Universidade de São Paulo
Instituto de Matemática e Estatistica
IME

EP3 - Laboratório de Métodos Numéricos

Patrícia da Silva Rodrigues (nºUSP 11315590)

Junho
2023

1 Parte I - Computando trabalho

1. Trabalho = Força × Distância:

$$\text{trabalho} = \text{força} \times \text{distância}$$

2. Trabalho como integral da força:

$$\text{trabalho} = \int_{x_0}^{x_n} F(x)\,dx$$

3. Trabalho como integral da força com o ângulo:

$$\text{trabalho} = \int_{x_0}^{x_n} F(x)\cos(\theta(x))\,dx$$

Tabela de registros:

x (m)	$F(x)$ (N)	$\theta(x)$ (rad)	$F(x)\cos(\theta(x))$
0	0.0	0.50	0.0000
5	9.0	1.40	1.5297
10	13.0	0.75	9.5120
15	14.0	0.90	8.7025
20	10.5	1.30	2.8087
25	12.0	1.48	1.0881
30	5.0	1.50	0.3537

Calcule o Polinômio Interpolador

Para calcular o polinômio interpolador, utilizei o **Polinômio Interpolador de Lagrange**, dado pela fórmula abaixo:

$$P(x) = \sum_{i=0}^{n} y_i \prod_{j=0, j\neq i}^{n} \frac{x - x_j}{x_i - x_j}$$

Implementação em C:

Seja Fx o vetor que possui os valores da tabela de $F(x)cos(\theta(x))$ e x os valores da tabela de x(m):

```c
    double fx[] = {0.0000, 1.5297, 9.5120, 8.7025, 2.8087,
    1.0881, 0.3537};
    double x[] = {0, 5, 10, 15, 20, 25, 30};
    int n = sizeof(x) / sizeof(x[0]);
    double *Pn = malloc(sizeof(double) * n);

    for (int i = 0; i < n; i++) {
        double resultado = Pn_Lagrange(fx, x, n, x[i]);
        printf("Pn(%g) = %g\n", x[i], resultado);
        Pn[i] = resultado;
    }
```

Na parte acima,interppolamos a função $f(x)*cos(\theta(x))$ para os pontos dados em x(m) na tabela,, aproximando-a por um Polinômio de Lagrange.

O código abaixo mostra o código que interpolou cada ponto

```
// Calcula os Pn(X)s de Lagrange
double Pn_Lagrange(double fx[], double xA[], int n, double x
    ) {
    double resultado = 0;

    for (int i = 0; i < n; i++) {
        resultado += fx[i] * calculaLi(x, xA, i, n);
    }

    return resultado;
}
```

Função que calcula o $L_j(x) = \prod_{j=0, j\neq i}^{n} \frac{x-x_j}{x_i-x_j}$:

```
// Calcula os Li(X)s de Lagrange
double calculaLi(double x, double xA[], int i, int n) {
    double numerador = 1;
    double denominador = 1;

    for (int k = 0; k < n; k++) {
        if (i != k) {
            numerador *= x - xA[k];
            denominador *= xA[i] - xA[k];
        }
    }
    return numerador / denominador;
}
```

