Notas de Aula em Física Computacional: Métodos Numéricos para a Solução de Equações Diferenciais

Francisco Anacleto Barros Fidelis de Moura

978-65-01-15717-7
ISBN

Notas de Aula em Física Computacional: Métodos Numéricos para a Solução de Equações Diferenciais

Francisco Anacleto Barros Fidelis de Moura

Prefácio

Este livro reúne notas de aula resumidas e organizadas sobre a solução numérica de equações diferenciais, com uma ênfase especial nas técnicas aplicadas à física computacional. Ele foi projetado para atender tanto estudantes de graduação quanto de pós-graduação, fornecendo uma base sólida e prática nas principais técnicas numéricas utilizadas no estudo de sistemas físicos. O objetivo central é garantir que o leitor adquira uma compreensão essencial dessas técnicas, destacando tanto o raciocínio teórico quanto a implementação prática, sempre com foco na aplicação a problemas físicos típicos enfrentados no ambiente acadêmico e profissional.

O conteúdo do livro está estruturado para oferecer, de forma clara e acessível, uma fundamentação teórica, seguida por orientações detalhadas sobre como aplicar esses métodos na prática. Cada técnica apresentada é acompanhada de exemplos práticos que demonstram sua utilidade na resolução de problemas reais. Isso permite que o leitor não apenas compreenda os conceitos abstratos envolvidos, mas também tenha a oportunidade de ver como esses conceitos podem ser usados diretamente para abordar questões concretas em diversas áreas da física.

Embora os códigos apresentados ao longo do livro estejam escritos na linguagem de programação C, é importante destacar que os métodos descritos são independentes de linguagem. Eles podem ser facilmente adaptados e implementados em outras linguagens amplamente utilizadas na comunidade científica e de engenharia, como Python, Fortran, MATLAB, entre outras. A escolha da linguagem de programação utilizada dependerá das preferências individuais do usuário, do contexto acadêmico ou das necessidades específicas de cada projeto. O uso de C neste material reflete a popularidade e a eficiência dessa linguagem em aplicações de alto desempenho, algo especialmente relevante em simulações de grande escala na física computacional.

As notas de aula foram elaboradas para focar nos aspectos mais importantes de cada técnica numérica. Isso garante que o leitor compreenda não apenas o "como", mas também o "porquê" por trás de cada método, promovendo uma compreensão profunda que é essencial para a aplicação eficaz dessas ferramentas em contextos diversos. A prática da implementação de algoritmos em códigos de computador é uma habilidade crucial para o desenvolvimento de novas soluções para problemas complexos, seja na pesquisa científica ou no mercado profissional.

O texto apresenta diversas aplicações práticas de técnicas numéricas, com muitos exemplos voltados para temas contemporâneos da física da matéria condensada, mecânica estatística, mecânica clássica, biofísica, entre outros. Mesmo que o aluno não tenha um conhecimento profundo em tópicos avançados, como a localização de Anderson, mecânica quântica ou potenciais não-lineares, ainda é possível aproveitar o material focando na solução numérica das equações diferenciais que surgem ao longo das discussões. A ênfase na abordagem computacional permite aplicar métodos numéricos para resolver problemas complexos, sem exigir o domínio completo dos conceitos físicos subjacentes.

Ao final, espero que este livro se torne uma fonte valiosa de aprendizado contínuo e uma referência prática indispensável para aqueles que se dedicam ao estudo e à aplicação de métodos numéricos na física. Seja você um estudante em busca de uma melhor compreensão ou um pesquisador procurando soluções computacionais eficientes para problemas complexos, este material foi desenvolvido para oferecer uma base sólida e uma perspectiva prática que o acompanhará em sua jornada através da física computacional.

Agradecimentos

Expresso minha mais profunda gratidão a todos que contribuíram para a realização deste trabalho. Em especial, agradeço à minha esposa Martha, ao meu filho Miguel e aos meus pais, Sr. Fidelis e Dona Daia, pelo apoio e incentivo constantes. Meus sinceros agradecimentos também aos meus alunos e alunas, cuja participação foi essencial na construção deste curso. Sou igualmente grato ao Instituto de Física da UFAL, bem como aos meus colegas, amigos e colaboradores, que sempre me inspiram a aprimorar a qualidade do nosso trabalho diário.

Índice

0.1 Introdução

As equações diferenciais ordinárias (EDOs) desempenham um papel central e indispensável na modelagem de uma ampla gama de fenômenos em diversas áreas do conhecimento, como física, química, biologia, economia e muitas outras. Essas equações são responsáveis por descrever como uma quantidade varia em função de outra variável, que, em muitos casos, é o tempo. Assim, elas são essenciais para a compreensão de sistemas dinâmicos, permitindo que se preveja o comportamento futuro de um sistema com base em informações sobre seu estado atual. Um exemplo clássico de aplicação das EDOs está no estudo do movimento de uma partícula sujeita a forças externas, onde as equações descrevem como a posição e a velocidade da partícula mudam ao longo do tempo. Outros exemplos incluem o crescimento de populações, onde as EDOs são utilizadas para modelar o aumento ou diminuição de indivíduos em uma população ao longo do tempo, e a propagação de doenças infecciosas, onde descrevem como o número de infectados evolui em uma determinada comunidade.

Além disso, processos físicos como o decaimento radioativo e a transferência de calor também são descritos por EDOs, sendo fundamentais para o entendimento desses fenômenos. Embora algumas equações diferenciais possam ser resolvidas de maneira analítica, ou seja, com uma solução exata expressa por meio de funções matemáticas conhecidas, muitos dos problemas práticos de interesse são mais complicados. Em geral, esses problemas envolvem equações diferenciais não lineares ou sistemas acoplados de equações, o que significa que suas variáveis estão inter-relacionadas de maneira complexa. Nessas situações, encontrar uma solução exata pode ser impossível ou extremamente difícil, o que faz com que métodos analíticos não sejam viáveis.

Em muitos cenários, as soluções numéricas tornam-se indispensáveis para abordar problemas complexos, onde as equações diferenciais ordinárias (EDOs) não podem ser resolvidas de forma analítica. Esse tipo de abordagem é crucial para obter soluções aproximadas que permitam entender e prever o comportamento de sistemas dinâmicos que seriam inacessíveis por métodos tradicionais. A importância das soluções numéricas é especialmente evidente em situações onde o sistema envolvido é altamente não linear ou composto por múltiplas equações interconectadas, como em sistemas acoplados ou com condições de contorno complicadas. Um exemplo típico é o estudo da meteorologia, onde a evolução do clima e de padrões atmosféricos depende de um grande número de variáveis interdependentes. As equações que regem esses fenômenos são muito complexas para serem resolvidas de maneira exata. Da mesma forma, na astrofísica, a simulação da dinâmica de corpos celestes, como estrelas, planetas e sistemas de galáxias, requer o uso de

soluções numéricas para lidar com as equações diferenciais que descrevem suas interações gravitacionais e suas trajetórias ao longo do tempo.

Outro exemplo de extrema importância é encontrado em simulações de engenharia, como o projeto de aeronaves, pontes e estruturas civis, onde as equações que descrevem as forças, tensões e deformações atuantes em materiais e componentes são altamente não lineares. Sem a possibilidade de calcular soluções analíticas, as soluções numéricas são o único caminho para prever o comportamento de uma estrutura sob diferentes condições de carga, garantindo a segurança e a eficácia de seu projeto.

No campo da biologia e da medicina, as equações diferenciais também modelam fenômenos complexos, como a propagação de doenças infecciosas em uma população ou o comportamento de redes neurais no cérebro. Em muitos casos, esses modelos envolvem múltiplos parâmetros que variam no tempo e no espaço, tornando inviável a obtenção de uma solução exata. As soluções numéricas, nesse contexto, fornecem previsões que podem ajudar na elaboração de políticas de saúde pública ou no desenvolvimento de tratamentos médicos.

Além disso, na física quântica e na mecânica estatística, a modelagem de sistemas com muitos corpos, como átomos em um material sólido ou partículas em uma simulação de fluido, requer o uso de soluções numéricas para resolver as equações diferenciais associadas à função de onda e à distribuição de partículas. Essas simulações permitem prever o comportamento de materiais em nível microscópico e são essenciais no desenvolvimento de novas tecnologias e materiais avançados.

Portanto, as soluções numéricas, longe de serem meros instrumentos auxiliares, são essenciais para a compreensão de fenômenos reais em diversos campos, oferecendo uma maneira poderosa de explorar sistemas complexos e fornecer previsões práticas onde métodos tradicionais falham.

0.1.1 Tipos de Métodos Numéricos

Diversos métodos numéricos foram desenvolvidos para resolver equações diferenciais ordinárias (EDOs), e cada um apresenta diferentes níveis de precisão, complexidade e eficiência computacional. Dependendo da natureza do problema, certos métodos são mais adequados, oferecendo soluções que equilibram entre simplicidade e precisão, ou que permitem a resolução de problemas complexos com maior eficiência. Entre os métodos mais amplamente utilizados, destacam-se:

Método de Euler: Este é o mais simples dos métodos numéricos e serve como uma introdução para resolver EDOs. Ele utiliza aproximações lineares, ou seja, a inclinação da curva no ponto atual é utilizada para pro-

jetar o próximo ponto. A simplicidade do método de Euler torna-o fácil de implementar e computacionalmente barato, mas sua precisão é limitada, especialmente em problemas onde o comportamento da solução é altamente não linear ou oscila de maneira significativa. Devido a essa baixa precisão, o método de Euler é mais usado para problemas simples ou como uma primeira aproximação para métodos mais sofisticados. Ele pode ser útil em modelos de crescimento populacional básico, onde os parâmetros são conhecidos e constantes, mas onde a precisão não é tão crítica.

Método de Runge-Kutta de 4ª ordem (RK4): Este é um dos métodos mais populares e amplamente utilizados. Ele proporciona um excelente equilíbrio entre precisão e eficiência computacional, sem ser excessivamente complexo. O RK4 calcula a solução em um ponto considerando quatro estimativas intermediárias, o que permite uma precisão muito maior do que o método de Euler. Este método é amplamente aplicado em problemas de mecânica clássica, simulações de sistemas dinâmicos, circuitos elétricos e em muitas áreas da engenharia. Sua eficiência o torna adequado para uma vasta gama de aplicações, desde a análise de sistemas de controle até simulações em física computacional.

Métodos de Runge-Kutta de alta ordem (6ª e 8ª ordem): Para problemas que exigem uma precisão ainda maior, como em simulações de longo prazo ou em sistemas altamente sensíveis a pequenas mudanças nas condições iniciais (exemplo típico em estudos de caos e turbulência), os métodos de Runge-Kutta de ordem superior são uma excelente escolha. Esses métodos aumentam a precisão ao custo de maior complexidade computacional, reduzindo drasticamente o erro acumulado ao longo de muitos passos de tempo. Eles são particularmente úteis em áreas como astronomia, onde simulações de órbitas planetárias precisam ser extremamente precisas, ou em física de partículas, onde erros acumulados podem alterar a interpretação dos resultados.

Método de Taylor é outro método numérico importante para a solução de EDOs, que se baseia na expansão em série de Taylor da função solução em torno de um ponto conhecido. Ao incluir os termos de ordem superior da série de Taylor, o método oferece uma precisão crescente à medida que mais termos são adicionados. A vantagem do método de Taylor está em sua capacidade de capturar o comportamento local da solução com alta precisão, o que o torna especialmente útil em problemas onde a função e suas derivadas de ordem superior são conhecidas ou podem ser calculadas facilmente. Esse método é aplicado em problemas onde a precisão é importante em uma região específica, como na simulação de trajetórias de satélites ou

na análise de sistemas controlados, onde pequenas variações nas condições iniciais podem levar a grandes diferenças nos resultados finais. Contudo, devido à necessidade de calcular derivadas de ordem superior, o método de Taylor pode ser computacionalmente intensivo, limitando seu uso a situações onde esses cálculos são viáveis.

Método de Verlet: Muito utilizado em simulações de dinâmica molecular e problemas de mecânica clássica, o método de Verlet é uma escolha natural quando a conservação de energia é fundamental para a precisão da simulação, como em sistemas de partículas. O método é especialmente eficaz em sistemas onde as forças derivadas de potenciais são conhecidas e onde o comportamento a longo prazo precisa ser analisado, como em simulações de materiais, interações moleculares ou na simulação de sistemas planetários. Devido à sua capacidade de preservar as propriedades energéticas do sistema, ele é amplamente empregado na simulação de gases, líquidos e sólidos.

