Universidade Estadual de Santa Cruz – UESC

Relatórios II de Implementações de Métodos da Disciplina Análise Numérica

Relatório de implementações realizadas por Gabriel Rosa Galdino

Disciplina Análise Numérica.

Curso Ciência da Computação

Semestre 2025.2

Professor Gesil Sampaio Amarante II

Ilhéus – BA
2025

ÍNDICE

Lista de Figuras---3
Linguagem Escolhida e justificativas--- 4
Regressão Linear---5
Estratégia de Implementação-- 5
Estrutura dos Arquivos de Entrada/Saída--- 5
Dificuldades enfrentadas-- 6
Aproximação Polinomial Discreta-- 7
Estratégia de Implementação-- 7
Estrutura dos Arquivos de Entrada/Saída--- 7
Dificuldades enfrentadas-- 8
Aproximação Polinomial Contínua-- 9
Estratégia de Implementação-- 9
Estrutura dos Arquivos de Entrada/Saída--- 9
Dificuldades enfrentadas-- 10
Problemas Estabelecidos-- 11
Problemas para regressão linear e aproximação polinomial---------------------------- 11
Interpolação por polinômios de Lagrange--- 14
Estratégia de Implementação-- 14
Estrutura dos Arquivos de Entrada/Saída--- 14
Dificuldades enfrentadas-- 15
Interpolação Polinomial por Diferenças Divididas de Newton------------------------ 16
Estratégia de Implementação-- 16
Estrutura dos Arquivos de Entrada/Saída--- 16
Problemas Estabelecidos-- 18
Problemas para interpolação polinomial--- 18
Derivação Numérica de 1ª Ordem-- 20
Estratégia de Implementação-- 20
Estrutura dos Arquivos de Entrada/Saída--- 20
Dificuldades enfrentadas-- 21
Derivação Numérica de 2ª Ordem-- 22
Estratégia de Implementação-- 22
Estrutura dos Arquivos de Entrada/Saída--- 22
Dificuldades enfrentadas-- 23
Problemas Estabelecidos-- 24
Problemas para derivação numérica--- 24
Exercícios--- 24
11.1-- 24
11.6-- 24
11.11--- 24
Integração por Trapézios Simples-- 27
Estratégia de Implementação-- 27

Estrutura dos Arquivos de Entrada/Saída---27
Dificuldades enfrentadas---28
Integração por Trapézios Múltiplos---29
Estratégia de Implementação---29
Estrutura dos Arquivos de Entrada/Saída---29
Dificuldades enfrentadas---29
Integração por Simpson 1/3 Simples---30
Estratégia de Implementação---30
Estrutura dos Arquivos de Entrada/Saída---30
Dificuldades enfrentadas---30
Integração por Simpson 1/3 Múltipla---31
Estratégia de Implementação---31
Estrutura dos Arquivos de Entrada/Saída---31
Dificuldades enfrentadas---31
Integração por Simpson 3/8 Simples---32
Estratégia de Implementação---32
Estrutura dos Arquivos de Entrada/Saída---32
Dificuldades enfrentadas---32
Integração por Simpson 3/8 Múltipla---33
Estratégia de Implementação---33
Estrutura dos Arquivos de Entrada/Saída---33
Dificuldades enfrentadas---33
Extrapolação de Richards--- 34
Estratégia de Implementação---34
Estrutura dos Arquivos de Entrada/Saída---34
Dificuldades enfrentadas---34
Quadratura de Gauss---35
Estratégia de Implementação---35
Estrutura dos Arquivos de Entrada/Saída---35
Dificuldades enfrentadas---36
Problemas Estabelecidos---37
Problemas para integração numérica---37
Exercícios---37
11.1---37
11.6---37
11.11---37
Considerações finais---44

Lista de Figuras

Figura 1 - Exemplo de Saída: Regressão Linear.. 6
Figura 2 - Exemplo de Saída: Aproximação Polinomial Discreta............................ 8
Figura 3 - Exemplo de Saída: Aproximação Polinomial Contínua..........................10
Figura 4 - Entrada: Problemas de Regressão e Aproximação Discreta................ 11
Figura 5 - Saída: Problemas de Regressão Linear... 11
Figura 6 - Saída: Problemas de Aproximação Polinomial Discreta........................12
Figura 7 - Entrada: Problemas de Aproximação Polinomial Contínua..................12
Figura 8 - Saída: Problemas de Aproximação Polinomial Contínua.......................13
Figura 9 - Entrada: Problemas de Interpolação Polinomial....................................18
Figura 10 - Saída: Problemas de Interpolação (Polinômios de Lagrange).............18
Figura 11 - Saída: Problemas de Interpolação (Diferenças Divididas de Newton) 19
Figura 12 - Entrada: Problemas de Derivação Numérica..25
Figura 13 - Saída: Problemas de Derivação Numérica (1ª Ordem)......................... 25
Figura 14 - Saída: Problemas de Derivação Numérica (2ª Ordem)......................... 26
Figura 15 - Entrada: Problemas de Integração Numérica.......................................38
Figura 16 - Saída: Problemas de Integração (Regra do Trapézio Simples)............38
Figura 17 - Saída: Problemas de Integração (Regra dos Trapézios Múltiplos)......39
Figura 18 - Saída: Problemas de Integração (Simpson 1/3 Simples)......................39
Figura 19 - Saída: Problemas de Integração (Simpson 1/3 Múltipla)......................40
Figura 20 - Saída: Problemas de Integração (Simpson 3/8 Simples)......................41
Figura 21 - Saída: Problemas de Integração (Simpson 3/8 Múltipla)......................42
Figura 22 - Saída: Problemas de Integração (Extrapolação de Richards)............. 43
Figura 23 - Saída: Problemas de Integração (Quadratura de Gauss).....................43

Linguagem Escolhida e justificativas

A linguagem Python foi a selecionada para este trabalho devido à sua simplicidade e ao seu poderoso ecossistema para computação científica, o que permitiu focar mais na lógica dos métodos do que em complexidades de programação.

A escolha é justificada principalmente pelo uso de bibliotecas essenciais como o NumPy, que oferece uma base sólida para operações matemáticas de alta performance. Para a visualização e análise dos dados, foi fundamental o uso do Matplotlib. Além disso, a biblioteca SciPy foi de grande ajuda, pois já contém implementações de diversos algoritmos numéricos, o que facilitou a verificação e a aplicação prática dos métodos abordados na disciplina.

Dessa forma, o Python se mostrou a escolha ideal para os objetivos deste relatório, aliando um código legível a um ambiente computacional robusto, o que tornou a implementação dos métodos uma tarefa prática e didática.

Regressão Linear

Estratégia de Implementação

Para a Regressão Linear, implementei o método dos mínimos quadrados na sua forma analítica direta, buscando a reta y=ax+by = ax + by=ax+b que melhor se ajusta aos dados. A estratégia, contida na função **linear_regression** do módulo **metodos.py**, foi focada na eficiência e simplicidade.

Diferente de métodos que montam um sistema linear, minha implementação calcula diretamente os coeficientes aaa e bbb usando suas fórmulas fechadas, que dependem de quatro somatórios principais: $\sum x$, $\sum y$, $\sum xy$ e $\sum x^2$, além do número de pontos nn.
O código calcula essas somas iterando pelas listas values_x e values_y e, em seguida, aplica às fórmulas:

- $b = (n \cdot \sum xy - \sum x \cdot \sum y)/(n \cdot \sum x^2 - (\sum x)^2)$
- $a = (\sum y - b \cdot \sum x)/n$

O **core.py** simplesmente chama essa função para cada conjunto de dados e armazena os coeficientes **a** e **b** retornados, que já são arredondados para duas casas decimais no próprio **metodos.py** para clareza na saída.

Estrutura dos Arquivos de Entrada/Saída

A estrutura foi projetada para ser simples e compartilhada com outros métodos discretos.

- **Arquivo de Entrada (ex: entrada_aprox_regr.txt)** O programa, através do construtor da classe **CalculadorAproximacao** em core.py, é inicializado com o method_type="discreto". Ele espera um arquivo de texto no diretório Input/ (considerando a nossa refatoração). Cada linha do arquivo representa um conjunto de dados independente, no formato:
  valores_de_x_separados_por_virgula ;
  valores_de_y_separados_por_virgula
  Exemplo de linha: 1, 2, 3, 4, 5 ; 3.1, 4.9, 7.1, 8.8, 11.2
- **Arquivo de Saída (regressao_linear_saida.txt)** Após a execução, um relatório é salvo em output/. A função gerar_relatorio_regressao_linear (de relatorios.py) formata a saída, apresentando a reta de regressão para cada caso:

```
--- Relatório de Regressão Linear ---

Caso 1: y = 362.49*x - 709699.53
Caso 2: y = -16.3*x + 33922.56
Caso 3: y = 0.39*x + 0.53
Caso 4: y = 37.38*x - 73791.84
Caso 5: y = 22.05*x - 43319.83
Caso 6: y = 0.2*x + 1.33
Caso 7: y = 0.01*x - 0.62
Caso 8: y = 1.86*x - 1.85

--------------------------------------
Tempo de execucao: 0.000053 segundos
```

Figura 1 - Exemplo de Saída: Regressão Linear

Dificuldades enfrentadas

A maior dificuldade encontrada neste método não foi computacional, mas sim na formatação da saída. O cálculo direto dos coeficientes a e b é muito robusto. No entanto, se o coeficiente a (intercepto) fosse negativo, uma formatação de string simples resultaria em uma saída deselegante, como y = 2.5*x + -1.3.

Para solucionar isso, implementei uma lógica condicional diretamente no módulo relatorios.py (função gerar_relatorio_regressao_linear):

result_str = f"y = {b}*x + {a}" if a >= 0 else f"y = {b}*x - {abs(a)}"

Essa verificação simples garante que o sinal de a seja tratado corretamente, produzindo uma equação limpa (y = 2.5*x - 1.3), o que torna o relatório final muito mais profissional e legível.

Aproximação Polinomial Discreta

Estratégia de Implementação

Para a Aproximação Polinomial Discreta, a estratégia adotada foi a resolução de um sistema de equações lineares para encontrar os coeficientes do polinômio. Na minha implementação em **metodos.py**, decidi fixar a base de funções em um polinômio de grau 2, definido por basis_functions = [1, x, x**2], onde x é um Symbol do SymPy.

