_anterior)) );


        if( (abs(this->calcular(xk_prox)) < precisao) ||
            (abs(xk_prox - xk_atual)      < precisao) )
        {
            retorno.setNumIter(numIter);
            retorno.setRaiz(xk_prox);
            free(intervalo);
            return retorno;
        }

        xk_anterior = xk_atual;
        xk_atual = xk_prox;

        numIter++;
    }

    retorno.setRaiz(xk_atual);
    retorno.setNumIter(numIter);
    retorno.setError(true);
    free(intervalo);
    return retorno;
}
```

d) Calibrar o sistema usando como padrão a4=1, a3=-5, a2=6, a1=4, a0=-8, e p=3.

e) Fornecer um quadro resposta, com quantidade calculada para cada método dado.

f) Fornecer um quadro comparativo, com todos os dados para cada método dado.
Dados de entrada: n (número de reações), ak (k=0 a 4) e p (para cada opção) e ε (precisão).
Dados de saída: quadros resposta (com c para cada reação e método) e comparativo

- Aqui apenas fornecemos os um quadro comparativo apresentando o resultado de cada método dado
-

Diagrama de Classes

Polinômio

+ grau: Int
+ coeficientes: vector<double>
+ raizesResultado: double*

+ calcular(x: double): double
+ gerarDerivada(): Polinomio
+ getIntervalo(ret: double*): void
+ calcularRaizNewtonMultiplicidade(int, double): Resultado
+ calcularRaizSecanteMultiplicidade(double, int): Resultado
+ calcularRaizNewtonPolinomios(double): Resultado

Resultado

+ raiz: Double
+ numIter: Int
+ error: Bool
+ chuteInicial: double
+ polinomio: String
+ metodo: String

+setRaiz(double): void
+setNumIter(int): void

+setError(bool): void
+setChuteInicial(double): void
+setPolinomio(string): void
+setMetodo(string)
+getRaiz():double
+getNumIter():int
+getError():bool
+getChuteInicial:double
+getPolinomio:string
+getMetodo:string

Classe Polinômio

- Permite a manipulação de polinômios de qualquer grau
- Pode ser útil em trabalhos futuros
- Facilita debugagem e melhora a legibilidade do código

Classe Polinômio

```cpp
double calcular(double x) //Calcula valor do polinômio para um certo valor x
{
    double resultado = 0;

    for (int i = 0; i < this->grau + 1; i++)
    {
        resultado *= x;
        resultado += this->coeficientes[i];
    }

    return resultado;
}
```

```cpp
Polinomio gerarDerivada(void) const
{
    vector<double> novosCoeficientes;
    unsigned short int novoGrau = this->grau - 1;

    for(int i = 0; i <= novoGrau; i++)
    {
        novosCoeficientes.push_back(this->coeficientes[i] * (this->grau - i));
    }

    return Polinomio(novoGrau, novosCoeficientes);

}
```

Classe Resultado

Objetos dessa classe sempre são gerados quando um dos métodos para encontrar raízes é chamado.

Eles facilitam a construção do quadro comparativo, pois contêm todas as informações referentes ao cálculo da aproximação: se o método foi ou não bem-sucedido em encontrar a aproximação, quantas iterações foram necessárias, a aproximação encontrada etc.

Exemplos

$P1(x) = x^4 - 4x^3 - 18x^2 + 108x - 135$. Multiplicidade 3.

$P2(x) = x^4 + 4x^3 - 26x^2 - 60x + 225$. Multiplicidade 2.

$P3(x) = x^4 - 20x^3 + 150x^2 - 500x + 625$. Multiplicidade 4.

$P4(x) = x^4 - 5x^2 + 4$. Multiplicidade 1.

Conclusões

- Isolamento, para o caso dos métodos de multiplicidade, não foi feito de maneira ideal
- O método de Newton para multiplicidade encontra raízes com multiplicidade maior do que 1 em menos iterações que o método de Newton para polinômios, confirmando a maior taxa de convergência
- Dos três métodos, para os exemplos da página anterior, o método de Newton para Multiplicidade é o que requer o menor número de iterações para encontrar a raíz.