Métodos Multistep: Diferentemente dos métodos de passo único, os métodos multistep, como os de Adams-Bashforth e Adams-Moulton, utilizam informações de múltiplos pontos anteriores da solução para prever o próximo valor. Isso permite que os métodos multistep atinjam uma maior precisão sem exigir passos de tempo muito pequenos, o que pode ser computacionalmente custoso. Esses métodos são particularmente úteis em problemas onde é necessário realizar simulações em longo prazo, como na previsão climática ou na dinâmica de sistemas financeiros, onde a eficiência computacional é crucial para lidar com a quantidade de dados e as exigências de precisão.

Métodos de Diferenças Finitas: Utilizados principalmente para resolver EDOs de ordem superior e equações diferenciais parciais (EDPs), os métodos de diferenças finitas aproximam as derivadas por meio de valores discretos da função em uma malha de tempo ou espaço. Eles são muito empregados em problemas de fronteira e em sistemas com geometrias complexas, como em análises de transferência de calor, fluxo de fluidos e problemas de elasticidade estrutural. Aplicações típicas incluem a modelagem de estruturas de edifícios, simulações de fluxo de calor em motores, ou na solução de equações que descrevem a difusão de poluentes no meio ambiente.

Métodos para Equações Estocásticas: Em muitos sistemas físicos, biológicos e econômicos, a incerteza ou a aleatoriedade desempenham um papel crucial. Para esses sistemas, as equações diferenciais estocásticas (EDEs) descrevem a evolução temporal com um componente de ruído, como o comportamento de mercados financeiros ou o movimento browniano de partículas. Métodos numéricos adaptados, como o método de Euler modifi-

cado para estocástica, são usados para lidar com essas flutuações aleatórias. Aplicações comuns incluem simulações financeiras (como a modelagem de preços de ativos) e em biologia, para simular sistemas onde o comportamento é influenciado por variações aleatórias no ambiente ou nas condições iniciais.

Cada um desses métodos tem suas vantagens e desvantagens, sendo mais adequado para tipos específicos de problemas. Métodos de passo único, como o de Euler ou RK4, são úteis em simulações de curto prazo ou problemas de baixa complexidade, enquanto métodos multistep ou de alta ordem são preferidos em simulações de longo prazo ou em sistemas sensíveis, onde a precisão é crítica. Os métodos de diferenças finitas são particularmente úteis para resolver problemas de fronteira e geometria complexa, enquanto os métodos estocásticos são essenciais para a modelagem de sistemas onde a incerteza é um fator inerente.

0.1.2 Estabilidade e Precisão

Dois aspectos cruciais no desenvolvimento de soluções numéricas para EDOs são a estabilidade e a precisão. A precisão de um método numérico refere-se à proximidade da solução calculada em relação à solução exata, enquanto a estabilidade está relacionada à capacidade do método de lidar com pequenas variações nos dados de entrada ou no próprio cálculo, sem amplificar erros ao longo do tempo.

Métodos como o de Euler são conhecidos por sua baixa estabilidade e precisão, especialmente para problemas "rígidos" (stiff), onde diferentes escalas de tempo coexistem no mesmo sistema. Nesses casos, métodos mais sofisticados, como os Runge-Kutta de alta ordem ou métodos implícitos, são preferíveis. Além disso, a escolha do tamanho do passo (Δt) é crítica. Passos menores aumentam a precisão, mas também aumentam o custo computacional, sendo essencial encontrar um equilíbrio entre eficiência e exatidão.

0.1.3 Aplicações de Soluções Numéricas de EDOs

As equações diferenciais ordinárias (EDOs) e suas soluções numéricas são ferramentas essenciais em várias áreas do conhecimento, sendo cruciais para modelar e entender a dinâmica de sistemas complexos. Essas equações descrevem como as variáveis de um sistema mudam ao longo do tempo ou em relação a outras variáveis, e seu uso é disseminado em contextos onde o comportamento evolutivo é fundamental. Abaixo estão algumas aplicações

detalhadas de EDOs em diferentes campos:

- **Física**: Em física, as EDOs são usadas para modelar sistemas dinâmicos em escala macroscópica e microscópica. Por exemplo, o movimento de planetas e satélites é descrito por EDOs que envolvem forças gravitacionais, onde as soluções numéricas são necessárias devido à complexidade das interações. Simulações de dinâmica molecular também se beneficiam dessas equações, permitindo prever o comportamento de átomos e moléculas em um sistema físico. Além disso, EDOs são fundamentais no estudo de sistemas oscilatórios, como o pêndulo simples ou sistemas mais complicados, como pêndulos acoplados e ressonâncias, onde soluções exatas nem sempre são possíveis.

- **Engenharia**: Na engenharia, as EDOs são amplamente utilizadas para modelar e projetar sistemas de controle, onde o comportamento dinâmico de sistemas mecânicos ou elétricos precisa ser regulado. A análise de circuitos elétricos, por exemplo, envolve EDOs para descrever a variação de corrente e tensão ao longo do tempo em resposta a sinais externos. Além disso, em processos de transferência de calor e massa, como em trocadores de calor ou reatores químicos, as EDOs permitem prever como o calor ou substâncias se difundem em sistemas com geometrias complexas. Aqui, métodos numéricos são indispensáveis para lidar com a natureza não linear desses fenômenos.

- **Biologia**: Na biologia, as EDOs são aplicadas para entender a dinâmica populacional e interações entre espécies em ecossistemas. Por exemplo, modelos de predador-presa, como o modelo de Lotka-Volterra, são resolvidos numericamente para prever oscilações populacionais ao longo do tempo. No campo da epidemiologia, as EDOs são fundamentais para modelar a propagação de doenças, permitindo prever o número de infectados ao longo do tempo com base em parâmetros como taxa de transmissão e recuperação. Além disso, redes ecológicas e sistemas biológicos mais complexos, que envolvem interações entre muitos organismos ou processos internos, frequentemente exigem métodos numéricos para análise.

- **Economia**: Em economia, as EDOs desempenham um papel importante na modelagem de sistemas dinâmicos, como a evolução dos preços de ativos, taxas de juros e inflação. Por exemplo, modelos de crescimento econômico ou de controle de políticas monetárias muitas vezes envolvem a solução de EDOs para entender a dinâmica de longo prazo

de variáveis macroeconômicas. A imprevisibilidade dos mercados financeiros também pode ser modelada por EDOs, onde fatores externos introduzem variabilidade e tornam a resolução numérica uma necessidade.

- **Química**: No campo da química, as EDOs são usadas para descrever o comportamento de reações químicas e processos cinéticos. Modelos que envolvem reações em série ou paralelas, onde as concentrações de reagentes e produtos mudam ao longo do tempo, são resolvidos numericamente, especialmente quando existem múltiplas etapas reacionais ou condições iniciais complexas. Em química quântica, por exemplo, a modelagem de dinâmicas moleculares e transições entre estados eletrônicos também depende fortemente de soluções numéricas.

O uso de soluções numéricas para EDOs se torna crucial em cenários onde a complexidade dos sistemas supera as capacidades de uma solução analítica. Essas técnicas fornecem meios de simular o comportamento temporal de sistemas dinâmicos, permitindo explorar profundamente fenômenos naturais e tecnológicos em diversas áreas. Com isso, os métodos numéricos oferecem uma compreensão detalhada e previsões precisas para sistemas que, de outra forma, seriam inacessíveis ao estudo tradicional.

0.1.4 Referências

Para aprofundamento no estudo das soluções numéricas de EDOs, as seguintes referências são recomendadas:

- Atkinson, K., Han, W., & Stewart, D. E. (2009). *Numerical Solution of Ordinary Differential Equations*. Wiley.
- Butcher, J. C. (2008). *Numerical Methods for Ordinary Differential Equations*. John Wiley & Sons.
- Press, W. H., Teukolsky, S. A., Vetterling, W. T., & Flannery, B. P. (2007). *Numerical Recipes: The Art of Scientific Computing*. Cambridge University Press.
- Hairer, E., Nørsett, S. P., & Wanner, G. (1993). *Solving Ordinary Differential Equations I: Nonstiff Problems*. Springer.

0.2 Método de Euler

O método de Euler é um dos métodos numéricos mais simples e amplamente utilizados para resolver equações diferenciais ordinárias (EDOs). Sua abordagem baseia-se na utilização da derivada em um ponto inicial para estimar o valor da função no próximo ponto, utilizando uma aproximação linear local. A simplicidade do método o torna uma excelente escolha introdutória, embora sua precisão dependa diretamente do tamanho do passo utilizado. Consideremos uma equação diferencial ordinária da forma:

$$y'(t) = f(t, y), \quad y(t_0) = y_0,$$

onde $y'(t)$ representa a derivada de $y(t)$ em relação a t, $f(t, y)$ é uma função que define a dinâmica da EDO, e y_0 é a condição inicial no tempo t_0. O método de Euler estima a solução em um intervalo de tempo subdividido em pequenos passos Δt, realizando a seguinte aproximação recursiva:

$$y_{n+1} = y_n + \Delta t f(t_n, y_n),$$

onde Δt é o tamanho do passo, y_n é a aproximação numérica de $y(t_n)$, e $t_n = t_0 + n\Delta t$. O método, portanto, calcula o valor de y em cada novo ponto t_{n+1}, utilizando o valor no ponto anterior t_n e a inclinação da curva dada por $f(t_n, y_n)$. Agora, consideremos um exemplo simples de uma EDO linear definida por:

$$y'(t) = -2y(t), \quad y(0) = 1,$$

que descreve uma taxa de decaimento exponencial de $y(t)$. Para aplicar o método de Euler, escolhemos um tamanho de passo $\Delta t = 0.1$. Nosso objetivo é encontrar uma solução aproximada para $y(t)$ nos valores de t entre 0 e 1. O procedimento consiste em calcular y_{n+1} a partir de y_n, conforme a fórmula dada, em cada passo de tempo sucessivo. Ao aplicar o método de Euler, devemos lembrar que sua precisão depende do tamanho de Δt: passos menores proporcionam uma maior precisão, porém à custa de mais iterações, enquanto passos maiores podem introduzir erro significativo, especialmente para equações com soluções rapidamente variáveis. A seguir, apresentamos a implementação em linguagem C para a solução aproximada desta EDO:

```c
#include <stdio.h>

double y, h, t;
int n, N;

int main() {
    y = 1.0;    // valor inicial de y(0)
    h = 0.1;    // tamanho do passo
    t = 0.0;    // tempo inicial
    N = 10;     // número de iterações (0 a 1 com passo 0.1)

    // Loop do método de Euler
    for (n = 0; n < N; n++) {
        printf("t = %lf, y = %lf\n", t, y);
        y = y + h * (-2 * y);  // y'(t) = -2y
        t = t + h;
    }

    return 0;
}
```

Neste código, utilizamos uma aproximação simples do método de Euler para resolver a EDO. Inicialmente, a variável y é definida com a condição inicial $y(0) = 1$, e o tempo t começa em zero. O programa avança com *n_steps* iterações, em cada uma das quais o valor de y é atualizado usando a equação de Euler $y_{n+1} = y_n + \Delta t \cdot (-2y_n)$, correspondente à EDO dada. Cada iteração exibe o valor de t e y, permitindo acompanhar a evolução da solução aproximada ao longo do tempo. Ao final do processo, temos uma estimativa numérica de $y(t)$ no intervalo especificado. Esse exemplo simples ilustra o poder do método de Euler para resolver EDOs de forma iterativa, aplicando um procedimento numérico direto, embora relativamente sensível ao tamanho do passo e à natureza da equação em questão.

0.3 Solução Numérica da Equação da Massa-Mola com Resistência do Ar: Método de Euler

Neste exemplo, vamos resolver numericamente a equação diferencial que descreve o movimento de um sistema massa-mola com resistência do ar. Este sistema é governado pela segunda lei de Newton e resulta em uma equação diferencial de segunda ordem, dada por:

$$m\frac{d^2x}{dt^2} = -kx - \beta v,$$

onde: - $x(t)$ é a posição da massa como função do tempo t, - $v = \frac{dx}{dt}$ é a

velocidade da massa, - k é a constante elástica da mola, - β é o coeficiente de resistência do ar (dissipação), - m é a massa do objeto.

A equação acima modela a dinâmica de um sistema oscilatório onde, além da força restauradora da mola, há uma força de resistência proporcional à velocidade, que representa a resistência do ar. Para resolver essa equação diferencial de segunda ordem, é útil reescrevê-la como um sistema de duas equações diferenciais de primeira ordem.

Podemos expressar o sistema como:

$$\frac{dx}{dt} = v$$

$$m\frac{dv}{dt} = -kx - \beta v,$$

onde a primeira equação define a relação entre a velocidade v e a posição x, e a segunda equação descreve a aceleração da massa, levando em conta tanto a força restauradora da mola quanto a força de resistência do ar.