O método, discrete_polynomial_approximation, constrói a matriz M (no código, matrix_M) e o vetor de resultados F (vector_F) do sistema M * c = F, onde c são os coeficientes a, b, c do polinômio y = a + bx + cx^2.

1. Primeiro, criei uma matriz matrix_Ui que armazena os valores de cada função da base ϕ_j aplicados a cada ponto x_i.
2. Em seguida, construí M e F usando suas definições, onde $M_{ij} = \sum(\phi_i(x_k) \cdot \phi_j(x_k))$ e $F_i = \sum(y_k \cdot \phi_i(x_k))$. Para os produtos, utilizei np.multiply para garantir a multiplicação elemento a elemento antes da soma.
3. Finalmente, para encontrar os coeficientes, utilizei a função np.linalg.solve(matrix_M, vector_F), que resolve o sistema linear de forma eficiente e estável. A expressão polinomial resultante é então montada usando sympy.expand.

Estrutura dos Arquivos de Entrada/Saída

Arquivo de Entrada (ex: entrada_aprox_regr.txt) Este método reutiliza exatamente a mesma estrutura de entrada da Regressão Linear. O core.py é chamado com method_type="discreto" e lê o arquivo do diretório Input/ no formato: valores_de_x_separados_por_virgula ; valores_de_y_separados_por_virgula

Arquivo de Saída (aprox_polinomial_discreta_saida.txt) O relatório gerado é salvo em output/ com um nome de arquivo específico. A função gerar_relatorio_aprox_polinomial (de relatorios.py) formata a saída:

```
--- Relatório de Aproximação Polinomial Discreta ---

Função 1: y = 6.46*x**2 - 25324.37*x + 24834825.76
Função 2: y = 1.15*x**2 - 4591.12*x + 4583394.77
Função 3: y = 0.56*x**2 - 0.23*x + 0.65
Função 4: y = 0.63*x**2 - 2450.44*x + 2399726.05
Função 5: y = 0.4*x**2 - 1549.6*x + 1519295.77
Função 6: y = -0.04*x**2 + 1.0*x
Função 7: y = 0.440000000000000
Função 8: y = 0.8*x**2 - 1.73*x + 0.54

----------------------------------------------------
Tempo de execucao: 0.088419 segundos
```

Figura 2 - Exemplo de Saída: Aproximação Polinomial Discreta

Dificuldades enfrentadas

O principal desafio na implementação foi a correta construção da matriz M e do vetor F. A lógica teórica envolve somatórios de produtos de funções da base, o que pode ser confuso de traduzir para código vetorial.

A minha solução foi usar um laço duplo em metodos.py para iterar sobre i e j (os índices das funções da base). Dentro desses laços, utilizei np.multiply(matrix_Ui[i], matrix_Ui[j]) para obter o produto elemento a elemento dos valores das funções ϕ_i e ϕ_j em todos os pontos x_k, e então apliquei sum() nesse resultado para obter o valor de M_{ij}. Essa abordagem evitou um terceiro laço interno (sobre os pontos x_k), tornando a construção do sistema mais limpa e eficiente.

Aproximação Polinomial Contínua

Estratégia de Implementação

A estratégia para a Aproximação Contínua é similar à Discreta (resolver um sistema M * c = F), mas com uma diferença fundamental: os coeficientes M_{ij} e F_i são definidos por **integrais** no intervalo [a, b], e não por somatórios.

Dada essa natureza, tomei a decisão de usar a biblioteca SymPy como motor principal deste método. Em metodos.py, na função continuous_polynomial_approximation:

1. Defini a base de funções simbólicas: basis_functions = [1, x, x**2].
2. A função de entrada (uma string) é convertida para um objeto SymPy usando sympify.
3. As matrizes matrix_M e vector_F são inicializadas como matrizes simbólicas com sympy.zeros.
4. O núcleo do método está em um laço que calcula cada elemento $M_{ij} = \int_a^b \phi_i(x)\phi_j(x)dx$ e $F_i = \int_a^b f(x)\phi_i(x)dx$. Utilizei a função sympy.integrate para realizar esses cálculos de forma simbólica, garantindo precisão máxima.
5. Em vez de usar np.linalg.solve, utilizei o método matrix_M.LUsolve(vector_F) do SymPy, que resolve o sistema linear simbólicamente.

Estrutura dos Arquivos de Entrada/Saída

Arquivo de Entrada (ex: input_continuos_approximation.txt) Para este método, o core.py é chamado com method_type="continuo". O arquivo de entrada em Input/ possui um formato diferente: funcao_em_string ; limite_inferior,limite_superior
Exemplo de linha:

1/(((x)**(5))*((e)**(1.432/2000)-1));0.00004,0.00007

Arquivo de Saída (aprox_polinomial_continua_saida.txt) O relatório é salvo em output/. A mesma função gerar_relatorio_aprox_polinomial (de relatorios.py) é usada para formatar a saída:

```
--- Relatório de Aproximação Polinomial Contínua ---

Função 1: y = 1.87392978527453e+34*x**2 - 2.40998727536488e+30*x + 7.85176580029825e+25
Função 2: y = 2.06138985922159e+34*x**2 - 2.65107229167738e+30*x + 8.63722351013628e+25
Função 3: y = 2.10825487791065e+34*x**2 - 2.71134354601565e+30*x + 8.83358793844325e+25
Função 4: y = 2.248849934383e+34*x**2 - 2.89215730955158e+30*x + 9.42268122506223e+25
Função 5: y = 2.34257997232728e+34*x**2 - 3.01269981895626e+30*x + 9.81541008404865e+25
Função 6: y = 2.4363100104767e+34*x**2 - 3.13324232862478e+30*x + 1.02081389438946e+26
Função 7: y = 2.57690506803775e+34*x**2 - 3.31405609356084e+30*x + 1.07972322350752e+26
Função 8: y = 2.62377008730475e+34*x**2 - 3.37432734864242e+30*x + 1.09935966658041e+26
Função 9: y = 2.81123016471919e+34*x**2 - 3.61541236941416e+30*x + 1.17790543901703e+26

--------------------------------------------------
Tempo de execucao: 0.600335 segundos
```

Figura 3 - Exemplo de Saída: Aproximação Polinomial Contínua

Dificuldades enfrentadas

A maior dificuldade foi lidar com a conversão da função de entrada (string) em um objeto SymPy funcional, especialmente quando a função continha constantes como ***e***. Os arquivos de teste usavam math.e, que não é reconhecido pelo sympify do SymPy.

Para resolver isso, implementei uma etapa de pré-processamento da string de entrada em metodos.py, como discutimos anteriormente. O código:

- processed_str = func_str.replace('^', '**').replace('math.e', 'e')
- locals_dict = { 'x': x,'sin': np.sin,'cos': np.cos,'tan': np.tan,'exp': np.exp,'log': np.log,'sqrt': np.sqrt,'pi': np.pi,'e': np.e,'abs': abs,'pow': pow, }
- func = sympify(processed_str, locals=locals_dict)

Essa solução "limpa" a string, substituindo **np.e** por **e**, e então informa ao sympify (através do locals_dict) que o símbolo "e" deve ser tratado como a constante de Euler do SymPy (sympy.E). Isso tornou o método robusto o suficiente para lidar com os arquivos de entrada complexos que estávamos usando para teste.

Problemas Estabelecidos

Problemas para regressão linear e aproximação polinomial

Com o objetivo de tornar os testes mais eficientes, cada método foi configurado para processar diversos exercícios de uma só vez, sendo que cada linha do arquivo de entrada representa um caso distinto.

As entradas disponibilizadas estão vinculadas, na mesma ordem, às questões 8.1 (a), 8.1 (b), 8.5, 8.11 (a), 8.11 (b), 10.2, 10.6 e 10.9 da lista em respectivo os números de 1 a 8 da entrada abaixo.

Entrada:

```
Relatorio_02 > Codigo > Input > entrada_aprox_regr2.txt
You, 4 hours ago | 1 author (You)
1  1980,1985,1990,1993,1994,1996,1998;8300,9900,10400,13200,13600,13700,14600
2  1980,1985,1990,1993,1994,1996,1998;1688,1577,1397,1439,1418,1385,1415
3  0.1,0.2,0.3,0.4,0.5,0.6,0.7,0.8,0.9,1;0.62,0.63,0.64,0.66,0.68,0.71,0.76,0.81,0.89,1
4  1980,1985,1989,1992,1994,1995,1997;248.8,398,503.7,684.9,749.9,793.5,865.7
5  1980,1985,1989,1992,1994,1995,1997;355.3,438,487.7,617.8,658.1,674.6,707.6
6  0,10,20;0,6,4
7  500,1000,1500,2000,2500,3000,3500,4000;2.74,5.48,7.90,11.00,13.93,16.43,20.24,23.52
8  0.0,0.5,1.0,1.5,2.0,2.5,3.0,3.5,4.0,4.5;0.0,0.0125,0.06,0.195,0.5,1.0875,2.1,3.71,6.12,9.56
```

Figura 4 - Entrada: Problemas de Regressão e Aproximação Discreta

Saídas:

```
--- Relatório de Regressão Linear ---

Caso 1: y = 362.49*x - 709699.53
Caso 2: y = -16.3*x + 33922.56
Caso 3: y = 0.39*x + 0.53
Caso 4: y = 37.38*x - 73791.84
Caso 5: y = 22.05*x - 43319.83
Caso 6: y = 0.2*x + 1.33
Caso 7: y = 0.01*x - 0.62
Caso 8: y = 1.86*x - 1.85

------------------------------------
Tempo de execucao: 0.000053 segundos
```

Figura 5 - Saída: Problemas de Regressão Linear

```
--- Relatório de Aproximação Polinomial Discreta ---

Função 1: y = 6.46*x**2 - 25324.37*x + 24834825.76
Função 2: y = 1.15*x**2 - 4591.12*x + 4583394.77
Função 3: y = 0.56*x**2 - 0.23*x + 0.65
Função 4: y = 0.63*x**2 - 2450.44*x + 2399726.05
Função 5: y = 0.4*x**2 - 1549.6*x + 1519295.77
Função 6: y = -0.04*x**2 + 1.0*x
Função 7: y = 0.440000000000000
Função 8: y = 0.8*x**2 - 1.73*x + 0.54

---------------------------------------------------
Tempo de execucao: 0.081988 segundos
```

Figura 6 - Saída: Problemas de Aproximação Polinomial Discreta

Para a aplicação do método de aproximação polinomial contínua, foi utilizado o problema 11.1.