Os pontos retornados para x(m) da tabela foram

x	**Pn(x)**
0	0.0000
5	1.5297
10	9.5120
15	8.7025
20	2.8087
25	1.0881
30	0.3537

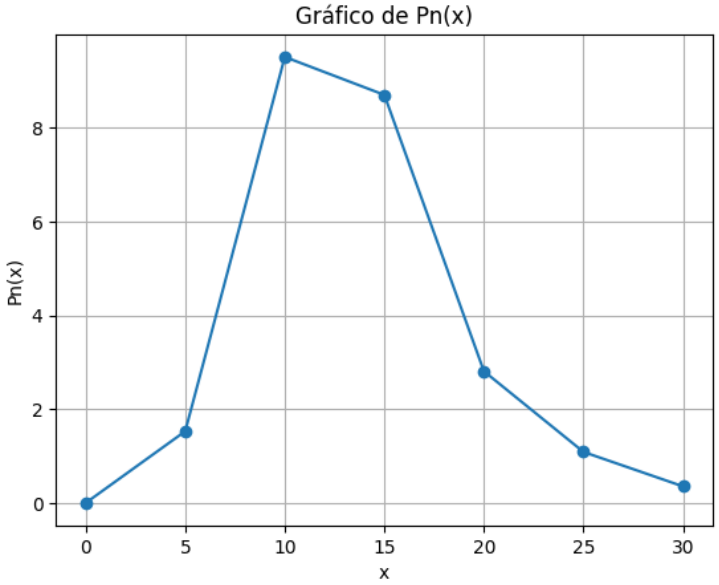

Figura 1: Descrição da figura

Regra do Trapézio composto

Considerando o tamanho dos sub-intervalos é constante = h. Assim, definimos h = (b - a) / N, onde N é o número de sub-intervalos (= número de nós - 1), e temos: xi = a + i h

Logo,

$$I(f) = \int_a^b f(x)\,dx = \sum_{i=1}^{N} \int_{x_{i-1}}^{x_i} f(x)\,dx$$

Aplicando para cada subintervalo, obtemos:

$$T_N(f) = \frac{h}{2} \sum_{i=1}^{N} (f(x_{i-1}) + f(x_i))$$

No código abaixo está a implementação:

```
    // Regra do Trap zio
double* regraDoTrapezio(double x[], double fx[], int n) {
    // a, b s o os pontos extremos
    double a = x[0];
    double b = x[n - 1];

    // h    o tamanho de cada n-1 subintervalos (
    equidistantes)
    double h = (b - a) / n;

```

```
    double* area = (double*)malloc((n - 1) * sizeof(double));
    // calcula a  rea  de cada subintervalo
    for (int i = 0; i < n - 1; i++) {
        area[i] = (h / 2) * (fx[i] + fx[i + 1]);
    }

    return area;
}
```

Os resultados:
Área do gráfico usando Trapézio:

Subintervalo	Valor
1	3.27793
2	23.6608
3	39.0311
4	24.6669
5	8.35029
6	3.08957

Soma da área total = Soma das áreas dos trapézio = 102.077 m^2
Regra de Simpson Composta
A regra de simpson composta é dada por:

$$S_N(f) = \frac{h}{3}\sum_{i=1}^{N/2}(f(x_{2i-2}) + 4f(x_{2i-1}) + f(x_{2i}))$$

Implementação em C:

```
    // Regra de Simpson
double* regraDeSimpson(double x[], double fx[], int n) {
    // a, b s o os pontos extremos
    double a = x[0];
    double b = x[n - 1];

    // h    o tamanho de cada n-1 subintervalos (equidistantes)
    double h = (b - a) / n;

    double* area = (double*)malloc((n - 1) * sizeof(double));
    // calcula a  rea  de cada par de subintervalos usando a regra de Simpson
    for (int k = 1; k < (n)/2; k++) {
        area[k] = (h / 3) * (fx[2*k-2] + 4 * fx[2*k - 1] + fx[2*k]);
    }

    return area;
}
```

As áreas calculadas para cada subintervalo:

Subintervalo	Área
1	1.13181×10^{-313}
3	67.3296
5	8.35029

Soma da área total = Soma das áreas das parábolas = 75.6799 m^2

2 Parte II: Interpolação por Monte Carlo (unidimensional e multidimensional

Para programar os casos pedidos de interpolação de monte carlos, as seguintes funções foram implementadas:

geradorPontosAleatorios

```
    double* geradorPontosAleatorios(double a, double b, int
    n) {
    srand(time(NULL));

    double* pontos = (double*)malloc(n * sizeof(double));

    for (int i = 0; i < n; i++) {
        pontos[i] = (((double)rand() / RAND_MAX) * (b - a))
    + a;
    }

    pontos[0] = a;
    pontos[n-1] = b;

    return pontos;
}
```

A função gera um array de n pontos aleatórios no intervalo [a, b], com os pontos extremos sendo a e b. Os pontos são gerados usando a função rand() para obter valores aleatórios e são armazenados em um array alocado dinamicamente.