Para resolver numericamente esse sistema de equações diferenciais, aplicamos o método de Euler, uma abordagem simples que utiliza uma aproximação linear para atualizar as variáveis x e v em pequenos incrementos de tempo Δt.As equações aproximadas no método de Euler para atualizar x e v a cada passo de tempo n são:

$$x_{n+1} = x_n + \Delta t \cdot v_n$$

$$v_{n+1} = v_n + \Delta t \cdot \left(\frac{-kx_n - \beta v_n}{m} \right)$$

Aqui: - x_n e v_n são as aproximações da posição e velocidade no tempo t_n, - x_{n+1} e v_{n+1} são as aproximações no tempo $t_{n+1} = t_n + \Delta t$, - Δt é o tamanho do passo de tempo utilizado para avançar a solução.

A cada passo Δt, a posição x é atualizada com base na velocidade v, e a velocidade v é atualizada usando a força total que age sobre a massa, que é a soma da força restauradora da mola $-kx$ e a força de resistência $-\beta v$.

Para implementar essa solução, usamos o método de Euler em um programa que itera por pequenos intervalos de tempo, atualizando x e v em cada passo. O resultado será a trajetória $x(t)$ e a velocidade $v(t)$ ao longo do tempo. A seguir, um código em C poderia ser utilizado para implementar o método de Euler. O código calcula a posição e velocidade da massa a cada

passo de tempo e salva os resultados em um arquivo para análise posterior. O arquivo de saída, chamado `resultados.dat`, contém os valores de t, $x(t)$ e $v(t)$ para visualização e interpretação da evolução do sistema.

Este método oferece uma maneira simples de simular sistemas dinâmicos reais, como um oscilador com resistência, mesmo quando uma solução analítica exata pode ser difícil ou impossível de obter. O uso de métodos numéricos, como o de Euler, permite resolver problemas físicos complexos com base em uma discretização direta da equação diferencial.

```
#include <stdio.h>

#define N 1000   // Número de passos de tempo
#define DELTA_T 0.01  // Passo de tempo

// Parâmetros do sistema
const double m = 1.0;   // Massa
const double k = 1.0;   // Constante elástica da mola
const double beta = 0.2; // Coeficiente de resistência do ar

// Função para resolver o sistema usando o método de Euler
void metodo_de_euler(double x0, double v0, double t_total) {
    double x = x0;  // Posição inicial
    double v = v0;  // Velocidade inicial
    double t = 0.0; // Tempo inicial

    FILE *file = fopen("resultados.dat", "w");
    fprintf(file, "# Tempo\tPosicao\tVelocidade\n");

    for (int i = 0; i < N; i++) {
        fprintf(file, "%lf\t%lf\t%lf\n", t, x, v);

        // Atualiza as variáveis usando o método de Euler
        double a = (-k * x - beta * v) / m;
        x = x + DELTA_T * v;
        v = v + DELTA_T * a;
        t = t + DELTA_T;
    }

    fclose(file);
}

int main() {
    double x_inicial = 1.0; // Posição inicial (deslocamento inicial)
    double v_inicial = 0.0; // Velocidade inicial (em repouso)

    metodo_de_euler(x_inicial, v_inicial, N * DELTA_T);

    return 0;
}
```

No código acima, o sistema massa-mola com resistência do ar é resolvido numericamente utilizando o método de Euler. As variáveis são inicializadas com os valores de posição e velocidade iniciais. O loop `for` itera através

dos N passos de tempo, atualizando a posição e a velocidade de acordo com as equações discretizadas do método de Euler. O arquivo de saída `resultados.dat` contém três colunas: o tempo, a posição e a velocidade a cada passo de tempo. Isso permite visualizar como o sistema evolui ao longo do tempo. O gráfico da posição e da velocidade pode ser facilmente plotado usando qualquer software de plotagem, como o `gnuplot`. O método de Euler, embora simples, é uma maneira eficaz de resolver sistemas de equações diferenciais ordinárias. No entanto, é importante observar que, para problemas onde uma maior precisão é necessária, outros métodos de integração numérica, como o método de Runge-Kutta de quarta ordem, podem ser mais adequados.

0.4 Método de Euler Melhorado

O método de Euler melhorado, também conhecido como método de Heun ou método de Euler de segunda ordem, oferece uma aproximação mais precisa ao utilizar uma correção baseada em uma média das inclinações no início e no final do intervalo de cada passo de tempo. A fórmula do método de Euler melhorado é dada por:

$$y_{n+1} = y_n + \Delta t \cdot \frac{f(t_n, y_n) + f(t_{n+1}, y_n + \Delta t f(t_n, y_n))}{2},$$

onde: - y_n é a aproximação de $y(t_n)$, - Δt é o tamanho do passo de tempo, - $f(t_n, y_n)$ representa a derivada da função no ponto (t_n, y_n), - $t_{n+1} = t_n + \Delta t$.

Diferente do método de Euler simples, que utiliza apenas a inclinação no ponto inicial do intervalo para determinar a próxima aproximação, o método de Euler melhorado calcula uma previsão inicial da inclinação no final do intervalo, ajustando a solução com base na média dessas inclinações. Isso proporciona uma maior precisão, especialmente quando o comportamento da função varia significativamente ao longo do intervalo. Vamos aplicar esse método à mesma equação diferencial ordinária (EDO) utilizada anteriormente:

$$y'(t) = -2y(t), \quad y(0) = 1,$$

onde a solução descreve uma exponencial decrescente. Usando o método de Euler melhorado, com um passo de tempo $\Delta t = 0.1$, podemos aproximar

a solução de $y(t)$ em valores de t entre 0 e 1. Neste caso, o método de Euler simples forneceria uma aproximação razoável, mas com a versão melhorada, obtemos uma solução numérica mais precisa ao reduzir o erro global. A abordagem corrigida se adapta melhor ao comportamento real da solução, especialmente em situações onde a taxa de variação da função não é linear. Abaixo está o esboço de um código que implementa este procedimento, utilizando a fórmula de Euler melhorado para obter a solução numérica da EDO. Esse código é uma ferramenta eficiente para resolver equações diferenciais onde a precisão numérica é crucial, e o método melhora significativamente os resultados sem aumentar drasticamente a complexidade computacional.

```
#include <stdio.h>

double y, h, t, k1, k2;
int n, N;

int main() {
    y = 1.0;    // valor inicial de y(0)
    h = 0.1;    // tamanho do passo
    t = 0.0;    // tempo inicial
    N = 10;     // número de iterações

    // Loop do método de Euler Melhorado
    for (n = 0; n < N; n++) {
        printf("t = %lf, y = %lf\n", t, y);
        k1 = -2 * y;
        k2 = -2 * (y + h * k1);
        y = y + (h / 2.0) * (k1 + k2);
        t = t + h;
    }

    return 0;
}
```

O código fornece uma implementação básica do método de Euler melhorado para a solução de uma EDO com uma função derivada simples. Ele imprime os valores da solução e do tempo em cada passo, permitindo a visualização da evolução da solução ao longo do tempo.

0.4.1 Lançamento Oblíquo com Efeito Magnus no Plano x-y

Neste exemplo, vamos estudar o movimento de uma bola no plano x-y, levando em consideração o efeito Magnus devido à rotação da bola. A força de Magnus gera uma força perpendicular ao vetor de velocidade e ao eixo de rotação da bola, resultando em uma trajetória curva. Para resolver

as equações de movimento, usaremos o método de Euler melhorado. As equações que regem o movimento da bola no plano x-y são:

$$m\frac{d^2x}{dt^2} = -\frac{1}{2}C_D\rho A v\frac{dx}{dt} + F_{\text{Magnus},x}$$

$$m\frac{d^2y}{dt^2} = -mg - \frac{1}{2}C_D\rho A v\frac{dy}{dt} + F_{\text{Magnus},y}$$

onde:

- m é a massa da bola,
- g é a aceleração da gravidade,
- C_D é o coeficiente de arrasto,
- ρ é a densidade do ar,
- A é a área da seção transversal da bola,
- v é a velocidade da bola.

A força de Magnus é uma força que surge em um corpo esférico em rotação, devido à interação entre o fluido (ar) ao redor e o movimento de rotação da bola. Essa força é perpendicular tanto à direção do movimento do corpo quanto ao eixo de rotação e pode ser expressa pela fórmula geral:

$$\vec{F}_{\text{Magnus}} = C_L \cdot \rho \cdot A \cdot v \cdot \vec{\omega} \times \vec{v}$$

Onde:

- C_L é o coeficiente de lift (elevação);
- ρ é a densidade do fluido (ar);
- A é a área de seção transversal da bola;
- v é a magnitude da velocidade translacional da bola;
- $\vec{\omega}$ é o vetor de rotação da bola;
- $\vec{v}$ é o vetor velocidade da bola.

Para deduzir formalmente as componentes da força de Magnus nas direções x e y, precisamos realizar o produto vetorial entre a velocidade angular $\vec{\omega}$ e a velocidade translacional $\vec{v}$. Para simplificar o problema, consideramos que a rotação da bola ocorre no plano x-y, ou seja, $\vec{\omega}$ aponta na direção z, perpendicular ao plano de movimento. Assim, temos:

$$\vec{\omega} = (0, 0, \omega)$$

$$\vec{v} = (v_x, v_y, 0)$$

O produto vetorial $\vec{\omega} \times \vec{v}$ é dado por:

$$\vec{\omega} \times \vec{v} = \begin{vmatrix} \hat{i} & \hat{j} & \hat{k} \\ 0 & 0 & \omega \\ v_x & v_y & 0 \end{vmatrix} = \hat{i}\,(0 \cdot 0 - \omega \cdot v_y) - \hat{j}\,(0 \cdot 0 - \omega \cdot v_x) + \hat{k}\,(0 \cdot v_y - 0 \cdot v_x)$$

Isso resulta em:

$$\vec{\omega} \times \vec{v} = (-\omega v_y, \omega v_x, 0)$$

Portanto, as componentes da força de Magnus nas direções x e y serão proporcionais a essas componentes do produto vetorial. Substituindo na equação da força de Magnus, obtemos:

$$F_{\text{Magnus},x} = C_L \cdot \rho \cdot A \cdot v \cdot (-\omega v_y)$$

$$F_{\text{Magnus},y} = C_L \cdot \rho \cdot A \cdot v \cdot (\omega v_x)$$

Aqui, $v = \sqrt{v_x^2 + v_y^2}$ é a magnitude da velocidade translacional da bola.

A força de Magnus na direção x é proporcional a $-v_y$, ou seja, depende da componente da velocidade na direção y. Já a força na direção y é proporcional a v_x, a componente da velocidade na direção x. Isso significa que a força de Magnus atua perpendicularmente à direção da velocidade, desviando a trajetória da bola, o que é característico desse tipo de força. A magnitude da força de Magnus também é proporcional à velocidade angular ω, o que implica que quanto maior a rotação da bola, mais significativa será a influência da força de Magnus sobre a sua trajetória. Sem rotação ($\omega = 0$), a força de Magnus desaparece e a trajetória da bola segue o movimento parabólico clássico, afetado apenas pela gravidade e resistência do ar. Vamos agora implementar a solução numérica usando o método de Euler melhorado. O código em C a seguir resolve as equações do movimento com o efeito Magnus.