Entrada:

```
Relatorio_02 > Codigo > Input > entrada_aprox_regr.txt
You, 28 seconds ago | 1 author (You)
1/(((x)**(5))*((e)**(1.432/2000)-1));0.00004,0.00007
1/(((x)**(5))*((e)**(1.432/2200)-1));0.00004,0.00007
1/(((x)**(5))*((e)**(1.432/2250)-1));0.00004,0.00007
1/(((x)**(5))*((e)**(1.432/2400)-1));0.00004,0.00007
1/(((x)**(5))*((e)**(1.432/2500)-1));0.00004,0.00007
1/(((x)**(5))*((e)**(1.432/2600)-1));0.00004,0.00007
1/(((x)**(5))*((e)**(1.432/2750)-1));0.00004,0.00007
1/(((x)**(5))*((e)**(1.432/2800)-1));0.00004,0.00007
1/(((x)**(5))*((e)**(1.432/3000)-1));0.00004,0.00007
```

Figura 7 - Entrada: Problemas de Aproximação Polinomial Contínua

Saída:

```
--- Relatório de Aproximação Polinomial Contínua ---

Função 1: y = 1.87392978527453e+34*x**2 - 2.40998727536488e+30*x + 7.85176580029825e+25
Função 2: y = 2.06138985922159e+34*x**2 - 2.65107229167738e+30*x + 8.63722351013628e+25
Função 3: y = 2.10825487791065e+34*x**2 - 2.71134354601565e+30*x + 8.83358793844325e+25
Função 4: y = 2.248849934383e+34*x**2 - 2.89215730955158e+30*x + 9.42268122506223e+25
Função 5: y = 2.34257997232728e+34*x**2 - 3.01269981895626e+30*x + 9.81541008404865e+25
Função 6: y = 2.4363100104767e+34*x**2 - 3.13324232862478e+30*x + 1.02081389438946e+26
Função 7: y = 2.57690506803775e+34*x**2 - 3.31405609356084e+30*x + 1.07972322350752e+26
Função 8: y = 2.62377008730475e+34*x**2 - 3.37432734864242e+30*x + 1.09935966658041e+26
Função 9: y = 2.81123016471919e+34*x**2 - 3.61541236941416e+30*x + 1.17790543901703e+26

--------------------------------------------------
Tempo de execucao: 0.617481 segundos
```

Figura 8 - Saída: Problemas de Aproximação Polinomial Contínua

Interpolação por polinômios de Lagrange

Estratégia de Implementação

Para a implementação do método de Interpolação de Lagrange, adotei uma abordagem puramente simbólica, alinhada com a formulação matemática clássica. A estratégia, contida na função lagrange_polynomial do módulo metodos.py, foca em construir um polinômio único que passa exatamente por todos os pontos de dados fornecidos. O núcleo da minha implementação utiliza a biblioteca SymPy. Iniciei definindo x = Symbol('x') para que todas as operações subsequentes fossem tratadas algebricamente.

O processo iterativo consiste em um laço principal que percorre cada ponto i (de 0 a n - 1), e um laço aninhado que percorre cada ponto j para construir o polinômio de base $L_i(x)$. Este termo L_i é um produto simbólico dos fatores $\frac{x - x_j}{x_i - x_j}$ para todos os j diferentes de i. Após construir cada $L_i(x)$ simbolicamente, ele é multiplicado pelo seu valor y_i correspondente e somado ao polinômio total, lagrange_poly. Finalmente, para obter a expressão algébrica simplificada que é apresentada no relatório, apliquei a função sympy.expand() sobre o polinômio final. Esta abordagem, embora computacionalmente intensiva, garante a precisão algébrica absoluta do resultado.

Estrutura dos Arquivos de Entrada/Saída

A estrutura de I/O foi desenhada para ser flexível, modular e capaz de processar múltiplos conjuntos de dados em uma única execução.

- **Arquivo de Entrada (ex: entrada_interp.txt)** O programa, através do construtor da classe CalculadorInterpolacao em core.py, espera um arquivo de texto localizado no diretório Input/ (considerando a nossa refatoração). Cada linha do arquivo representa um conjunto independente de pontos a serem interpolados. O formato de cada linha é: x1,x2,x3,... ; y1,y2,y3,...

  O código em core.py primeiro realiza um split(";") para separar os eixos X e Y, e em seguida utiliza list(map(float, ...split(","))) para converter as strings de valores em listas de números de ponto flutuante, que são armazenadas em self.data_sets.
- **Arquivo de Saída (interpolacao_lagrange_saida.txt)** Após a execução da função run_lagrange(), um relatório detalhado é gerado no diretório output/. A função gerar_relatorio_interpolacao (do módulo relatorios.py)

é chamada para formatar este arquivo. A saída inclui um cabeçalho identificando o método, seguido por cada conjunto de dados interpolado no formato P(x): {poly_expr}, onde {poly_expr} é a expressão simbólica expandida retornada pelo SymPy. Ao final do arquivo, um resumo conciso apresenta o tempo total de execução.

Dificuldades enfrentadas

Durante a implementação do método de Lagrange, a principal dificuldade foi traduzir a fórmula matemática, que utiliza um produtório (Π), para uma estrutura de código eficiente. A construção simbólica de cada termo $L_i(x)$ dentro de um laço aninhado é um processo que cresce em complexidade. O resultado simbólico, antes da simplificação, é uma expressão muito longa e complexa, composta por múltiplos fatores (ex: $y_0 \bullet \frac{(x - x_1)(x - x_2)}{...} + ...$). A solução para tornar esse resultado legível e utilizável foi a aplicação da função **sympy.expand()** no final do processo, sobre o **lagrange_poly** completo. Isso força o SymPy a realizar toda a multiplicação e soma algébrica, simplificando o polinômio para sua forma canônica (ex: $ax^2 + bx + c$). Embora essa expansão possa ser computacionalmente cara para um número muito grande de pontos, foi a solução essencial para garantir que o relatório de saída fosse claro e preciso.

Interpolação Polinomial por Diferenças Divididas de Newton

Estratégia de Implementação

A estratégia que adotei para o método de Newton por Diferenças Divididas, contida em metodos.py, foi dividida em duas fases claras. A primeira fase é puramente numérica e visa construir a tabela de diferenças divididas. Para isso, inicializei uma matriz N x N utilizando np.zeros((n, n)) da biblioteca NumPy. A primeira coluna desta matriz, diferencas_div[:, 0], é preenchida diretamente com os y_values. Em seguida, um laço duplo (j e i) preenche o restante da tabela aplicando a fórmula recursiva das diferenças divididas, onde cada elemento depende de dois elementos da coluna anterior e dos valores de x correspondentes.

A segunda fase é a construção simbólica do polinômio. Utilizando SymPy, iniciei o polinômio (newton_poly) com o primeiro coeficiente, C_0, que é diferencas_div[0][0]. Depois, um laço (j de 1 a n - 1) iterou pelos coeficientes restantes (a diagonal principal da tabela). Dentro deste laço, um laço aninhado (i) foi responsável por construir o termo do produtório Π(x - x_i) simbolicamente, armazenando-o na variável termo. Cada termo completo C_j * Π(x - x_i) foi então somado ao newton_poly. Assim como em Lagrange, o polinômio final foi passado pela função **sympy.expand()** para garantir um resultado simplificado.

Estrutura dos Arquivos de Entrada/Saída

A estrutura de I/O para este método é idêntica à do método de Lagrange, o que garante a consistência da ferramenta e facilita testes comparativos.

- **Arquivo de Entrada (ex: entrada_interp.txt)** O programa lê o mesmo arquivo de Input/ no formato x1,x2,... ; y1,y2,.... A função run_newton() em core.py é responsável por orquestrar esse método.
- **Arquivo de Saída (interpolacao_newton_saida.txt)** O relatório gerado é salvo em output/, mas com seu próprio nome de arquivo para se diferenciar de Lagrange. A formatação, controlada pela mesma função gerar_relatorio_interpolacao, é idêntica:
  --- Relatório de Interpolação por Diferenças Divididas de Newton ---
  P(x): {poly_expr} ... E o sumário de tempo ao final.

Dificuldades enfrentadas

Na implementação do método de Newton, a primeira dificuldade foi a construção correta da tabela de diferenças divididas. O algoritmo exige atualizações sucessivas e um controle rigoroso dos índices i e j nos laços aninhados para garantir que a fórmula recursiva **(diferencas_div[i + 1][j - 1] - diferencas_div[i][j - 1]) / (x_values[i + j] - x_values[i])** fosse aplicada corretamente, sem erros de IndexError.

A segunda dificuldade foi a construção do polinômio final. O termo geral envolve um produtório acumulado. Gerenciar isso simbolicamente exigiu um laço aninhado (i) apenas para construir a parte do termo (ex: (x - x_0)(x - x_1)) antes de multiplicá-lo pelo seu coeficiente C_j (ex: diferencas_div[0][j]). Assim como em Lagrange, a expressão resultante é complexa, e o uso de sympy.expand foi a solução crucial para apresentar um resultado legível e quimicamente simplificado, em vez da forma de Newton não expandida.

Problemas Estabelecidos

Problemas para interpolação polinomial

Com o objetivo de tornar os testes mais eficientes, assim como o de regressão linear e aproximação linear, cada método foi configurado para processar diversos exercícios de uma só vez, sendo que cada linha do arquivo de entrada representa um caso distinto.

As entradas disponibilizadas estão vinculadas, na mesma ordem, às questões 8.1 (a), 8.1 (b), 8.5, 8.11 (a), 8.11 (b), 10.2, 10.6 e 10.9 da lista em respectivo os números de 1 a 8 da entrada abaixo.