calculaSenos

```
    double* calculaSenos(double pontos[], int n){
    double* senos = (double*)malloc(n * sizeof(double));

    for(int i = 0; i < n; i++){
        senos[i] = sin(pontos[i]);
    }

    return senos;
}
```

Recebe um array de pontos e seu tamanho n, e calcula os valores de sin(pontos[i]) para cada ponto do array. Os valores são armazenados em um novo array alocado dinamicamente, que é retornado pela função.

```
    double G(int i, double x) {
    switch (i) {
        case 1:
            return sin(x);
        case 2:
            return pow(x, 3);
        case 3:
            return exp(-x);
        default:
            return 0.0;  // Valor padr o caso i n o
   corresponda a nenhum caso
    }
}
```

Retorna a g correspondente a opção selecionada

```
double integracaoDeMonteCarlo(double a, double b, int n, int
    func) {
    double* pontos = geradorPontosAleatorios(a, b, n);
    double somaGs = 0;

    for (int i = 0; i < n; i++) {
        somaGs += G(func, pontos[i]);
    }

    double I = ((b - a)/n) * somaGs;

    free(pontos);
    return I;
}
```

A função calcula a soma dos valores de G para cada ponto e retorna o valor aproximado da integral usando a fórmula (b - a)/n * somaGs

Para aproximar Pi

A função aproxima o valor de usando o método de Monte Carlo. A função gera n pontos aleatórios no quadrado unitário e conta quantos pontos estão dentro do círculo unitário. Com base nessa proporção, a função calcula e retorna o valor aproximado de .

Para calcular a função em questão, utilizamos o que estava no pdf do enunciado no qual dizia que

$_n = \frac{1}{n} \sum_{i=1}^{n} g(U_{i1}, U_{i2}, ..., U_{id})$

Pelo enunciado, $\int_0^1 \int_0^1 g(x, y)\, dx\, dy$

pode ser aproximada pelo somatório das variáveis aleatórias de g(x, y) dividido por n e sabemos

g(x, y) = $\begin{cases} 1, & \text{se } x^2 + y^2 \leq 1 \\ 0, & \text{caso contrário} \end{cases}$

Aproximei as duas integrais pelo somatório das variáveis aleatórias dividido por n para n.

Implementação:

```
    double aproximaPi(int n) {
    int pontosDentroCirculo = 0;

    for (int i = 0; i < n; i++) {
        double x = (double)rand() / RAND_MAX;
        double y = (double)rand() / RAND_MAX;
        if (x*x + y*y <= 1) {
            pontosDentroCirculo++;
        }
    }

    double piAproximado = 4.0 * pontosDentroCirculo / n;

    return piAproximado;
}
```

Testes

A seguir segue o menu de opções:

```
        switch (opcao) {
        case 1:
            resultado = integracaoDeMonteCarlo(0, 1, n, 1);
            break;
        case 2:
            resultado = integracaoDeMonteCarlo(3, 7, n, 2);
            break;
        case 3:
            resultado = integracaoDeMonteCarlo(0, log(
    DBL_MAX) / log(M_E), n, 3);
            break;
        case 4:
            resultado = aproximaPi(n);
            break;
        default:
            printf("Op   o inv lida!\n");
            return 0;
    }
```

Obs: Para calcular o exercício 3, parti do principio de que é praticamente impossível calcular a integral de [0, INFINITY] em C e isso gera uma quebra devido a uma limitação de natureza própria do computador. Portanto, in a integral partindo do principio:

$$e^x = \text{MAX} \Rightarrow \ln(e^x) = \ln(\text{MAX}) \Rightarrow A = \pi r^2 = \pi \left(\frac{D}{2}\right)^2 = \frac{\pi D}{4} = \pi = 4A$$

A variável $\text{DLB}_MAX correspondeaomaximovalorquepossvelreprensentaremfloat.$

Resultados: Caso 1: Para n = 10000 = O resultado da integração é: 0.456428
Caso 2: Para n = 10000 = O resultado da integração é: 583.021908
Caso 3: Para n = 10000 = O resultado da integração é: 1.061474
Cas0 4: Para n = 10000 = O resultado da integração é: 3.171200