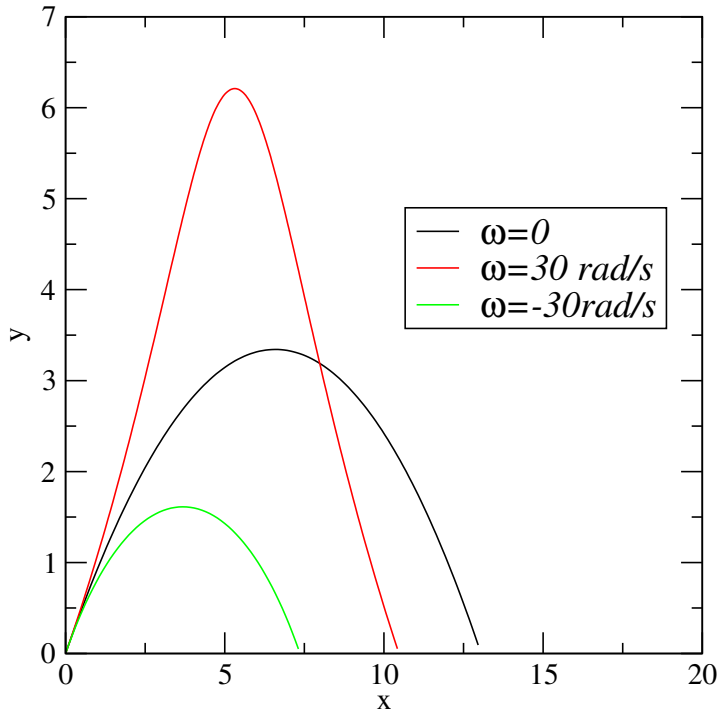

Figure 1: Trajetórias da bola no plano x-y considerando diferentes valores da velocidade angular ω. A linha preta representa o caso sem rotação ($\omega = 0$). A linha vermelha mostra a trajetória com $\omega = 30 rad/s$, e a linha verde com $\omega = -30 rad/s$. As curvas ilustram como a rotação da bola influencia a trajetória devido ao efeito Magnus, desviando-a lateralmente. A direção do desvio depende do sinal e da magnitude da rotação.

```
#include <stdio.h>
#include <math.h>

// Definições de constantes
#define NMAX 10000
#define G 9.81
#define CD 0.47
#define CL 0.2
#define RHO 1.225
#define A 0.01
#define M 1.0
#define OMEGA 0.1  // Velocidade angular da bola (rad/s)
#define DT 0.01  // Passo de tempo
#define TMAX 5.0  // Tempo máximo de simulação

// Variáveis globais
double x, y, vx, vy, ax, ay;
double t;
double v, Fm_x, Fm_y;

int N = 200;  // Será ajustado no main()

// Função que calcula as acelerações com o efeito Magnus
void calcula_aceleracao() {
```

```
    // Velocidade da bola
    v = sqrt(vx * vx + vy * vy);

    // Força de Magnus
    Fm_x = CL * RHO * v * A * OMEGA * (-vy);
    Fm_y = CL * RHO * v * A * OMEGA * (vx);

    // Aceleração no eixo x e y
    ax = (-0.5 * CD * RHO * A * v * vx + Fm_x) / M;
    ay = (-M * G - 0.5 * CD * RHO * A * v * vy + Fm_y) / M;
}

// Método de Euler melhorado
void euler_melhorado() {
    double k1x, k1y, k2x, k2y;
    double k1vx, k1vy, k2vx, k2vy;

    // Primeira parte do passo de Euler (previsão)
    k1x = vx * DT;
    k1y = vy * DT;

    k1vx = ax * DT;
    k1vy = ay * DT;

    // Atualiza as velocidades intermediárias
    vx += k1vx / 2.0;
    vy += k1vy / 2.0;

    // Calcula as novas acelerações com as velocidades intermediárias
    calcula_aceleracao();

    // Segunda parte do passo de Euler (correção)
    k2x = vx * DT;
    k2y = vy * DT;

    k2vx = ax * DT;
    k2vy = ay * DT;

    // Atualiza posição e velocidade
    x += k2x;
    y += k2y;

    vx += k2vx;
    vy += k2vy;
}

int main() {
    // Definir condições iniciais
    x = 0.0;
    y = 0.0;
    vx = 10.0;  // Velocidade inicial em x
    vy = 10.0;  // Velocidade inicial em y
    t = 0.0;

    // Abrir arquivo para gravar os resultados
    FILE *output = fopen("lancamento_magnus.dat", "w");

    // Loop de simulação
    while (t <= TMAX && y >= 0) {
```

```
        // Escrever os valores de tempo e posições
        fprintf(output, "%lf\t%lf\t%lf\n", x, y,t);

        // Calcular as novas acelerações
        calcula_aceleracao();

        // Atualizar as posições e velocidades com Euler melhorado
        euler_melhorado();

        // Atualizar o tempo
        t += DT;
    }

    // Fechar o arquivo
    fclose(output);

    return 0;
}
```

No código acima, as variáveis `x`, `y`, `vx`, `vy`, `ax`, e `ay` são declaradas globalmente antes do `main`, conforme a estrutura que preferimos. O método de Euler melhorado é implementado na função `euler_melhorado`, que prevê e corrige as posições e velocidades da bola, enquanto a função `calcula_aceleracao` calcula as acelerações com o efeito Magnus. Os resultados da simulação são gravados no arquivo `lancamento_magnus.dat`, contendo as posições da bola ao longo do tempo. A simulação mostra que a trajetória da bola é fortemente afetada pelo efeito Magnus, com uma curvatura evidente no movimento. O arquivo gerado pode ser usado para visualizar a trajetória no plano x-y e verificar a contribuição do efeito Magnus para o desvio lateral da bola. Na figura 1 mostro a trajetórias da bola no plano x-y considerando diferentes valores da velocidade angular ω. A linha preta representa o caso sem rotação ($\omega = 0$). A linha vermelha mostra a trajetória com $\omega = 30 rad/s$, e a linha verde com $\omega = -30 rad/s$. As curvas ilustram como a rotação da bola influencia a trajetória devido ao efeito Magnus, desviando-a lateralmente. A direção do desvio depende do sinal e da magnitude da rotação. Os demais dados da simulação desta figura estão indicados dentro do programa.

0.5 Método de Taylor de 2ª, 3ª e 4ª ordem

Vamos resolver a equação diferencial simples

$$\frac{dx}{dt} = -x$$

usando o método de Taylor. Esta equação descreve um processo de decaimento exponencial onde a taxa de variação da função $x(t)$ é proporcional ao valor negativo da própria função. Um exemplo de aplicação desse tipo de

equação pode ser o decaimento radioativo ou o resfriamento de um objeto. Vamos considerar , por exemplo, a condição inicial como $x(0) = 0.1$. Utilizaremos o método de Taylor para calcular a solução numérica para $x(t)$ em vários pontos no tempo. A expansão de Taylor de uma função $x(t)$ em torno de $t = 0$ é dada por:

$$x(t) = x(0) + \frac{dx(0)}{dt}t + \frac{1}{2!}\frac{d^2x(0)}{dt^2}t^2 + \frac{1}{3!}\frac{d^3x(0)}{dt^3}t^3 + \cdots$$

Sabemos que:

$$\frac{dx}{dt} = -x$$

Portanto, podemos calcular as derivadas de $x(t)$ sucessivamente:

1. Primeira derivada: $\frac{dx}{dt} = -x$
2. Segunda derivada: $\frac{d^2x}{dt^2} = -\frac{dx}{dt} = x$
3. Terceira derivada: $\frac{d^3x}{dt^3} = \frac{d^2x}{dt^2} = -x$
4. E assim por diante...

Desta feita, a expansão de Taylor para a função $x(\Delta t)$ será:

$$x(\Delta t) \approx x(0) - x(0)\Delta t + \frac{x(0)(\Delta t)^2}{2!} - \frac{x(0)(\Delta t)^3}{3!} + \cdots$$

Agora podemos implementar essa solução em um programa de C para calcular $x(t)$ numericamente para diferentes valores de t. Para implementar o método de Taylor em C, escolhemos um valor pequeno para o passo de tempo Δt e somamos sucessivamente os termos da série de Taylor. A seguir está o código em C para resolver a equação $\frac{dx}{dt} = -x$ usando o método de Taylor.

```
#include <stdio.h>
#include <math.h>

#define T_MAX 10    // Tempo máximo de simulação
```

```
#define N_TERMS 10  // Número de termos da série de Taylor
#define DELTA_T 0.1 // Passo de tempo

// Função para calcular o fatorial de um número
double factorial(int n) {
    if (n == 0) return 1;
    double fact = 1;
    for (int i = 1; i <= n; i++) {
        fact *= i;
    }
    return fact;
}

// Função para calcular a série de Taylor
double taylor_series(double x0, double t) {
    double result = x0;
    double term = x0;

    for (int k = 1; k < N_TERMS; k++) {
        // Alterna os sinais (-1)^k e calcula a potência do tempo e o fatorial
        term *= (-t / k);
        result += term;
    }

    return result;
}

int main() {
    double x0 = 0.1; // Condição inicial x(0) = 0.1
    double t = 0.0;

    printf("t\t\tx(t)\n");
    while (t <= T_MAX) {
        // Calcula x(t) usando a série de Taylor
        double x_t = taylor_series(x0, t);
        printf("%.2f\t\t%.10f\n", t, x_t);

        t += DELTA_T; // Avança no tempo
    }

    return 0;
}
```

Neste exemplo, utilizamos o método de Taylor para resolver numericamente a equação diferencial $\frac{dx}{dt} = -x$. O código acima implementa a série de Taylor e calcula a solução para diferentes valores de t, permitindo uma aproximação numérica da solução exata. Vamos considerar outro exemplo como sendo:

$$y'(t) = t + y, \quad y(0) = 1.$$

Utilizaremos o método de Taylor de 2ª ordem com $\Delta t = 0.1$ para resolver esta equação. A implementação comentada se encontra a seguir:

```
#include <stdio.h>

double y, h, t;
int n, N;

int main() {
    y = 1.0;    // valor inicial de y(0)
    h = 0.1;    // tamanho do passo
    t = 0.0;    // tempo inicial
    N = 10;     // número de iterações

    // Loop do método de Taylor 2ª ordem
    for (n = 0; n < N; n++) {
        printf("t = %lf, y = %lf\n", t, y);
        y = y + h * (t + y) + (h * h / 2.0) * (1 + t + y);
        t = t + h;
    }

    return 0;
}
```

O programa implementa o método de Taylor de segunda ordem para a solução numérica de uma equação diferencial ordinária. Inicialmente, define-se o valor da solução `y` como 1.0, o tamanho do passo `h` como 0.1 e o tempo inicial `t` como 0.0. O loop realiza 10 iterações, atualizando `y` e `t` a cada passo. A fórmula usada inclui termos de Taylor para uma melhor aproximação da solução. Os valores de `t` e `y` são impressos a cada iteração para visualização dos resultados.

0.6 Métodos de Runge-Kutta

O método de Runge-Kutta é um método numérico amplamente utilizado para resolver equações diferenciais ordinárias da forma:

$$\frac{dy}{dt} = f(t, y), \quad y(t_0) = y_0.$$

A ideia principal por trás dos métodos de Runge-Kutta é aproximar a solução $y(t)$ através de uma combinação ponderada de diferentes estimativas da inclinação da função. A precisão do método depende do número de avaliações da função $f(t, y)$, que definem a ordem do método. Aqui, deduziremos os métodos de Runge-Kutta de $2^{\underline{a}}$, $3^{\underline{a}}$ e $4^{\underline{a}}$ ordens.

Método de Runge-Kutta de $2^{\underline{a}}$ Ordem

Para deduzir o método de Runge-Kutta de $2^{\underline{a}}$ ordem, começamos com a expansão de Taylor da solução exata $y(t)$ em torno do ponto t_n:

$$y(t_{n+1}) = y(t_n) + \Delta t \frac{dy}{dt} + \frac{\Delta t^2}{2} \frac{d^2y}{dt^2} + O(\Delta t^3),$$

onde $\Delta t = t_{n+1} - t_n$ é o passo de tempo. O objetivo é encontrar uma aproximação y_{n+1} para $y(t_{n+1})$ que seja precisa até termos de $O(\Delta t^2)$. Para isso, assumimos que a inclinação média pode ser escrita como uma combinação ponderada de duas inclinações: uma no início do intervalo, $f(t_n, y_n)$, e outra no meio do intervalo, $f(t_n + \Delta t/2, y_n + k_1/2)$. Definimos:

$$k_1 = \Delta t f(t_n, y_n),$$

$$k_2 = hf\left(t_n + \frac{\Delta t}{2}, y_n + \frac{k_1}{2}\right).$$

Então, a aproximação de y_{n+1} é dada por:

$$y_{n+1} = y_n + k_2.$$

A escolha dos coeficientes de k_1 e k_2 garante que o método seja preciso até termos de $O(\Delta t^2)$, ou seja, que ele seja de $2^{\underline{a}}$ ordem.

Método de Runge-Kutta de 3ª Ordem

Para aumentar a precisão, podemos usar três avaliações da função $f(t, y)$, resultando no método de Runge-Kutta de 3ª ordem. Neste caso, introduzimos três inclinações k_1, k_2 e k_3:

$$k_1 = \Delta t f(t_n, y_n),$$

$$k_2 = hf\left(t_n + \frac{\Delta t}{2}, y_n + \frac{k_1}{2}\right),$$

$$k_3 = hf\left(t_n + \Delta t, y_n - k_1 + 2k_2\right).$$

A aproximação de y_{n+1} é dada por:

$$y_{n+1} = y_n + \frac{1}{6}(k_1 + 4k_2 + k_3).$$

Os coeficientes 1/6, 4/6 e 1/6 são escolhidos de forma a garantir que o método seja preciso até termos de $O(\Delta t^3)$, ou seja, que seja de 3ª ordem.