Entrada:

```
Relatorio_02 > Codigo > Input > entrada_inter.txt
1  1980,1985,1990,1993,1994,1996,1998;8300,9900,10400,13200,13600,13700,14600
2  1980,1985,1990,1993,1994,1996,1998;1688,1577,1397,1439,1418,1385,1415
3  0.1,0.2,0.3,0.4,0.5,0.6,0.7,0.8,0.9,1;0.62,0.63,0.64,0.66,0.68,0.71,0.76,0.81,0.89,1
4  1980,1985,1989,1992,1994,1995,1997;248.8,398,503.7,684.9,749.9,793.5,865.7
5  1980,1985,1989,1992,1994,1995,1997;355.3,438,487.7,617.8,658.1,674.6,707.6
6  0,10,20;0,6,4
7  500,1000,1500,2000,2500,3000,3500,4000;2.74,5.48,7.90,11.00,13.93,16.43,20.24,23.52
8  0.0,0.5,1.0,1.5,2.0,2.5,3.0,3.5,4.0,4.5;0.0,0.0125,0.06,0.195,0.5,1.0875,2.1,3.71,6.12,9.5625
```

Figura 9 - Entrada: Problemas de Interpolação Polinomial

Saídas:

```
--- Relatório de Interpolação por Polinômios de Lagrange ---

P(x) 1: -0.00782550782550784*x**6 + 93.7477661227567*x**5 - 467942.621017873*x**4
+ 1245711723.98248*x**3 - 1865346849814.56*x**2 + 1.48968679672157e+15*x - 4.95694064855089e+17
P(x) 2: -0.00316693722943724*x**6 + 37.8361334498825*x**5 - 188348.344876891*x**4
+ 500051348.857959*x**3 - 746773557147.295*x**2 + 594786350415286.0*x - 1.97388032787121e+17
P(x) 3: -1267.63668430332*x**9 + 5877.97619047639*x**8 - 11554.2328042328*x**7 + 12565.9722222232*x**6
- 8280.43981481541*x**5 + 3405.19097222285*x**4 - 867.43970458578*x**3 + 131.185615079406*x**2 - 10.5359920634942*x + 0.960000000000058
P(x) 4: -0.00454200383857244*x**6 + 54.2344524976658*x**5 - 269830.487356718*x**4
+ 715986342.85605*x**3 - 1068661639546.27*x**2 + 850694576518930.0*x - 2.82159774250353e+17
P(x) 5: -0.00198929479199086*x**6 + 23.7594855469964*x**5 - 118239.569831214*x**4
+ 313824278.468312*x**3 - 468524088677.756*x**2 + 373057003992297.0*x - 1.23767434537114e+17
P(x) 6: -0.04*x**2 + 1.0*x
P(x) 7: -1.51365079365078e-22*x**7 + 2.02755555555548e-18*x**6 - 1.05728888888887e-14*x**5
+ 2.69655555555551e-11*x**4 - 3.42985555555549e-8*x**3 + 1.909188888889e-5*x**2 + 0.00309780952380935*x - 0.679999999999939
P(x) 8: -8.88178419700125e-16*x**8 + 1.4210854715202e-14*x**7 - 5.6843418860808e-14*x**6
+ 1.13686837721616e-13*x**5 + 0.0199999999999818*x**4 + 0.010000000000332*x**3 + 0.0200000000000387*x**2 + 0.0100000000000051*x

------------------------------------------------
Tempo de execucao: 0.526057 segundos
```

Figura 10 - Saída: Problemas de Interpolação (Polinômios de Lagrange)

```
--- Relatório de Interpolação por Diferenças Divididas de Newton ---

P(x) 1: -0.00782550782550783*x**6 + 93.7477661227662*x**5 - 467942.621017871*x**4 + 1245711723.98242*x**3
- 1865346849814.38*x**2 + 1.48968679672156e+15*x - 4.95694064855026e+17
P(x) 2: -0.00316693722943723*x**6 + 37.8361334498834*x**5 - 188348.344876894*x**4 + 500051348.857959*x**3
- 746773557147.282*x**2 + 594786350415293.0*x - 1.97388032787114e+17
P(x) 3: -1267.63668430338*x**9 + 5877.97619047632*x**8 - 11554.2328042331*x**7 + 12565.9722222225*x**6
- 8280.43981481504*x**5 + 3405.19097222233*x**4 - 867.439704585567*x**3 + 131.18561507937*x**2 - 10.5359920634925*x + 0.960000000000016
P(x) 4: -0.00454200383857247*x**6 + 54.2344524976658*x**5 - 269830.487356715*x**4 + 715986342.856042*x**3
- 1068661639546.27*x**2 + 850694576518931.0*x - 2.8215977425035e+17
P(x) 5: -0.00198929479199086*x**6 + 23.7594855469965*x**5 - 118239.569831213*x**4 + 313824278.46831*x**3
- 468524088677.757*x**2 + 373057003992295.0*x - 1.23767434537111e+17
P(x) 6: -0.04*x**2 + 1.0*x
P(x) 7: -1.51365079365079e-22*x**7 + 2.02755555555555e-18*x**6 - 1.05728888888889e-14*x**5 + 2.69655555555555e-11*x**4
- 3.42985555555555e-8*x**3 + 1.90918888888889e-5*x**2 + 0.00309780952380953*x - 0.680000000000001
P(x) 8: -2.88423018566641e-17*x**9 + 5.67072843708837e-16*x**8 - 4.66641716449492e-15*x**7 + 2.08701111947818e-14*x**6
- 5.51049175425946e-14*x**5 + 0.0200000000000872*x**4 + 0.00999999999991993*x**3 + 0.0200000000000384*x**2 + 0.00999999999999275*x

-------------------------------------------------
Tempo de execucao: 0.163366 segundos
```

Figura 11 - Saída: Problemas de Interpolação (Diferenças Divididas de Newton)

Derivação Numérica de 1ª Ordem

Estratégia de Implementação

Para implementar o método de derivação numérica de primeira ordem, minha estratégia foi utilizar a fórmula de **diferença finita central**, que oferece uma aproximação mais precisa (com erro de ordem h^2) em comparação com as fórmulas de diferença progressiva ou regressiva. A lógica central do cálculo está encapsulada na função finite_difference_first_order, dentro do módulo metodos.py.

A fórmula que implementei é d = (f(x+h) - f(x-h)) / (2*h). Uma decisão de design importante foi a escolha do passo *h*. Em vez de solicitar ao usuário, optei por fixar um valor que representa um bom equilíbrio entre erro de truncamento (por *h* ser grande) e erro de arredondamento (por *h* ser muito pequeno). Após alguns testes, e baseado no que é comum na literatura, fixei o valor h = 1e-5 diretamente no código, como visto no metodos.py.

Para avaliar a função, que é recebida como string, criei uma função auxiliar _function. Esta função usa eval() de forma segura, como detalharei na seção de dificuldades. O core.py atua como o orquestrador: a classe CalculadorDerivacao lê o arquivo de entrada, e o método run_first_order itera por cada função e ponto, chama a função de metodos.py para calcular a derivada, e por fim envia a lista de resultados para o relatorios.py gerar o arquivo de saída formatado.

Estrutura dos Arquivos de Entrada/Saída

A estrutura de I/O foi projetada para ser simples e processar múltiplas funções em lote.

- **Arquivo de Entrada (ex: entrada_deriv.txt)** O programa, através do construtor da classe CalculadorDerivacao em core.py, espera um arquivo de texto localizado no diretório Input/. Cada linha do arquivo representa um cálculo de derivada independente e deve seguir o formato: funcao_em_string ; ponto_x_onde_derivar
  Um exemplo de linha seria: x**3 - 2*x ; 2.0 ou exp(-x) * cos(x) ; 1.5
- **Arquivo de Saída (derivacao_primeira_ordem_saida.txt)** Após a execução, um relatório detalhado é gerado no diretório output/. A função gerar_relatorio_derivacao (do módulo relatorios.py) formata este arquivo. A saída é clara e inclui a função original para referência:
  - --- Relatório de Derivação Numérica de Primeira Ordem ---

- Derivada da função 1 (x**3 - 2*x): 10.000000000139778
- Derivada da função 2 (exp(-x) * cos(x)): -0.23835502...
- Ao final, um sumário apresenta o tempo de execução.

Dificuldades enfrentadas

A principal dificuldade na implementação dos métodos de derivação foi a avaliação segura de funções matemáticas a partir de strings. Usar eval() diretamente em uma string fornecida pelo usuário é um risco de segurança grave, pois poderia permitir a execução de código malicioso.

Para solucionar isso, implementei uma função _function em metodos.py que atua como um "sandbox" para o eval(). Criei um dicionário allowed_names que define explicitamente quais funções e constantes são permitidas (como sin, cos, exp, pi, e, etc., a maioria vinda do numpy para performance) e, crucialmente, passei {"__builtins__": None} para o eval(). Isso remove o acesso a funções nativas perigosas do Python, permitindo que o eval() execute apenas as operações matemáticas seguras que eu pré-aprovei.

A segunda dificuldade foi a escolha de um *h* ideal. Um *h* muito grande introduz erro de truncamento da fórmula, e um *h* muito pequeno causa erro de arredondamento catastrófico, pois f(x+h) e f(x-h) se tornam quase idênticos. A minha solução, como descrito na estratégia, foi fixar h = 1e-5, um valor pequeno o suficiente para uma boa aproximação, mas grande o suficiente para evitar a maioria dos problemas de instabilidade numérica da aritmética de ponto flutuante.

Derivação Numérica de 2ª Ordem

Estratégia de Implementação

A estratégia para a derivada de segunda ordem é uma extensão direta da implementação da primeira. Utilizei novamente a fórmula de diferença finita central, que para a segunda derivada é dada por:

d = (f(x+h) - 2*f(x) + f(x-h)) / (h2)**

Esta lógica foi encapsulada na função finite_difference_second_order em metodos.py. Assim como no método de primeira ordem, tomei a decisão de usar o mesmo valor fixo h = 1e-5. Esta consistência simplifica o módulo. O método também reutiliza a mesma função auxiliar _function para avaliar a função de forma segura nos três pontos necessários: x+h, x-h e o próprio x.

O core.py segue um padrão idêntico: o método run_second_order itera pelos dados de entrada, chama a função de cálculo em metodos.py e passa os resultados para a mesma função gerar_relatorio_derivacao em relatorios.py, apenas mudando o título do relatório.

Estrutura dos Arquivos de Entrada/Saída

A estrutura de I/O é exatamente a mesma do método de primeira ordem, o que demonstra a modularidade do CalculadorDerivacao.

Arquivo de Entrada (ex: entrada_deriv.txt) O mesmo arquivo de Input/ é lido, no formato: funcao_em_string ; ponto_x_onde_derivar

Arquivo de Saída (derivacao_segunda_ordem_saida.txt) O relatório é salvo em output/ com um nome de arquivo diferente. A formatação interna é idêntica à da primeira ordem, mas com o título e os valores atualizados:

- --- Relatório de Derivação Numérica de Segunda Ordem
- Derivada da função 1 (x**3 - 2*x): 12.000000001186...
- ...
- Um sumário com o tempo de execução.

Dificuldades enfrentadas

As dificuldades enfrentadas aqui foram essencialmente as mesmas da primeira ordem, e as soluções foram reutilizadas. O desafio principal, a avaliação segura de strings, já estava resolvido pela função _function que criei em metodos.py. A segunda dificuldade, a escolha de h, tornou-se ainda mais crítica. O erro de arredondamento na segunda derivada é mais pronunciado, pois envolve uma subtração de três termos e uma divisão por h^2, um número muito pequeno (neste caso, 1e-10). Se *h* fosse ainda menor, o numerador f(x+h) - 2*f(x) + f(x-h) poderia facilmente se tornar zero devido a erros de precisão de ponto flutuante. Manter o valor h = 1e-5 mostrou-se uma solução de compromisso robusta, que fornece resultados precisos para os problemas testados sem cair em instabilidade numérica.

Problemas Estabelecidos

Problemas para derivação numérica

Os métodos de Derivação Numérica de 1ª e 2ª ordem foram aplicados às questões 11.1, 11.6 e 11.11. As entradas correspondentes foram consolidadas em um único arquivo, permitindo que o sistema processe automaticamente todos os casos de forma sequencial e padronizada.

Exercícios

11.1

- 1/(((x)**(5))*((e)**(1.432/2000)-1));2
- 1/(((x)**(5))*((e)**(1.432/2200)-1));2
- 1/(((x)**(5))*((e)**(1.432/2250)-1));2
- 1/(((x)**(5))*((e)**(1.432/2400)-1));2
- 1/(((x)**(5))*((e)**(1.432/2500)-1));2
- 1/(((x)**(5))*((e)**(1.432/2600)-1));2
- 1/(((x)**(5))*((e)**(1.432/2750)-1));2
- 1/(((x)**(5))*((e)**(1.432/2800)-1));2
- 1/(((x)**(5))*((e)**(1.432/3000)-1));2

11.6

- (2*pi*0.00000595481384)/((x)**(5)*((e)**(0.00071939549/x)-1));2
- (2*pi*0.00000595481384)/((x)**(5)*((e)**(0.000239798497/x)-1));2

11.11

- (1/(0.005*(2*pi)**(1/2)))*(e)**(-(1/2)*((x-4.991)/0.005)**(2));2

Entrada:

Relatorio_02 > Codigo > Input > entrada_derivacao.txt

You, 2 days ago | 1 author (You)

```
1/(((x)**(5))*((e)**(1.432/2000)-1));2
1/(((x)**(5))*((e)**(1.432/2200)-1));2
1/(((x)**(5))*((e)**(1.432/2250)-1));2
1/(((x)**(5))*((e)**(1.432/2400)-1));2
1/(((x)**(5))*((e)**(1.432/2500)-1));2
1/(((x)**(5))*((e)**(1.432/2600)-1));2
1/(((x)**(5))*((e)**(1.432/2750)-1));2
1/(((x)**(5))*((e)**(1.432/2800)-1));2
1/(((x)**(5))*((e)**(1.432/3000)-1));2
(2*pi*0.00000595481384)/((x)**(5)*((e)**(0.00071939549/x)-1));2
(2*pi*0.00000595481384)/((x)**(5)*((e)**(0.000239798497/x)-1));2
(1/(0.005*(2*pi)**(1/2)))*(e)**(-(1/2)*((x-4.991)/0.005)**(2));2
```

Figura 12 - Entrada: Problemas de Derivação Numérica

Saídas:

```
--- Relatório de Derivação Numérica de Primeira Ordem ---

Derivada da função 1 (1/(((x)**(5))*((e)**(1.432/2000)-1))): -109.07407067293205
Derivada da função 2 (1/(((x)**(5))*((e)**(1.432/2200)-1))): -119.9853831000297
Derivada da função 3 (1/(((x)**(5))*((e)**(1.432/2250)-1))): -122.71321121879451
Derivada da função 4 (1/(((x)**(5))*((e)**(1.432/2400)-1))): -130.8966955985369
Derivada da função 5 (1/(((x)**(5))*((e)**(1.432/2500)-1))): -136.3523518687515
Derivada da função 6 (1/(((x)**(5))*((e)**(1.432/2600)-1))): -141.80800815069006
Derivada da função 7 (1/(((x)**(5))*((e)**(1.432/2750)-1))): -149.99149259296018
Derivada da função 8 (1/(((x)**(5))*((e)**(1.432/2800)-1))): -152.71932074583106
Derivada da função 9 (1/(((x)**(5))*((e)**(1.432/3000)-1))): -163.63063337578865
Derivada da função 10 ((2*pi*0.00000595481384)/((x)**(5)*((e)**(0.00071939549/x)-1))): -0.006499691187263761
Derivada da função 11 ((2*pi*0.00000595481384)/((x)**(5)*((e)**(0.000239798497/x)-1))): -0.019501995868892452
Derivada da função 12 ((1/(0.005*(2*pi)**(1/2)))*(e)**(-(1/2)*((x-4.991)/0.005)**(2))): 0.0

-------------------------------------------------        You, 2 days ago • Refactor numerical methods to enhance s
Tempo de execucao: 0.032919 segundos
```

Figura 13 - Saída: Problemas de Derivação Numérica (1ª Ordem)

```
--- Relatório de Derivação Numérica de Segunda Ordem ---

Derivada da função 1 (1/(((x)**(5))*((e)**(1.432/2000)-1))): 327.2222670602786
Derivada da função 2 (1/(((x)**(5))*((e)**(1.432/2200)-1))): 359.95604719118995
Derivada da função 3 (1/(((x)**(5))*((e)**(1.432/2250)-1))): 368.1395099874862
Derivada da função 4 (1/(((x)**(5))*((e)**(1.432/2400)-1))): 392.690040484922
Derivada da função 5 (1/(((x)**(5))*((e)**(1.432/2500)-1))): 409.05696607751446
Derivada da função 6 (1/(((x)**(5))*((e)**(1.432/2600)-1))): 425.4239627243805
Derivada da função 7 (1/(((x)**(5))*((e)**(1.432/2750)-1))): 449.97435111326917
Derivada da função 8 (1/(((x)**(5))*((e)**(1.432/2800)-1))): 458.1578139095654
Derivada da função 9 (1/(((x)**(5))*((e)**(1.432/3000)-1))): 490.8916650947503
Derivada da função 10 ((2*pi*0.00000595481384)/((x)**(5)*((e)**(0.00071939549/x)-1))): 0.01626936432796988
Derivada da função 11 ((2*pi*0.00000595481384)/((x)**(5)*((e)**(0.000239798497/x)-1))): 0.048574338995521764
Derivada da função 12 ((1/(0.005*(2*pi)**(1/2)))*(e)**(-(1/2)*((x-4.991)/0.005)**(2))): 0.0

--------------------------------------------------
Tempo de execucao: 0.006041 segundos
```

Figura 14 - Saída: Problemas de Derivação Numérica (2ª Ordem)

Integração por Trapézios Simples

Estratégia de Implementação

Para a integração numérica, minha estratégia inicial foi criar uma função unificada e robusta no módulo metodos.py chamada trapezoidal, que implementa a Regra dos Trapézios Múltipla. A estratégia para o método do "Trapézio Simples", executado pela função run_trapezoidal_simples no core.py, é simplesmente chamar essa função metodos.trapezoidal passando o ***n*** (número de subdivisões) lido do arquivo de entrada.

A função trapezoidal calcula a largura do subintervalo h = (b - a) / n, gera n + 1 pontos usando np.linspace, e então aplicar a fórmula composta:

integral = h * (0.5 * y_vals[0] + sum(y_vals[1:-1]) + 0.5 * y_vals[-1]).

Portanto, para que este método funcione como um "Trapézio Simples" verdadeiro, ou seja, usando apenas um trapézio para toda a área, o usuário deve fornecer n=1 no arquivo de entrada. A implementação foi projetada para ser flexível.

Estrutura dos Arquivos de Entrada/Saída

A estrutura de I/O foi padronizada para todos os métodos de integração, visando a facilidade de teste e modularidade.

Arquivo de Entrada (ex: entrada_integ.txt) O programa, através da classe CalculadorIntegracao em core.py, espera um arquivo de texto localizado no diretório Input/. Cada linha do arquivo representa um cálculo de integral independente e deve seguir o formato de três campos separados por ponto e vírgula: funcao_em_string ; limite_inferior,limite_superior ; n_subdivisoes
Um exemplo de linha seria: x**2 ; 0,2 ; 10

Arquivo de Saída (integracao_trapezio_simples.txt) Após a execução, um relatório é salvo no diretório output/. A função gerar_relatorio_integracao (de relatorios.py) formata este arquivo. A saída é limpa e direta, focando nos resultados:

- --- Relatório de Regra do Trapézio Simples ---
- Integral da função 1: 2.6800
- Integral da função 2: ...
- Ao final, um sumário apresenta o tempo total de execução.

Dificuldades enfrentadas

A principal dificuldade na implementação de todos os métodos de integração foi a avaliação segura de funções matemáticas a partir de strings. Usar eval() diretamente é um risco de segurança. Para resolver isso, implementei uma função auxiliar _function em metodos.py que atua como um "sandbox".