Método de Runge-Kutta de 4ª Ordem

O método de Runge-Kutta de 4ª ordem é o mais popular e envolve quatro avaliações da função $f(t, y)$. Definimos quatro inclinações k_1, k_2, k_3 e k_4:

$$k_1 = \Delta t f(t_n, y_n),$$

$$k_2 = hf\left(t_n + \frac{\Delta t}{2}, y_n + \frac{k_1}{2}\right),$$

$$k_3 = hf\left(t_n + \frac{\Delta t}{2}, y_n + \frac{k_2}{2}\right),$$

$$k_4 = \Delta t f(t_n + \Delta t, y_n + k_3).$$

A aproximação de y_{n+1} é dada por:

$$y_{n+1} = y_n + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4).$$

Aqui, os coeficientes 1/6, 2/6, 2/6 e 1/6 são determinados para garantir que o erro global seja $O(\Delta t^4)$, tornando o método de 4ª ordem. Os métodos de Runge-Kutta são obtidos ajustando combinações ponderadas de diferentes estimativas da inclinação da função, de modo a garantir a precisão desejada para uma dada ordem. O método de 2ª ordem usa duas avaliações da função, o de 3ª ordem usa três, e o de 4ª ordem, quatro, com coeficientes adequados para minimizar o erro local e global da solução aproximada. O erro nos métodos de Runge-Kutta pode ser analisado em termos de dois tipos de erro: o erro local e o erro global. O *erro local* refere-se ao erro cometido em cada passo do método, que é da ordem de $O(\Delta t^{p+1})$, onde p é a ordem do método. Já o *erro global* é a acumulação desses erros ao longo de todos os passos e, para um método de ordem p, o erro global é da ordem de $O(\Delta t^p)$. No caso do método de Runge-Kutta de 2ª ordem, o erro local é $O(\Delta t^3)$ e o erro global é $O(\Delta t^2)$. Para o método de 4ª ordem, o erro local é $O(\Delta t^5)$ e o erro global é $O(\Delta t^4)$, o que faz dele um método bastante preciso para uma escolha adequada de passo h. Embora os métodos de ordem mais alta apresentem menores erros globais, é importante balancear a precisão com o custo computacional, já que métodos de ordem superior requerem mais avaliações da função $f(t, y)$.

Método de Runge-Kutta de 5ª Ordem com 6 Estágios

O método de Runge-Kutta de 5ª ordem é um método explícito de integração numérica que utiliza 6 estágios para aproximar a solução de uma equação diferencial ordinária (EDO) da forma

$$\frac{dy}{dt} = f(t, y), \quad y(t_0) = y_0.$$

Para calcular a solução aproximada y_{n+1} a partir do ponto (t_n, y_n), o método envolve a seguinte fórmula:

$$y_{n+1} = y_n + b_1k_1 + b_2k_2 + b_3k_3 + b_4k_4 + b_5k_5 + b_6k_6.$$

onde valores de k_i são calculados como:

$$\begin{aligned}
k_1 &= \Delta t f(t_n, y_n), \\
k_2 &= \Delta t f\left(t_n + c_2\Delta t, y_n + a_{21}k_1\right), \\
k_3 &= \Delta t f\left(t_n + c_3\Delta t, y_n + a_{31}k_1 + a_{32}k_2\right), \\
k_4 &= \Delta t f\left(t_n + c_4\Delta t, y_n + a_{41}k_1 + a_{42}k_2 + a_{43}k_3\right), \\
k_5 &= \Delta t f\left(t_n + c_5\Delta t, y_n + a_{51}k_1 + a_{52}k_2 + a_{53}k_3 + a_{54}k_4\right), \\
k_6 &= \Delta t f\left(t_n + c_6\Delta t, y_n + a_{61}k_1 + a_{62}k_2 + a_{63}k_3 + a_{64}k_4 + a_{65}k_5\right).
\end{aligned}$$

Para o método de Runge-Kutta de 5ª ordem, os coeficientes específicos podem ser os coeficientes do método Dormand-Prince (RK5(4)):

i) coeficiente c_i:

$$\begin{aligned}
c_2 &= \frac{1}{5}, \\
c_3 &= \frac{3}{10}, \\
c_4 &= \frac{4}{5}, \\
c_5 &= \frac{8}{9}, \\
c_6 &= 1,
\end{aligned}$$

ii) coeficiente $a_{i,j}$:

$i\backslash j$	1	2	3	4	5	6
2	$\frac{1}{5}$					
3	$\frac{3}{40}$	$\frac{9}{40}$				
4	$\frac{44}{45}$	$-\frac{56}{15}$	$\frac{32}{9}$			
5	$\frac{19372}{6561}$	$-\frac{25360}{2187}$	$\frac{64448}{6561}$	$-\frac{212}{729}$		
6	$\frac{4390}{2187}$	$-\frac{25360}{2187}$	$\frac{64448}{6561}$	$-\frac{212}{729}$	$\frac{11}{84}$	

iii) coeficiente b_i:

$$
\begin{aligned}
b_1 &= \frac{35}{384}, \\
b_2 &= \frac{500}{1113}, \\
b_3 &= \frac{125}{192}, \\
b_4 &= -\frac{2187}{6784}, \\
b_5 &= \frac{11}{84}, \\
b_6 &= 0.
\end{aligned}
$$

O método de Runge-Kutta de 5ª ordem possui um erro local de $O(\Delta t^6)$, e o erro global de $O(\Delta t^5)$. Isso significa que a precisão do método melhora rapidamente com o tamanho do passo Δt, o que é uma grande vantagem quando se precisa de alta precisão. No entanto, alguns aspectos importantes devem ser considerados:

- **Escolha do Tamanho do Passo**: Apesar de a ordem alta garantir precisão, o método ainda depende da escolha adequada do passo Δt. Passos muito grandes podem levar a erros significativos, enquanto passos muito pequenos podem resultar em um aumento no custo computacional sem benefício substancial de precisão.

- **Estabilidade Numérica**: Métodos de ordem mais alta podem ser sensíveis a erros numéricos acumulados, e é crucial verificar a estabilidade do método para a classe de problemas em questão. O método

de Runge-Kutta de 5ª ordem geralmente é mais estável, mas ainda é importante monitorar a estabilidade, especialmente em problemas rígidos.

- **Complexidade Computacional**: A precisão adicional dos métodos de alta ordem vem com um custo computacional adicional, pois mais avaliações da função são necessárias. Em problemas de larga escala ou em sistemas com muitos equacionamentos diferenciais, a escolha do método deve equilibrar precisão e eficiência.

Método de Runge-Kutta de 8ª Ordem

Outro método de Runge-Kutta interessante é a versão de 8ª ordem. É um método explícito para resolver equações diferenciais ordinárias com alta precisão. Ele utiliza 9 estágios para obter uma aproximação precisa da solução. O método de Runge-Kutta de 8ª ordem usa os seguintes coeficientes para calcular os estágios k_i:

$$
\begin{aligned}
k_1 &= \Delta t \cdot f(t_n, y_n), \\
k_2 &= \Delta t \cdot f\left(t_n + c_2\Delta t, y_n + a_{21}k_1\right), \\
k_3 &= \Delta t \cdot f\left(t_n + c_3\Delta t, y_n + a_{31}k_1 + a_{32}k_2\right), \\
k_4 &= \Delta t \cdot f\left(t_n + c_4\Delta t, y_n + a_{41}k_1 + a_{42}k_2 + a_{43}k_3\right), \\
k_5 &= \Delta t \cdot f\left(t_n + c_5\Delta t, y_n + a_{51}k_1 + a_{52}k_2 + a_{53}k_3 + a_{54}k_4\right), \\
k_6 &= \Delta t \cdot f\left(t_n + c_6\Delta t, y_n + a_{61}k_1 + a_{62}k_2 + a_{63}k_3 + a_{64}k_4 + a_{65}k_5\right), \\
k_7 &= \Delta t \cdot f\left(t_n + c_7\Delta t, y_n + a_{71}k_1 + a_{72}k_2 + a_{73}k_3 + a_{74}k_4 + a_{75}k_5 + a_{76}k_6\right), \\
k_8 &= \Delta t \cdot f\left(t_n + c_8\Delta t, y_n + a_{81}k_1 + a_{82}k_2 + a_{83}k_3 + a_{84}k_4 + a_{85}k_5 + a_{86}k_6 + a_{87}k_7\right), \\
k_9 &= \Delta t \cdot f\left(t_n + c_9\Delta t, y_n + a_{91}k_1 + a_{92}k_2 + a_{93}k_3 + a_{94}k_4 + a_{95}k_5 + a_{96}k_6 + a_{97}k_7 + a_{98}k
\end{aligned}
$$

Os coeficientes c_i são apresentados na tabela abaixo:

i	c_i
2	$\frac{1}{18}$
3	$\frac{1}{12}$
4	$\frac{1}{8}$
5	$\frac{5}{24}$
6	$\frac{1}{6}$
7	$\frac{7}{24}$
8	$\frac{1}{2}$
9	1

Os coeficientes a_{ij} são organizados na tabela a seguir:

$i\backslash j$	1	2	3	4	5	6	7	8	9
2	$\frac{1}{18}$	0	0	0	0	0	0	0	0
3	$\frac{1}{12}$	$\frac{1}{12}$	0	0	0	0	0	0	0
4	$\frac{1}{8}$	$\frac{1}{8}$	0	$\frac{1}{8}$	0	0	0	0	0
5	$\frac{5}{24}$	$\frac{5}{24}$	0	$\frac{5}{24}$	$\frac{5}{24}$	0	0	0	0
6	$\frac{1}{6}$	$\frac{1}{6}$	$\frac{1}{6}$	$\frac{1}{6}$	$\frac{1}{6}$	$\frac{1}{6}$	0	0	0
7	$\frac{7}{24}$	$\frac{7}{24}$	$\frac{7}{24}$	$\frac{7}{24}$	$\frac{7}{24}$	$\frac{7}{24}$	$\frac{7}{24}$	0	0
8	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	0
9	1	1	1	1	1	1	1	1	1

Os coeficientes b_i são dados pela tabela a seguir:

i	b_i
1	$\frac{16}{135}$
2	0
3	$\frac{6656}{12825}$
4	$\frac{28561}{56430}$
5	0
6	0
7	0
8	0
9	$\frac{1}{3}$

O método de Runge-Kutta de 8ª ordem possui um erro local de $O(\Delta t^9)$, e o erro global de $O(\Delta t^8)$. Isso garante uma alta precisão em comparação com métodos de ordem inferior. No entanto, a complexidade do método e o custo computacional aumentam com a ordem, tornando-o mais exigente em termos de recursos computacionais. A escolha do tamanho do passo Δt deve ser cuidadosamente feita para balancear precisão e eficiência, considerando que métodos de ordem mais alta podem acumular erros numéricos e de truncamento em problemas rigorosos.

Os métodos de Runge-Kutta de alta ordem, como o método de 8ª ordem, são projetados para alcançar alta precisão na solução de equações diferenciais ordinárias. No entanto, a ordem elevada do método não garante automaticamente a sua estabilidade. A estabilidade é uma propriedade crucial que deve ser considerada separadamente da ordem do método. Métodos de alta ordem podem ser instáveis em alguns casos, especialmente quando aplicados a problemas com características particulares, como rigidez. A estabilidade global de um método é a sua capacidade de manter uma solução numericamente estável ao longo do tempo, mesmo quando o passo de integração é grande. Em resumo, embora métodos de alta ordem como o Runge-Kutta de 8ª ordem ofereçam alta precisão, a sua estabilidade não é garantida. A escolha do método deve considerar tanto a precisão quanto a estabilidade, especialmente em problemas com características desafiadoras como rigidez. Vamos iniciar a aplicação de diversos métodos de RK em alguns exemplos computacionais. Vamos aplicar o método de Runge-Kutta de 2ª ordem à seguinte equação diferencial:

$$y'(t) = y - t^2 + 1, \quad y(0) = 0.5.$$

O objetivo é encontrar a solução para $y(t)$ no intervalo $t \in [0, 2]$ com um passo de $\Delta t = 0.2$.

Passo 1: No ponto inicial, temos $t_0 = 0$ e $y_0 = 0.5$. Calculamos k_1 como:

$$k_1 = f(t_0, y_0) = 0.5 - 0^2 + 1 = 1.5.$$

Passo 2: Agora, calculamos k_2 no ponto médio $t_0 + \Delta t/2 = 0.1$:

$$k_2 = f\left(0.1, 0.5 + 0.1 \cdot 1.5\right) = f(0.1, 0.65) = 0.65 - 0.1^2 + 1 = 1.64.$$

Passo 3: Finalmente, calculamos y_1 usando a fórmula de Runge-Kutta de $2^{\underline{a}}$ ordem:

$$y_1 = 0.5 + 0.2 \cdot \frac{1.5 + 1.64}{2} = 0.5 + 0.2 \cdot 1.57 = 0.814.$$

Resultado: Repetindo o processo para os próximos intervalos, obtemos uma aproximação para $y(t)$ ao longo do intervalo de integração. A implementação deste procedimento em C pode ser encontrada a seguir:

```
#include <stdio.h>

double y, h, t, k1, k2;
int n, N;

int main() {
    y = 0.5;    // valor inicial de y(0)
    h = 0.2;    // tamanho do passo
    t = 0.0;    // tempo inicial
    N = 10;     // número de iterações

    // Loop do método de Runge-Kutta 2ª ordem
    for (n = 0; n < N; n++) {
        printf("t = %lf, y = %lf\n", t, y);

        // Cálculo de k1 e k2
        k1 = y - t * t + 1;
        k2 = (y + h * k1 / 2) - (t + h / 2) * (t + h / 2) + 1;

        // Atualização de y e t
        y = y + h * (k1 + k2) / 2;
        t = t + h;
    }

    return 0;
}
```

O programa utiliza o método de Runge-Kutta de segunda ordem para resolver uma equação diferencial ordinária. Começa com um valor inicial `y` de 0.5, um tamanho de passo `h` de 0.2, e tempo inicial `t` igual a 0.0. Em cada iteração, calcula os coeficientes `k1` e `k2` com base na fórmula diferencial e usa esses valores para atualizar `y` e `t`. O loop executa 10 iterações, imprimindo os valores de `t` e `y` a cada passo para análise dos resultados. Vamos aplicar

este método de 2^a ordem em um exemplo interessante dentro do contexto da localização de Anderson.