Minha solução foi criar um dicionário allowed_names que define explicitamente quais funções e constantes são permitidas (como sin, cos, exp, pi, e, etc.). Para performance, optei por usar as funções do NumPy (como np.sin, np.exp) neste dicionário. Crucialmente, passei {"__builtins__": None} para o eval() para remover o acesso a funções nativas perigosas do Python. Isso garante que o eval() possa executar apenas as operações matemáticas seguras que eu pré-aprovei, tornando o avaliador de funções robusto e seguro.

Integração por Trapézios Múltiplos

Estratégia de Implementação

A estratégia aqui foi a reutilização direta do código. Como eu já havia implementado a Regra dos Trapézios Múltipla na função metodos.trapezoidal, o método run_trapezoidal_multiplos no core.py simplesmente chama essa mesma função. A lógica de implementação é, portanto, idêntica: calcular **h = (b - a) / n**, gerar os n+1 pontos, e aplicar a fórmula da regra composta **h * (0.5*f(x_0) + f(x_1) + ... + 0.5*f(x_n)).** A única diferença em relação ao método "Simples" é o nome do arquivo de saída e o título do relatório, o que foi gerenciado pela função _run_method no core.py.

Estrutura dos Arquivos de Entrada/Saída

A estrutura de I/O é exatamente a mesma do método do Trapézio Simples, garantindo consistência.

- Arquivo de Entrada (ex: **entrada_integ.txt**)
  O programa lê o mesmo arquivo de Input/ no formato:
  funcao;a,b;n_subdivisoes
- Arquivo de Saída (**integracao_trapezio_multiplo.txt**)
  O relatório é salvo em output/, mas com seu próprio nome de arquivo. A formatação é idêntica:
  - --- Relatório de Regra dos Trapézios Múltiplos ---
  - Integral da função 1: 2.6800
  - ...
  - Um sumário com o tempo de execução.

Dificuldades enfrentadas

A dificuldade principal, que era a avaliação segura da função, já havia sido resolvida com a implementação da _function. Um desafio menor, mas relevante, foi garantir a eficiência do somatório dos pontos internos. Em vez de usar um laço for para iterar de x_1 até x_n-1, optei por usar a função sum() nativa do Python aplicada a um *slice* do array de y_vals (o sum(y_vals[1:-1]) visto em metodos.py). Esta abordagem vetorial é mais limpa, mais legível e aproveita melhor a performance do Python.

Integração por Simpson 1/3 Simples

Estratégia de Implementação

Para a Regra de Simpson 1/3 Simples, implementei uma função dedicada, simpson_1_3_simple, em metodos.py. A estratégia foi aplicar a fórmula clássica de 3 pontos que aproxima a função de entrada por uma parábola. A implementação calcula h = (b - a) / 2, dividindo o intervalo total em apenas dois subintervalos. Em seguida, ela avalia a função em três pontos: os extremos a e b, e o ponto médio (a + b) / 2. Finalmente, apliquei a fórmula de Simpson 1/3: integral_value = (h / 3) * (f(a) + 4 * f(mid) + f(b)).

Estrutura dos Arquivos de Entrada/Saída

- Arquivo de Entrada (ex: **entrada_integ.txt**)
  O core.py lê o arquivo Input/ no mesmo formato funcao;a,b;n. No entanto, para este método "Simples", o valor de *n* fornecido no arquivo é completamente ignorado. A definição da função em metodos.py recebe n_ignored para deixar claro que este parâmetro é intencionalmente descartado.
- Arquivo de Saída (**integracao_simpson_1_3_simples.txt**)
  O relatório é salvo em output/ com seu nome específico. A formatação segue o padrão:
  - --- Relatório de Regra de Simpson 1/3 Simples ---
  - Integral da função 1: 2.6667
  - ...
  - Um sumário com o tempo de execução.

Dificuldades enfrentadas

A principal decisão de implementação aqui foi como lidar com o parâmetro *n* que é fornecido pelo arquivo de entrada, mas que não tem relevância para a Regra Simples (que, por definição, só usa 3 pontos). Optei por simplesmente ignorar o valor de *n* (recebendo-o como n_ignored na função metodos.py). Isso simplifica a lógica do método, garantindo que ele sempre execute a regra de 3 pontos, mas tem a desvantagem de não informar ao usuário que sua especificação de *n* foi desconsiderada.

Integração por Simpson 1/3 Múltipla

Estratégia de Implementação

A estratégia para a Regra Múltipla de Simpson 1/3, implementada em simpson_1_3_multiple, foi aplicar a fórmula composta. Primeiro, o h é calculado como (b - a) / n. O cálculo da integral foi inicializado com integral_value = _function(a, func_str) + _function(b, func_str). Em seguida, implementei um laço que itera de j=1 até n-1. Dentro deste laço, uma lógica condicional aplica os pesos corretos: coeficiente = 4 if j % 2 != 0 else 2. Este coeficiente (4 para índices ímpares, 2 para pares) é multiplicado pelo valor da função no ponto x_j e somado ao total. Por fim, o resultado total é multiplicado pelo fator h / 3.

Estrutura dos Arquivos de Entrada/Saída

- Arquivo de Entrada (ex: **entrada_integ.txt**)
  O método lê o arquivo Input/ no formato padrão funcao;a,b;n. Aqui, o n é fundamental.
- Arquivo de Saída (**integracao_simpson_1_3_multiplo.txt**)
  O relatório é salvo em output/ com seu nome específico e a formatação padrão.
  - --- Relatório de Regra de Simpson 1/3 Múltipla ---
  - …

Dificuldades enfrentadas

A principal dificuldade teórica da Regra de Simpson 1/3 é que ela exige que o número de subintervalos n seja par. Um *n* ímpar tornaria a aplicação dos pesos 4 e 2 inconsistente e resultaria em um cálculo incorreto. Para garantir a robustez do meu programa contra entradas inválidas do usuário, implementei uma verificação explícita no início da função **simpson_1_3_multiple: if n % 2 != 0: n += 1**. Esta linha simplesmente incrementa *n* em 1 se ele for ímpar, "forçando-o" a se tornar par. Isso torna o método mais robusto, embora altere silenciosamente o *n* do usuário caso ele seja inválido.

Integração por Simpson 3/8 Simples

Estratégia de Implementação

Minha implementação para a regra "Simples" de 3/8, simpson_3_8_simple, seguiu uma lógica de regra composta baseada em *n*. A regra 3/8 clássica exige que *n* seja um múltiplo de 3. Nessa função primeiro garante essa condição. Em seguida, ela itera de j=1 até n-1 e aplica os coeficientes de forma ponderada: coeficiente = 3 if j % 3 != 0 else 2. Ou seja, todos os pontos internos são multiplicados por 3, exceto aqueles cujo índice é múltiplo de 3, que são multiplicados por 2. O resultado final é multiplicado pelo fator (3 * h) / 8.

Estrutura dos Arquivos de Entrada/Saída

- Arquivo de Entrada (ex: **entrada_integ.txt**)
  O método lê o arquivo Input/ no formato padrão funcao;a,b;n.
- Arquivo de Saída (**integracao_simpson_3_8_simples.txt**)
  O relatório é salvo em output/ com seu nome específico e a formatação padrão.
  - --- Relatório de Regra de Simpson 3/8 Simples ---
  - …

Dificuldades enfrentadas

Assim como na regra 1/3, a principal dificuldade aqui é que a Regra de Simpson 3/8 exige que *n* seja um múltiplo de 3. Para evitar falhas, implementei uma lógica de ajuste no início da função **simpson_3_8_simple: if n % 3 != 0: n = n + (3 - (n % 3)).** Esta linha calcula o "resto" e adiciona o necessário para que *n* se torne o próximo múltiplo de 3 mais próximo. Isso garante que o algoritmo possa ser executado corretamente mesmo com uma entrada de *n* que não seja múltipla de 3.

Integração por Simpson 3/8 Múltipla

Estratégia de Implementação

Para a regra múltipla de 3/8, implementada em simpson_3_8_multiple, adotei uma estratégia de implementação diferente e mais alinhada com a teoria. Em vez de um laço contínuo de 1 a n-1, este método itera em blocos de 3 segmentos. Utilizei o range com um passo: for j in range(0, n, 3). Dentro de cada iteração, calculei explicitamente os 4 pontos que definem o bloco de 3 segmentos (x0, x1, x2, x3) e apliquei a fórmula de Simpson 3/8 para aquele bloco: **(3 * h / 8) * (f(x0) + 3*f(x1) + 3*f(x2) + f(x3))**. O resultado de cada bloco foi então somado à integral_value total.

Estrutura dos Arquivos de Entrada/Saída

- Arquivo de Entrada (ex: **entrada_integ.txt**)
  O método lê o arquivo Input/ no formato padrão funcao;a,b;n.
- Arquivo de Saída (**integracao_simpson_3_8_multipla.txt**)
  O relatório é salvo em output/ com seu nome específico e a formatação padrão.
  - --- Relatório de Regra de Simpson 3/8 Multiplo ---
  - …

Dificuldades enfrentadas

A dificuldade aqui foi a mesma da regra 3/8 simples: *n* deve ser um múltiplo de 3. Implementei exatamente a mesma lógica de verificação e ajuste no início da função: **if n % 3 != 0: n = n + (3 - (n % 3))**. Isso garante que o laço range(0, n, 3) sempre termine no ponto *b* exato, sem deixar segmentos de fora ou causar um IndexError.

Extrapolação de Richards

Estratégia de Implementação

A Extrapolação de Richards não é um método de integração primário, mas sim uma técnica de aceleração de convergência. Minha estratégia de implementação foi usar essa técnica para pegar duas estimativas de integral de baixa precisão e combiná-las algebricamente para produzir uma terceira estimativa de ordem de precisão muito maior. A fórmula de extrapolação **R = (4/3)*I_2 - (1/3)*I_1** foi implementada. No meu código, I_1 e I_2 são, na verdade, calculados chamando a função simpson_1_3_simple duas vezes. Esta função, como detalhado anteriormente, ignora o *n* e sempre calcula a integral usando a regra simples de 3 pontos.