0.7 Modelagem do Modelo de Anderson para Dois Sítios com Runge-Kutta de 2ª Ordem

O modelo de Anderson é um modelo fundamental na física de sistemas desordenados, utilizado para estudar a localização de estados eletrônicos em um potencial desordenado. Nesta seção, focaremos em um modelo simplificado com dois sítios e suas respectivas energias. Vamos explorar o comportamento do sistema usando métodos numéricos, especificamente o método de Runge-Kutta de $2^{\underline{a}}$ ordem, para resolver as equações diferenciais associadas. Consideramos um sistema com dois sítios, cada um com uma energia específica ϵ_1 e ϵ_2. O Hamiltoniano do sistema pode ser descrito pela seguinte matriz Hamiltoniana H:

$$H = \begin{pmatrix} \epsilon_1 & -J \\ -J & \epsilon_2 \end{pmatrix}$$

onde $J = 1$ é o parâmetro de acoplamento entre os sítios. O problema pode ser formulado como um sistema de equações diferenciais para as funções de onda $\psi_1(t)$ e $\psi_2(t)$, associadas a cada sítio. As equações diferenciais são obtidas a partir da equação de Schrödinger dependente do tempo:

$$i\hbar \frac{d}{dt} \begin{pmatrix} \psi_1(t) \\ \psi_2(t) \end{pmatrix} = H \begin{pmatrix} \psi_1(t) \\ \psi_2(t) \end{pmatrix}$$

Com $\hbar = 1$ (unidade de energia e tempo), isso se simplifica para:

$$i \frac{d}{dt} \begin{pmatrix} \psi_1(t) \\ \psi_2(t) \end{pmatrix} = \begin{pmatrix} \epsilon_1 & -1 \\ -1 & \epsilon_2 \end{pmatrix} \begin{pmatrix} \psi_1(t) \\ \psi_2(t) \end{pmatrix}$$

Separando as equações, temos:

$$i\frac{d\psi_1(t)}{dt} = \epsilon_1\psi_1(t) - \psi_2(t)$$

$$i\frac{d\psi_2(t)}{dt} = \epsilon_2\psi_2(t) - \psi_1(t)$$

Para resolver numericamente essas equações diferenciais, utilizaremos o método de Runge-Kutta de 2ª ordem, também conhecido como método do ponto médio. O método é descrito pelos seguintes passos:

Seja $y(t) = \begin{pmatrix}\psi_1(t)\\ \psi_2(t)\end{pmatrix}$ e $f(t, y)$ a função que descreve as derivadas:

$$f(t, y) = \begin{pmatrix}-i(\epsilon_1\psi_1(t) - \psi_2(t))\\ -i(\epsilon_2\psi_2(t) - \psi_1(t))\end{pmatrix}$$

1. **Inicialização:**

$$y_n = \begin{pmatrix}\psi_1(t_n)\\ \psi_2(t_n)\end{pmatrix}$$

$$t_{n+1} = t_n + \Delta t$$

2. **Calcular os Incrementos:**

$$k_1 = \Delta t f(t_n, y_n)$$

$$k_2 = \Delta t f\left(t_n + \frac{\Delta t}{2}, y_n + \frac{k_1}{2}\right)$$

3. **Atualizar a Solução:**

$$y_{n+1} = y_n + k_2$$

Abaixo está o código em C que utiliza o método de Runge-Kutta de 2ª ordem para resolver as equações diferenciais associadas ao modelo de Anderson para dois sítios.

```
#include <stdio.h>
#include <stdlib.h>
#include <complex.h>
#include <math.h>

#define NMAX 10000

// Variáveis globais
int N; // Número de passos
double h; // Passo de integração
double epsilon1 = 1.0; // Energia do sítio 1
double epsilon2 = 1.0; // Energia do sítio 2
double t = 1.0; // Parâmetro de acoplamento

// Função que define as derivadas
void derivadas(double t, double complex *y, double complex *dy) {
    dy[0] = -I * (epsilon1 * y[0] - t * y[1]);
    dy[1] = -I * (epsilon2 * y[1] - t * y[0]);
}

int main() {
    double complex *y = (double complex *)malloc(2 * sizeof(double complex));
    double complex *dy = (double complex *)malloc(2 * sizeof(double complex));
    double complex *k1 = (double complex *)malloc(2 * sizeof(double complex));
    double complex *k2 = (double complex *)malloc(2 * sizeof(double complex));
    double t0 = 0.0; // Tempo inicial
    double t_final = 10.0; // Tempo final
    h = 0.01; // Passo de integração
    N = (int)((t_final - t0) / h);

    // Condições iniciais
    y[0] = 1.0 + 0.0*I; // psi_1 inicial
    y[1] = 0.0 + 0.0*I; // psi_2 inicial

    FILE *file = fopen("resultado.dat", "w");
    fprintf(file, "t, psi_1, psi_2\n");

  for (int n = 0; n < N; n++) {
   fprintf(file, "%.2f %.5f %.5f %.5f\n", t0, pow(cabs(y[0]),2.),pow(cabs(y[1]),2.),
   pow(cabs(y[0]),2.)+pow(cabs(y[1]),2.));

      // Calcula as derivadas
      derivadas(t0, y, dy);

      // Runge-Kutta 2ª ordem
      k1[0] = h * dy[0];
      k1[1] = h * dy[1];

    derivadas(t0 + h / 2, (double complex[]){y[0] + k1[0] / 2, y[1] + k1[1] / 2}, dy);

      k2[0] = h * dy[0];
```

```
      k2[1] = h * dy[1];
      y[0] += k2[0];
      y[1] += k2[1];

      t0 += h;
    }

    fclose(file);
    free(y);
    free(dy);
    free(k1);
    free(k2);

    return 0;
}
```

O programa fornecido simula a dinâmica de um sistema de dois sítios utilizando o método de Runge-Kutta de 2ª ordem para resolver equações diferenciais. A seguir, apresento uma análise detalhada do código e uma interpretação dos resultados. O código inclui as bibliotecas padrão para operações de entrada/saída, manipulação de números complexos e funções matemáticas. A macro `NMAX` define o tamanho máximo para o número de passos, embora não seja usada diretamente no código fornecido. As variáveis globais são definidas para o número de passos, o passo de integração, as energias dos sítios e o parâmetro de acoplamento. Essas variáveis são usadas em todo o código para configurar e controlar a simulação. A função `derivadas` calcula as derivadas das funções de onda `psi_1` e `psi_2`. As equações diferenciais são baseadas no modelo de Anderson, onde a unidade imaginária `I` é usada para lidar com o aspecto temporal das funções de onda. A função `main` realiza as seguintes etapas:

- Alocação dinâmica de memória para armazenar as variáveis de estado e os incrementos calculados.
- Definição das condições iniciais para as funções de onda.
- Criação e abertura do arquivo `resultado.dat` para gravação dos resultados.
- Loop de integração que utiliza o método de Runge-Kutta de 2ª ordem para atualizar as funções de onda ao longo do tempo. Os resultados são gravados no arquivo.
- Liberação da memória alocada antes de encerrar o programa.

O arquivo `resultado.dat` gerado pelo programa contém quatro colunas:

- Tempo (`t`): O tempo atual da simulação.
- $|\psi_1(t)|^2$: O quadrado da magnitude da função de onda no sítio 1.
- $|\psi_2(t)|^2$: O quadrado da magnitude da função de onda no sítio 2.
- $|\psi_1(t)|^2 + |\psi_2(t)|^2$: A soma dos quadrados das magnitudes das funções de onda, representando a norma total da função de onda. Esta norma deve ser igual a 1 para todo o tempo. Se esta quantidade for muito diferente de 1 temos um forte indício que o dt deve ser diminuído e/ou o método de integração não é bom para este problema.

Os resultados permitem analisar a distribuição da função de onda entre os dois sítios ao longo do tempo. A norma total da função de onda deve permanecer constante, indicando que o sistema é conservativo. As variações na distribuição de $|\psi_1(t)|^2$ e $|\psi_2(t)|^2$ refletem a transferência de probabilidade entre os sítios devido ao acoplamento.

0.8 O Modelo SIR e a Solução com o Método RK2

O **Modelo SIR** é um dos modelos mais simples usados na epidemiologia para descrever a propagação de doenças infecciosas. Ele é dividido em três compartimentos:

- $S(t)$: População suscetível que pode contrair a doença.
- $I(t)$: População infectada que pode transmitir a doença.
- $R(t)$: População removida (recuperada ou falecida), que não pode mais transmitir a doença.

Esses compartimentos evoluem ao longo do tempo t de acordo com o seguinte sistema de equações diferenciais:

$$\frac{dS}{dt} = -\beta SI,$$

$$\frac{dI}{dt} = \beta SI - \gamma I,$$

$$\frac{dR}{dt} = \gamma I,$$

onde:

- β é a taxa de transmissão, ou seja, a taxa com que indivíduos suscetíveis contraem a doença ao entrar em contato com infectados.
- γ é a taxa de recuperação, que representa a fração de indivíduos infectados que se recuperam (ou são removidos) por unidade de tempo.

Para resolver numericamente o sistema de equações diferenciais do modelo SIR, podemos usar o **método de Runge-Kutta de 2ª ordem (RK2)**. Este método é um dos métodos numéricos mais conhecidos para a solução de equações diferenciais ordinárias (EDOs) e proporciona uma aproximação de segunda ordem para a solução das EDOs. O RK2 utiliza dois passos intermediários para atualizar as variáveis de interesse em cada intervalo de tempo Δt. O método aplicado às equações do modelo SIR segue os seguintes passos:

1. Primeiro, calculamos os valores intermediários para S, I e R no meio do intervalo Δt:

$$k1_S = -\beta SI, \quad k1_I = \beta SI - \gamma I, \quad k1_R = \gamma I.$$

2. Em seguida, usamos esses valores para calcular os estados intermediários S_{mid}, I_{mid}, R_{mid}:

$$S_{mid} = S_n + 0.5 \cdot \Delta t \cdot k1_S,$$

$$I_{mid} = I_n + 0.5 \cdot \Delta t \cdot k1_I,$$

$$R_{mid} = R_n + 0.5 \cdot \Delta t \cdot k1_R.$$

3. Por fim, calculamos os incrementos $k2$ e atualizamos as populações S, I e R:

$$k2_S = -\beta S_{mid} I_{mid}, \quad k2_I = \beta S_{mid} I_{mid} - \gamma I_{mid}, \quad k2_R = \gamma I_{mid}.$$

4. As novas populações são então calculadas como:

$$S_{n+1} = S_n + \Delta t \cdot k2_S,$$

$$I_{n+1} = I_n + \Delta t \cdot k2_I,$$

$$R_{n+1} = R_n + \Delta t \cdot k2_R.$$

A seguir, temos um código em C que implementa o modelo SIR usando o método RK2 para integrar numericamente as equações diferenciais do modelo.

```
#include <stdio.h>

void rk2_step(double *S, double *I, double *R, double beta, double gamma, double h) {
    // Valores intermediários (meio passo de tempo)
    double k1_S, k1_I, k1_R;
    double k2_S, k2_I, k2_R;

    // Equações diferenciais do modelo SIR
    k1_S = -beta * (*S) * (*I);
    k1_I = beta * (*S) * (*I) - gamma * (*I);
    k1_R = gamma * (*I);

    double S_mid = *S + 0.5 * h * k1_S;
    double I_mid = *I + 0.5 * h * k1_I;
    double R_mid = *R + 0.5 * h * k1_R;

    k2_S = -beta * S_mid * I_mid;
    k2_I = beta * S_mid * I_mid - gamma * I_mid;
    k2_R = gamma * I_mid;

    // Atualizar os valores de S, I e R
    *S += h * k2_S;
    *I += h * k2_I;
```