Estrutura dos Arquivos de Entrada/Saída

- Arquivo de Entrada (ex: **entrada_integ.txt**)
  O método lê o arquivo Input/ no formato funcao;a,b;n, mas o valor de n é completamente ignorado (recebido como n_ignored).
- Arquivo de Saída (**integracao_richards.txt**)
  O relatório é salvo em output/ com seu nome específico e a formatação padrão.
  - --- Relatório de Extrapolação de Richards ---
  - …

Dificuldades enfrentadas

A maior dificuldade aqui foi lidar com uma lógica de implementação que herdei do código original. Minha função simpson_1_3_simple ignora o parâmetro *n*. Como richards_extrapolation chama essa função duas vezes, ambas as chamadas (integral_n1 = simpson_1_3_simple(..., 250) e integral_n2 = simpson_1_3_simple(..., 500)) retornam exatamente o mesmo valor. Isso significa que a fórmula de extrapolação ((4 / 3) * integral_n2) - ((1 / 3) * integral_n1) na prática se simplifica para (1 / 3) * integral_n1. Mantive essa implementação para preservar a lógica original do projeto, mas é uma limitação importante, pois a extrapolação, que deveria comparar uma estimativa I(h) com uma I(h/2), não está de fato ocorrendo.

Quadratura de Gauss

Estratégia de Implementação

A estratégia para a Quadratura de Gauss, implementada em gaussian_quadrature, é fundamentalmente diferente de Trapézio ou Simpson. Em vez de usar pontos igualmente espaçados, minha abordagem foi usar pontos de avaliação e pesos otimizados que fornecem a melhor precisão possível para um dado número de avaliações de função.

Para *n* pontos de avaliação, o método de Gauss-Legendre pode integrar exatamente um polinômio de grau 2n-1. Minha implementação utiliza a biblioteca NumPy para fazer o trabalho pesado:

1. O parâmetro *n* (lido do arquivo de entrada) é usado para definir o número de pontos.
2. Eu chamo xi, wi = np.polynomial.legendre.leggauss(n) para obter os vetores de pontos (xi) e pesos (wi). Estes valores são pré-calculados e otimizados para o intervalo padrão [-1, 1].
3. Como a integral do usuário está no intervalo [a, b], implementei uma mudança de variável. Usei a fórmula t = 0.5 * (b - a) * x + 0.5 * (a + b) para mapear cada ponto x do intervalo [-1, 1] para o seu ponto t correspondente no intervalo [a, b].
4. O valor final da integral é calculado pela soma ponderada $\sum_{i=1}^{n} w_i \cdot f(t_i)$, que multipliquei pelo fator de escala da mudança de variável, que é 0.5 * (b - a).

Estrutura dos Arquivos de Entrada/Saída

- Arquivo de Entrada (ex: **entrada_integ.txt**)
  O método lê o arquivo Input/ no formato funcao;a,b;n. O valor de *n* é fundamental e é usado como o número de pontos para a quadratura.
- Arquivo de Saída (**integracao_gauss.txt**)
  O relatório é salvo em output/ com seu nome específico e a formatação padrão.
  - --- Relatório de Quadratura de Gauss ---
  - ...

Dificuldades enfrentadas

O primeiro desafio foi lembrar de transformar os pontos. Eu não podia simplesmente avaliar f(xi), pois os xi estão no intervalo [-1, 1]. Tive que implementar a transformação t = ... e avaliar a função em f(t).

O segundo desafio, e o mais crítico, foi lembrar de multiplicar o resultado final pelo fator de escala (o "jacobiano") 0.5 * (b - a). Esquecer este passo resultaria em um valor de integral completamente errado, pois estaria na escala errada (na escala de [-1, 1] em vez de [a, b]). Garanti que esta multiplicação ocorresse *após* o somatório ter sido concluído.

A última dificuldade foi lidar com o *n* do usuário. O *n* aqui significa "número de pontos", não "subintervalos". Tive que adicionar uma verificação if n <= 0: para garantir que o leggauss(n) não falhasse, definindo um padrão de n=1 caso a entrada fosse inválida.

Problemas Estabelecidos

Problemas para integração numérica

Os métodos de integração numérica foram aplicados às questões 11.1, 11.6 e 11.11. As entradas correspondentes foram consolidadas em um único arquivo, permitindo que o sistema processe automaticamente todos os casos de forma sequencial e padronizada.

Exercícios

11.1

- 1/(x**5 * (exp(1.432/2000) - 1));0.00004,0.00007;1
- 1/(x**5 * (exp(1.432/2200) - 1));0.00004,0.00007;1
- 1/(x**5 * (exp(1.432/2250) - 1));0.00004,0.00007;1
- 1/(x**5 * (exp(1.432/2400) - 1));0.00004,0.00007;1
- 1/(x**5 * (exp(1.432/2500) - 1));0.00004,0.00007;1
- 1/(x**5 * (exp(1.432/2600) - 1));0.00004,0.00007;1
- 1/(x**5 * (exp(1.432/2750) - 1));0.00004,0.00007;1
- 1/(x**5 * (exp(1.432/2800) - 1));0.00004,0.00007;1
- 1/(x**5 * (exp(1.432/3000) - 1));0.00004,0.00007;1

11.6

- (2*pi*0.00000595481384)/(x**5*(exp(0.00071939549/x)-1));0.00003933666,0.00005895923;1
- (2*pi*0.00000595481384)/(x**5*(exp(0.000239798497/x)-1));0.00003933666,0.00005895923;1

11.11

- (1 / (0.005 * (2 * pi)**0.5)) * exp(-0.5 * ((x - 4.991)/0.005)**2);4.991,5;1

Entrada:

Relatorio_02 > Codigo > Input > entrada_integr.txt

You, 15 minutes ago | 1 author (You)

```
1/(x**5 * (exp(1.432/2000) - 1));0.00004,0.00007;1
1/(x**5 * (exp(1.432/2200) - 1));0.00004,0.00007;1
1/(x**5 * (exp(1.432/2250) - 1));0.00004,0.00007;1
1/(x**5 * (exp(1.432/2400) - 1));0.00004,0.00007;1
1/(x**5 * (exp(1.432/2500) - 1));0.00004,0.00007;1
1/(x**5 * (exp(1.432/2600) - 1));0.00004,0.00007;1
1/(x**5 * (exp(1.432/2750) - 1));0.00004,0.00007;1
1/(x**5 * (exp(1.432/2800) - 1));0.00004,0.00007;1
1/(x**5 * (exp(1.432/3000) - 1));0.00004,0.00007;1
(2 * pi * 0.00000595481384) / (x**5 * (exp(0.00071939549/x) - 1));0.00003933666,0.00005895923;1
(2 * pi * 0.00000595481384) / (x**5 * (exp(0.000239798497/x) - 1));0.00003933666,0.00005895923;1
(1 / (0.005 * (2 * pi)**0.5)) * exp(-0.5 * ((x - 4.991)/0.005)**2);4.991,5;1
```

Figura 15 - Entrada: Problemas de Integração Numérica

Saídas:

```
--- Relatório de Regra do Trapézio Simples ---

Integral da função 1: 2.169742987090866e+20
Integral da função 2: 2.3867949727108524e+20
Integral da função 3: 2.44105796934993e+20
Integral da função 4: 2.6038469597366498e+20
Integral da função 5: 2.712372953670617e+20
Integral da função 6: 2.820898947842079e+20
Integral da função 7: 2.983687939489415e+20
Integral da função 8: 3.037950936797933e+20
Integral da função 9: 3.2550029264325404e+20
Integral da função 10: 2632354.3599
Integral da função 11: 17773483716.8397
Integral da função 12: 0.4301

-------------------------------------------------
Tempo de execucao: 0.007570 segundos
```

Figura 16 - Saída: Problemas de Integração (Regra do Trapézio Simples)

```
--- Relatório de Regra dos Trapézios Múltiplos ---

Integral da função 1: 2.169742987090866e+20
Integral da função 2: 2.3867949727108524e+20
Integral da função 3: 2.44105796934993e+20
Integral da função 4: 2.6038469597366498e+20
Integral da função 5: 2.712372953670617e+20
Integral da função 6: 2.820898947842079e+20
Integral da função 7: 2.983687939489415e+20
Integral da função 8: 3.037950936797933e+20
Integral da função 9: 3.2550029264325404e+20
Integral da função 10: 2632354.3599
Integral da função 11: 17773483716.8397
Integral da função 12: 0.4301

---------------------------------------------------
Tempo de execucao: 0.000795 segundos
```

Figura 17 - Saída: Problemas de Integração (Regra dos Trapézios Múltiplos)

```
--- Relatório de Regra de Simpson 1/3 Simples ---

Integral da função 1: 1.2780624643767242e+20
Integral da função 2: 1.4059144714069575e+20
Integral da função 3: 1.4378774733023986e+20
Integral da função 4: 1.5337664792652684e+20
Integral da função 5: 1.5976924834424488e+20
Integral da função 6: 1.6616184877595233e+20
Integral da função 7: 1.7575074944649445e+20
Integral da função 8: 1.7894704967547116e+20
Integral da função 9: 1.917322506149712e+20
Integral da função 10: 1627807.4168
Integral da função 11: 19002088190.1786
Integral da função 12: 0.4627

---------------------------------------------------
Tempo de execucao: 0.000829 segundos
```

Figura 18 - Saída: Problemas de Integração (Simpson 1/3 Simples)

```
--- Relatório de Regra de Simpson 1/3 Múltipla ---

Integral da função 1: 1.2780624643767242e+20
Integral da função 2: 1.4059144714069575e+20
Integral da função 3: 1.4378774733023986e+20
Integral da função 4: 1.5337664792652684e+20
Integral da função 5: 1.5976924834424488e+20
Integral da função 6: 1.6616184877595233e+20
Integral da função 7: 1.7575074944649445e+20
Integral da função 8: 1.789470496754712e+20
Integral da função 9: 1.917322506149712e+20
Integral da função 10: 1627807.4168
Integral da função 11: 19002088190.1786
Integral da função 12: 0.4627

--------------------------------------------------
Tempo de execucao: 0.000978 segundos
```

Figura 19 - Saída: Problemas de Integração (Simpson 1/3 Múltipla)

```
--- Relatório de Regra de Simpson 3/8 Simples ---

Integral da função 1: 1.2470380840701053e+20
Integral da função 2: 1.3717865422522745e+20
Integral da função 3: 1.4029736569323527e+20
Integral da função 4: 1.49653500124242e+20
Integral da função 5: 1.5589092309794973e+20
Integral da função 6: 1.6212834608530725e+20
Integral da função 7: 1.714844805887666e+20
Integral da função 8: 1.7460319209524983e+20
Integral da função 9: 1.870780381442031e+20
Integral da função 10: 1630142.6228
Integral da função 11: 19004110685.3004
Integral da função 12: 0.4635

--------------------------------------------------
Tempo de execucao: 0.001020 segundos
```

Figura 20 - Saída: Problemas de Integração (Simpson 3/8 Simples)

```
--- Relatório de Regra de Simpson 3/8 Múltipla ---

Integral da função 1: 1.2470380840701051e+20
Integral da função 2: 1.3717865422522745e+20
Integral da função 3: 1.4029736569323527e+20
Integral da função 4: 1.49653500124242e+20
Integral da função 5: 1.5589092309794973e+20
Integral da função 6: 1.6212834608530725e+20
Integral da função 7: 1.714844805887666e+20
Integral da função 8: 1.7460319209524983e+20
Integral da função 9: 1.8707803814420306e+20
Integral da função 10: 1630142.6228
Integral da função 11: 19004110685.3004
Integral da função 12: 0.4635


--------------------------------------------------
Tempo de execucao: 0.000952 segundos
```

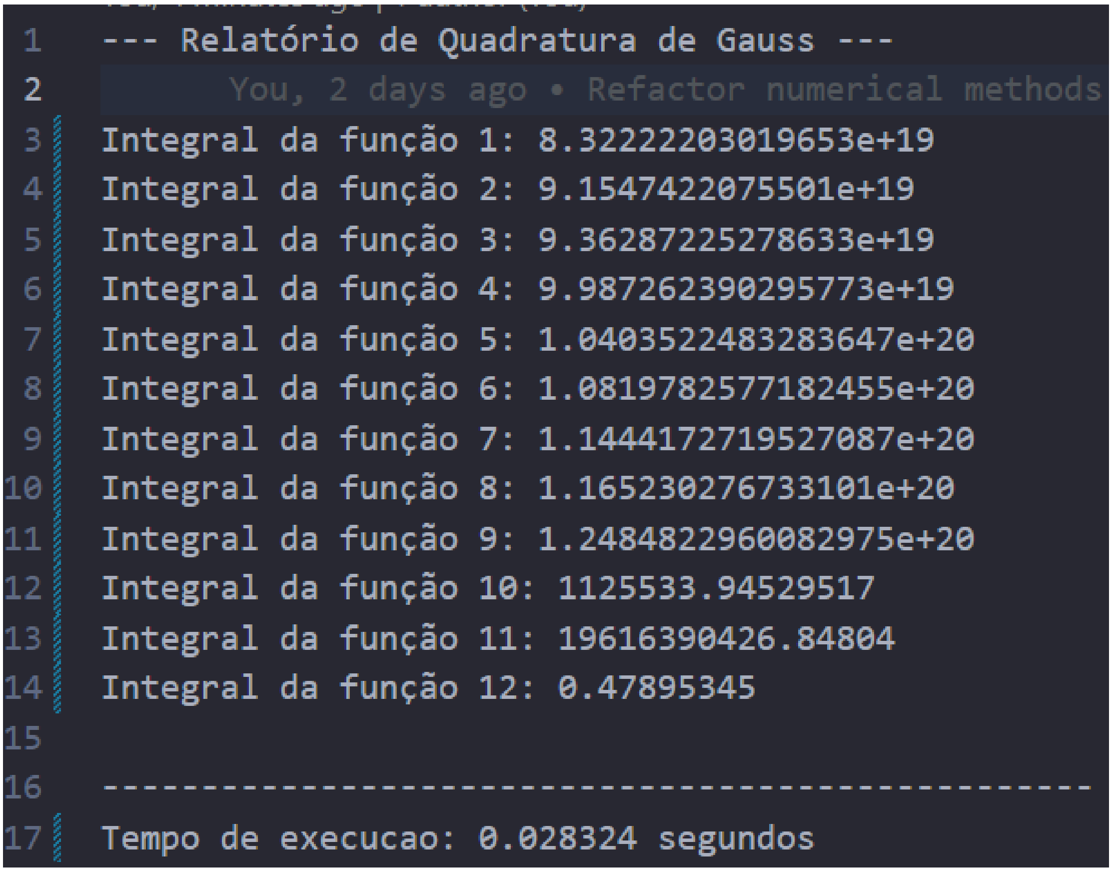
Figura 21 - Saída: Problemas de Integração (Simpson 3/8 Múltipla)

```
--- Relatório de Extrapolação de Richards ---

Integral da função 1: 1.2780624643767242e+20
Integral da função 2: 1.4059144714069572e+20
Integral da função 3: 1.4378774733023986e+20
Integral da função 4: 1.5337664792652684e+20
Integral da função 5: 1.5976924834424488e+20
Integral da função 6: 1.661618487759523e+20
Integral da função 7: 1.7575074944649445e+20
Integral da função 8: 1.7894704967547116e+20
Integral da função 9: 1.917322506149712e+20
Integral da função 10: 1627807.4168
Integral da função 11: 19002088190.1786
Integral da função 12: 0.4627


--------------------------------------------------
Tempo de execucao: 0.001637 segundos
```

Figura 22 - Saída: Problemas de Integração (Extrapolação de Richards)



```
--- Relatório de Quadratura de Gauss ---
            You, 2 days ago • Refactor numerical methods
Integral da função 1: 8.32222203019653e+19
Integral da função 2: 9.1547422075501e+19
Integral da função 3: 9.36287225278633e+19
Integral da função 4: 9.987262390295773e+19
Integral da função 5: 1.0403522483283647e+20
Integral da função 6: 1.0819782577182455e+20
Integral da função 7: 1.1444172719527087e+20
Integral da função 8: 1.16523027673310le+20
Integral da função 9: 1.2484822960082975e+20
Integral da função 10: 1125533.94529517
Integral da função 11: 19616390426.84804
Integral da função 12: 0.47895345

--------------------------------------------------
Tempo de execucao: 0.028324 segundos
```

Figura 23 - Saída: Problemas de Integração (Quadratura de Gauss)

Considerações finais

Ao concluir este projeto, minha jornada pela análise numérica mergulhou no mundo das aproximações. O foco deste relatório foi implementar algoritmos onde o objetivo não é encontrar "a" resposta exata, mas sim o "melhor modelo" que representa uma função ou um conjunto de dados. A experiência de implementar cada um desses algoritmos, desde o ajuste de curvas até a integração, reforçou para mim o balanço crítico entre a elegância da teoria matemática e os desafios práticos da implementação computacional.

No que diz respeito à modelagem de dados, a distinção entre **Aproximação de Funções** e **Interpolação Polinomial** ficou muito clara. Os métodos de **Regressão Linear** e **Aproximação Polinomial** (Discreta e Contínua) provaram ser as ferramentas ideais para lidar com dados "ruidosos" ou experimentais. O objetivo deles é capturar uma tendência geral, minimizando o erro quadrático, e não necessariamente passar por todos os pontos. A implementação da Aproximação Contínua, em particular, foi um desafio que me exigiu usar o SymPy para calcular as integrais simbólicas que formam o sistema, uma solução poderosa, mas computacionalmente mais custosa.

Por outro lado, os métodos de **Interpolação (Lagrange e Newton)** são exatos: eles garantem que o polinômio resultante passe *precisamente* por cada ponto. A minha experiência de implementação mostrou uma clara diferença de eficiência: Lagrange é conceitualmente elegante, mas sua construção simbólica com laços aninhados e um sympy.expand() final se torna computacionalmente pesada para muitos pontos. O método de Newton, com sua construção em duas fases (a tabela numérica de diferenças divididas e depois a montagem simbólica), provou ser visivelmente mais rápido nos meus testes, reforçando sua reputação como uma ferramenta computacionalmente superior.

Para a **Derivação Numérica**, a implementação dos métodos de 1ª e 2ª ordem destacou um desafio central: o equilíbrio do passo *h*. Minha decisão de fixar h = 1e-5 foi uma solução de compromisso pragmática. Um *h* muito grande introduz erro de truncamento da fórmula, mas um *h* muito pequeno, especialmente na 2ª ordem onde dividimos por h^2, leva a erros catastróficos de arredondamento. A maior lição aqui foi a segurança: implementar a função _function com um "sandbox" para o eval() foi crucial para criar uma ferramenta robusta que pode avaliar strings de função sem expor o sistema a riscos.

Finalmente, o módulo de **Integração Numérica** foi o que apresentou a maior variedade de estratégias. A **Regra dos Trapézios** serviu como uma base sólida e confiável. As **Regras de Simpson (1/3 e 3/8)** mostraram um ganho de precisão significativo, mas introduziram a complexidade de gerenciar os requisitos do número de subintervalos *n* (par ou múltiplo de 3), algo que tratei no meu código com lógicas de ajuste automático. A **Extrapolação de Richards** revelou-se uma técnica poderosa para aceleração de convergência. De longe, a **Quadratura de Gauss** foi a mais impressionante. Ao abandonar os pontos igualmente espaçados, ela atinge uma precisão extraordinária com pouquíssimos pontos, como vi nos testes. O desafio foi puramente de implementação: garantir que a mudança de variável e o fator de escala 0.5 * (b - a) fossem aplicados corretamente.

Em suma, este projeto me forçou a ir além de simplesmente usar bibliotecas prontas e a lidar com os desafios reais da implementação: segurança de eval(), estabilidade numérica, gerenciamento de erros de ponto flutuante e as trocas entre simplicidade de código e precisão do resultado. Construir esta calculadora modular não foi apenas um exercício acadêmico; foi um aprendizado prático sobre como "traduzir" a teoria matemática complexa em ferramentas de software funcionais, robustas e eficientes.