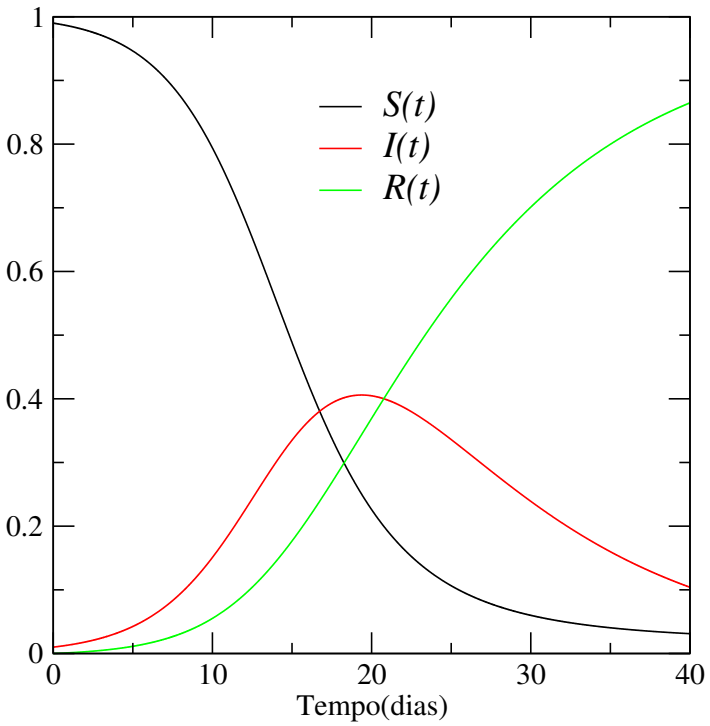

Figure 2: Evolução das populações suscetíveis (curva preta), infectadas (curva vermelha) e recuperadas (curva verde) ao longo do tempo. Inicialmente, a maioria da população é suscetível ($S(t)$), enquanto o número de infectados ($I(t)$) aumenta rapidamente após um ponto inicial. Atinge um pico antes de começar a decair à medida que mais indivíduos se recuperam ($R(t)$) e a população suscetível diminui. Ao final, a curva de infectados se aproxima de zero, enquanto a maior parte da população migra para o compartimento dos recuperados.

```
    *R += h * k2_R;
}

int main() {
    // Parâmetros do modelo SIR
    double beta = 0.4;  // Taxa de transmissão
    double gamma = 0.1; // Taxa de recuperação
    double S = 0.99;    // População inicial suscetível (99%)
    double I = 0.01;    // População inicial infectada (1%)
    double R = 0.0;     // População inicial recuperada (0%)
    double h = 0.01;     // Passo de tempo
    int steps = 4000;    // Número de passos de tempo

    // Abrir o arquivo para escrita
    FILE *file = fopen("SIR.dat", "w");
    if (file == NULL) {
        printf("Erro ao abrir o arquivo!\n");
        return 1;
    }

    // Loop para calcular as populações ao longo do tempo
    for (int i = 0; i < steps; i++) {
```

```
        double t = i * h;
        fprintf(file, "%.2f\t\t%.6f\t%.6f\t%.6f\n", t, S, I, R);
        rk2_step(&S, &I, &R, beta, gamma, h);
    }

    // Fechar o arquivo
    fclose(file);

    printf("Resultados gravados no arquivo 'SIR.dat'.\n");

    return 0;
}
```

O modelo SIR oferece uma visão simples, porém poderosa, da dinâmica de epidemias. As populações $S(t)$, $I(t)$ e $R(t)$ evoluem ao longo do tempo e, ao resolver as equações com o método RK2, conseguimos prever o comportamento da doença ao longo de um intervalo de tempo.

- A **população suscetível** $S(t)$ começa alta, mas diminui à medida que os indivíduos são infectados.
- A **população infectada** $I(t)$ inicialmente cresce, atinge um pico e depois diminui, à medida que os infectados se recuperam ou são removidos.
- A **população recuperada** $R(t)$ aumenta ao longo do tempo conforme os indivíduos se recuperam da infecção.

O método RK2, sendo um método de segunda ordem, oferece uma precisão razoável para a solução das EDOs, com um custo computacional menor comparado a métodos de ordem superior. Para uma simulação precisa, o tamanho do passo Δt deve ser escolhido de forma adequada. Passos menores aumentam a precisão, mas também aumentam o tempo de execução da simulação. Este código pode ser modificado para explorar diferentes cenários ajustando os valores de β, γ, o tamanho da população inicial S, I e R, ou o tamanho do passo Δt. A simulação também pode ser expandida para incluir efeitos mais complexos, como variações sazonais em β e γ, ou o impacto de intervenções, como vacinas ou quarentenas. O modelo SIR, resolvido aqui com o método de Runge-Kutta de 2ª ordem, é uma ferramenta fundamental para a compreensão da propagação de doenças infecciosas. O uso de técnicas numéricas como o RK2 permite resolver o sistema de equações diferenciais de forma eficiente, oferecendo insights importantes para a dinâmica epidemiológica.

0.9 Resolução do Modelo de Anderson com $N = 3$ Átomos Usando Runge-Kutta de Ordem 3 (RK3)

Neste exemplo, abordamos a solução do problema do modelo de Anderson com $N = 3$ átomos usando o método de Runge-Kutta de terceira ordem (RK3). O objetivo é resolver as equações diferenciais associadas ao sistema, considerando desordem dentro do intervalo $[-1, 1]$. Para $N = 3$, as equações diferenciais do modelo de Anderson considerando ($\hbar = 1$) são dadas por:

$$i\frac{d\psi_1}{dt} = -\psi_2 + \epsilon_1\psi_1$$

$$i\frac{d\psi_2}{dt} = -\psi_1 - \psi_3 + \epsilon_2\psi_2$$

$$i\frac{d\psi_3}{dt} = -\psi_2 + \epsilon_3\psi_3$$

onde $\psi_i(t)$ é a função de onda do i-ésimo átomo e ϵ_i é o termo de desordem para o i-ésimo átomo, variando aleatoriamente no intervalo $[-1, 1]$. O método de Runge-Kutta de 3ª ordem exige o cálculo de três coeficientes k_1, k_2 e k_3 para cada função ψ_i, onde $i = 1, 2, 3$.

O primeiro coeficiente é calculado a partir das equações diferenciais:

$$k_1^{(1)} = \Delta t \cdot (-i\,(-\psi_2 + \epsilon_1\psi_1))$$

$$k_1^{(2)} = \Delta t \cdot (-i\,(-\psi_1 - \psi_3 + \epsilon_2\psi_2))$$

$$k_1^{(3)} = \Delta t \cdot (-i\,(-\psi_2 + \epsilon_3\psi_3))$$

Utilizando os valores intermediários:

$$\tilde{\psi}_1 = \psi_1 + \frac{k_1^{(1)}}{2}, \quad \tilde{\psi}_2 = \psi_2 + \frac{k_1^{(2)}}{2}, \quad \tilde{\psi}_3 = \psi_3 + \frac{k_1^{(3)}}{2}$$

Calculamos os coeficientes k_2:

$$k_2^{(1)} = \Delta t \cdot \left(-i\left(-\tilde{\psi}_2 + \epsilon_1 \tilde{\psi}_1\right)\right)$$

$$k_2^{(2)} = \Delta t \cdot \left(-i\left(-\tilde{\psi}_1 - \tilde{\psi}_3 + \epsilon_2 \tilde{\psi}_2\right)\right)$$

$$k_2^{(3)} = \Delta t \cdot \left(-i\left(-\tilde{\psi}_2 + \epsilon_3 \tilde{\psi}_3\right)\right)$$

Agora, usamos os valores intermediários:

$$\hat{\psi}_1 = \psi_1 - k_1^{(1)} + 2k_2^{(1)}, \quad \hat{\psi}_2 = \psi_2 - k_1^{(2)} + 2k_2^{(2)}, \quad \hat{\psi}_3 = \psi_3 - k_1^{(3)} + 2k_2^{(3)}$$

Para calcular k_3:

$$k_3^{(1)} = \Delta t \cdot \left(-i\left(-\hat{\psi}_2 + \epsilon_1 \hat{\psi}_1\right)\right)$$

$$k_3^{(2)} = \Delta t \cdot \left(-i\left(-\hat{\psi}_1 - \hat{\psi}_3 + \epsilon_2 \hat{\psi}_2\right)\right)$$

$$k_3^{(3)} = \Delta t \cdot \left(-i\left(-\hat{\psi}_2 + \epsilon_3\hat{\psi}_3\right)\right)$$

Finalmente, atualizamos os valores de ψ_1, ψ_2 e ψ_3:

$$\psi_1(t+h) = \psi_1(t) + \frac{1}{6}\left(k_1^{(1)} + 4k_2^{(1)} + k_3^{(1)}\right)$$

$$\psi_2(t+h) = \psi_2(t) + \frac{1}{6}\left(k_1^{(2)} + 4k_2^{(2)} + k_3^{(2)}\right)$$

$$\psi_3(t+h) = \psi_3(t) + \frac{1}{6}\left(k_1^{(3)} + 4k_2^{(3)} + k_3^{(3)}\right)$$

O método de Runge-Kutta de 3ª ordem fornece uma aproximação numérica eficiente para a solução deste sistema de equações diferenciais. A cada passo de tempo Δt, calculamos os coeficientes intermediários k_1, k_2 e k_3, e então atualizamos os valores de ψ_1, ψ_2 e ψ_3. Uma implementação deste procedimento em C pode ser encontrando a seguir:

```
#include <stdio.h>
#include <stdlib.h>
#include <complex.h>
#include <math.h>

#define NMAX 10000
#define T 1.0 // Parâmetro de acoplamento

int N; // Número de sítios
double h; // Passo de integração
double complex *epsilon; // Energias dos sítios

void derivadas(double t, double complex *y, double complex *dy) {
    dy[0] = -I*(-y[1] + epsilon[0] * y[0]);
    dy[1] = -I*(-y[0] - y[2] + epsilon[1] * y[1]);
    dy[2] = -I*(-y[1] + epsilon[2] * y[2]);
}

int main() {
    double complex *y = (double complex *)malloc(4 * sizeof(double complex));
```

```
double complex *dy = (double complex *)malloc(4 * sizeof(double complex));
double complex *k1 = (double complex *)malloc(4 * sizeof(double complex));
double complex *k2 = (double complex *)malloc(4 * sizeof(double complex));
double complex *k3 = (double complex *)malloc(4 * sizeof(double complex));
double complex *y_temp = (double complex *)malloc(4 * sizeof(double complex));
double t_final = 10.0; // Tempo final
h = 0.01; // Passo de integração
N = 3; // Número de sítios
epsilon = (double complex *)malloc(N * sizeof(double complex));

// Inicialização das energias dos sítios
for (int i = 0; i < N; i++) {
    epsilon[i] = 2.0 * (((double)rand() / RAND_MAX) - 0.5);
}

// Condições iniciais
for (int i = 0; i < N; i++) {
    y[i] = 0.0 + 0.0*I;
}
y[1] = 1.0 + 0.0*I; // Função de onda localizada no segundo sítio

FILE *file = fopen("sigma3atom.dat", "w");

for (double t = h; t <= t_final; t += h) {
    double norma = 0.0;
    double sigma = 0.0;
    double soma_i = 0.0;
    double soma_i2 = 0.0;

    // Calcula a norma e o desvio médio quadrático
    for (int i = 0; i < N; i++) {
        norma += creal(y[i]) * creal(y[i]) + cimag(y[i]) * cimag(y[i]);
        soma_i += i * (creal(y[i]) * creal(y[i]) + cimag(y[i]) * cimag(y[i]));
        soma_i2 += i * i * (creal(y[i]) * creal(y[i]) + cimag(y[i]) * cimag(y[i]));
    }
    double media_i = soma_i / norma;
    sigma = soma_i2 / norma - media_i * media_i;

     if (t>h) fprintf(file, "%.3f %17.6g %17.6g\n", t, sigma, norma);

    // Runge-Kutta 3ª ordem
    derivadas(t, y, dy);

    for (int i = 0; i < N; i++) {
        k1[i] = h * dy[i];
    }

    for (int i = 0; i < N; i++) {
        y_temp[i] = y[i] + k1[i] / 2;
    }
    derivadas(t + h / 2, y_temp, dy);

    for (int i = 0; i < N; i++) {
        k2[i] = h * dy[i];
    }

    for (int i = 0; i < N; i++) {
        y_temp[i] = y[i] - k1[i] + 2 * k2[i];
    }
```

```
        derivadas(t + h, y_temp, dy);

        for (int i = 0; i < N; i++) {
            k3[i] = h * dy[i];
        }

        for (int i = 0; i < N; i++) {
            y[i] += (k1[i] + 4.*k2[i] + k3[i]) / 6.;
        }
    }

    fclose(file);
    free(y);
    free(dy);
    free(k1);
    free(k2);
    free(k3);
    free(y_temp);
    free(epsilon);

    return 0;
}
```

O programa gera um arquivo chamado `sigma3atom.dat` contendo os valores de sigma e norma ao longo do tempo. Esses valores são calculados para monitorar o comportamento do sistema ao longo da integração temporal.

0.10 Método de Runge-Kutta de 4ª ordem (RK4) : exemplos

O método de Runge-Kutta de 4ª ordem é amplamente utilizado devido à sua precisão e eficiência. Ele é uma aproximação numérica de quarta ordem para resolver equações diferenciais ordinárias e é mais preciso do que os métodos de ordem inferior, como o de 2ª ou 3ª ordem. A fórmula geral para o método de Runge-Kutta de 4ª ordem é:

$$y_{n+1} = y_n + \frac{\Delta t}{6}(k_1 + 2k_2 + 2k_3 + k_4),$$

onde:

$$k_1 = f(t_n, y_n),$$

$$k_2 = f\left(t_n + \frac{\Delta t}{2}, y_n + \frac{\Delta t}{2} k_1\right),$$

$$k_3 = f\left(t_n + \frac{\Delta t}{2}, y_n + \frac{\Delta t}{2} k_2\right),$$

$$k_4 = f(t_n + \Delta t, y_n + \Delta t k_3).$$

Aqui, Δt é o tamanho do passo, t_n e y_n são os valores de tempo e função no n-ésimo passo, respectivamente. Considere a seguinte equação diferencial ordinária:

$$y'(t) = -y + t^2 + 1, \quad y(0) = 0.5.$$

Vamos aplicar o método de Runge-Kutta de 4ª ordem com $\Delta t = 0.1$.

```
#include <stdio.h>

double y, h, t, k1, k2, k3, k4;
int n, N;

int main() {
    y = 0.5;    // valor inicial de y(0)
    h = 0.1;    // tamanho do passo
    t = 0.0;    // tempo inicial
    N = 10;     // número de iterações

    // Loop do método de Runge-Kutta 4ª ordem
    for (n = 0; n < N; n++) {
        printf("t = %lf, y = %lf\n", t, y);

     // Cálculo dos coeficientes k1, k2, k3 e k4
     k1 = -y + t * t + 1;
     k2 = -(y + h * k1 / 2.0) + (t + h / 2.0) * (t + h / 2.0) + 1;
     k3 = -(y + h * k2 / 2.0) + (t + h / 2.0) * (t + h / 2.0) + 1;
     k4 = -(y + h * k3) + (t + h) * (t + h) + 1;

     // Atualização de y e t
     y = y + (h / 6.0) * (k1 + 2 * k2 + 2 * k3 + k4);
     t = t + h;
    }

    return 0;
}
```

Neste exemplo, o método de Runge-Kutta de 4ª ordem é aplicado à equação diferencial $y'(t) = -y + t^2 + 1$, com condição inicial $y(0) = 0.5$ e passo $\Delta t = 0.1$. Ele fornece uma aproximação precisa da solução em um intervalo de tempo entre $t = 0$ e $t = 1$.

0.11 Modelo de Anderson 1D com N Sítios : solução usando RK4

Nesta seção, vamos resolver o modelo de Anderson unidimensional com N sítios utilizando o método de Runge-Kutta de quarta ordem. O modelo considera uma cadeia unidimensional de sítios com energias aleatórias e acoplamento entre os sítios. Vamos calcular a função de onda em cada sítio e analisar a evolução temporal da norma da função de onda e o desvio médio quadrático da função de onda. O modelo de Anderson unidimensional é descrito pela seguinte equação de Schrödinger com potencial desordenado:

$$i\frac{\partial \psi_i(t)}{\partial t} = \epsilon_i \psi_i(t) - J(\psi_{i+1}(t) + \psi_{i-1}(t)), \tag{1}$$

onde ϵ_i é a energia do sítio i, $J = 1$ é o acoplamento entre sítios vizinhos e $\psi_i(t)$ é a função de onda no sítio i. Para resolver essa equação numericamente, usamos o método de Runge-Kutta de quarta ordem. A função de onda $\psi_i(t)$ é atualizada em cada passo de tempo Δt. Uma das maneiras de medir o comportamento da função de onda é através do cálculo do desvio médio quadrático (DMC) dado por:

$$\sigma = \sum_i (i - \langle i \rangle)^2 |\psi_i(t)|^2, \tag{2}$$

onde $\langle i \rangle = \sum_i i|\psi_i(t)|^2$ é a posição média ponderada pela função de onda e $|\psi_i(t)|^2$ é a densidade de probabilidade no sítio i. O código a seguir implementa o modelo de Anderson unidimensional e resolve a equação de Schrödinger usando o método de Runge-Kutta de quarta ordem. O código calcula o desvio médio quadrático σ e a norma da função de onda e salva os resultados em um arquivo chamado `sigma1d.dat`.

```
#include <stdio.h>
#include <stdlib.h>
#include <complex.h>
```

```
#include <math.h>

#define NMAX 10000
#define T 1.0 // Parâmetro de acoplamento

int N; // Número de sítios
double h; // Passo de integração
double complex *epsilon; // Energias dos sítios

void derivadas(double t, double complex *y, double complex *dy) {
    for (int i = 0; i < N; i++) {
        dy[i] = -I * (epsilon[i] * y[i] -  (y[i+1] + y[i-1]));
    }
}

int main() {
  double complex *y = (double complex *)malloc(NMAX * sizeof(double complex));
  double complex *dy = (double complex *)malloc(NMAX * sizeof(double complex));
  double complex *k1 = (double complex *)malloc(NMAX * sizeof(double complex));
  double complex *k2 = (double complex *)malloc(NMAX * sizeof(double complex));
  double complex *k3 = (double complex *)malloc(NMAX * sizeof(double complex));
  double complex *k4 = (double complex *)malloc(NMAX * sizeof(double complex));
  double complex *y1 = (double complex *)malloc(NMAX * sizeof(double complex));
  double t_final = 100.0; // Tempo final
    int c100;
    h = 0.01; // Passo de integração
    N = 200; // Número de sítios
    epsilon = (double complex *)malloc(N * sizeof(double complex));

    // Inicialização das energias dos sítios
    for (int i = 0; i < N; i++) {
        epsilon[i] =2.*(((double)rand() / RAND_MAX) - 0.5);
    }

    // Condições iniciais
    for (int i = 0; i < N; i++) {
        y[i] = 0.0 + 0.0*I;
    }
    y[N/2] = 1.0 + 0.0*I; // Função de onda localizada no sítio central

    FILE *file = fopen("sigma1d.dat", "w");


   for (double t = h; t <= t_final; t += h) {
      double norma = 0.0;
      double sigma = 0.0;
      double soma_i = 0.0;
      double soma_i2 = 0.0;
      c100=c100+1;
      // Calcula a norma e o desvio médio quadrático
      for (int i = 0; i < N; i++) {
        norma += creal(y[i]) * creal(y[i]) + cimag(y[i]) * cimag(y[i]);
        soma_i += i * (creal(y[i]) * creal(y[i]) + cimag(y[i]) * cimag(y[i]));
        soma_i2 += i * i * (creal(y[i]) * creal(y[i]) + cimag(y[i]) * cimag(y[i]));
     }
       double media_i = soma_i / norma;
       sigma = soma_i2 / norma - media_i * media_i;

      if (c100>5) {
```

```
            fprintf(file, "%.3f %17.6g %17.6g\n", t, sigma, norma);
        c100=0;
    }
        // Runge-Kutta 4ª ordem
        derivadas(t, y, dy);

        for (int i = 0; i < N; i++) {
            k1[i] = h * dy[i];
        }

        for (int i = 0; i < N; i++) {
            y1[i] =y[i]+k1[i] / 2;
        }
        derivadas(t + h / 2, y1, dy);

        for (int i = 0; i < N; i++) {
            k2[i] = h * dy[i];
        }

        for (int i = 0; i < N; i++) {
            y1[i] =y[i]+k2[i] / 2;
        }
        derivadas(t + h / 2, y1, dy);

        for (int i = 0; i < N; i++) {
            k3[i] = h * dy[i];
        }

        for (int i = 0; i < N; i++) {
            y1[i] =y[i]+k3[i];
        }
        derivadas(t + h, y1, dy);

        for (int i = 0; i < N; i++) {
            k4[i] = h * dy[i];
        }

        for (int i = 0; i < N; i++) {
            y[i] += (k1[i] + 2.*k2[i] + 2.*k3[i] + k4[i]) / 6.;
        }
    }

    fclose(file);
    free(y);
    free(dy);
    free(k1);
    free(k2);
    free(k3);
    free(k4);
    free(epsilon);

    return 0;
}
```

Este código resolve o modelo de Anderson unidimensional usando o método de Runge-Kutta de quarta ordem. A função de onda é inicializada com um valor localizado no sítio central, e as energias dos sítios são atribuídas aleatoriamente. O programa calcula a evolução temporal da função de onda e de-

termina o desvio médio quadrático (DMC) e a norma da função de onda. O DMC é usado para avaliar a dispersão da função de onda, enquanto a norma representa a probabilidade total. Os resultados são salvos no arquivo `sigma1d.dat`, onde o tempo, o DMC e a norma são registrados. A análise desses dados oferece uma visão sobre a propagação e o espalhamento da função de onda no sistema desordenado.

0.12 Modelo de Anderson 2D : solução usando RK4

O modelo de Anderson em duas dimensões descreve a dinâmica de uma partícula quântica sujeita a desordem, representada por uma rede quadrada 2D com $N \times N$ sítios. A equação diferencial que rege a evolução temporal da função de onda $\psi_{i,j}(t)$ neste sistema é dada por:

$$i\frac{\partial \psi_{i,j}}{\partial t} = -\left(\psi_{i+1,j} + \psi_{i-1,j} + \psi_{i,j+1} + \psi_{i,j-1}\right) + \epsilon_{i,j}\psi_{i,j},$$

onde: - $\psi_{i,j}(t)$ representa a função de onda no sítio (i, j) da rede em um instante de tempo t, - $\epsilon_{i,j}$ são as energias dos sítios, que introduzem a desordem no sistema, sendo aleatoriamente distribuídas no intervalo $[-6, 6]$, - $i \pm 1, j \pm 1$ indicam os sítios vizinhos na rede 2D, - O termo $-\left(\psi_{i+1,j} + \psi_{i-1,j} + \psi_{i,j+1} + \psi_{i,j-1}\right)$ corresponde ao acoplamento entre a função de onda em (i, j) e seus vizinhos mais próximos, - O termo $\epsilon_{i,j}\psi_{i,j}$ reflete a interação local da função de onda com a desordem.

Para resolver numericamente essa equação diferencial parcial, aplicamos o método de Runge-Kutta de quarta ordem, que proporciona uma alta precisão na evolução temporal da função de onda. O método é adequado para sistemas de equações diferenciais como este, em que a dinâmica depende tanto das interações entre os sítios quanto da desordem aleatória introduzida pelas energias $\epsilon_{i,j}$. Inicialmente, as energias $\epsilon_{i,j}$ em cada sítio da rede são sorteadas de forma aleatória dentro do intervalo $[-6, 6]$. A função de onda $\psi_{i,j}(t)$ é definida inicialmente como não nula em um sítio central da rede (geralmente assumimos $\psi_{N/2,N/2} = 1$) e zero em todos os demais sítios. Durante a evolução temporal, uma das quantidades de interesse é o desvio médio quadrático (DMC) da função de onda, que mede a dispersão espacial da partícula na rede. O DMC é calculado pela expressão:

$$\sigma = \sum_{i,j} \left[(i - \langle i \rangle)^2 + (j - \langle j \rangle)^2 \right] |\psi_{i,j}|^2,$$

onde $\langle i \rangle$ e $\langle j \rangle$ são os valores médios das coordenadas i e j, respectivamente, ponderados pela densidade de probabilidade $|\psi_{i,j}|^2$, ou seja:

$$\langle i \rangle = \sum_{i,j} i |\psi_{i,j}|^2 \quad \text{e} \quad \langle j \rangle = \sum_{i,j} j |\psi_{i,j}|^2.$$

Esses valores médios fornecem a posição esperada da partícula na rede, e o DMC quantifica o quanto a partícula se espalhou em torno dessa posição média. A seguir, o código em C implementa o método de Runge-Kutta de quarta ordem para calcular a evolução temporal de $\psi_{i,j}(t)$, bem como a norma da função de onda e o DMC ao longo do tempo. A norma serve como um critério de verificação para garantir que a função de onda esteja devidamente normalizada durante a evolução.

```
#include <stdio.h>
#include <stdlib.h>
#include <complex.h>
#include <math.h>

#define NMAX 100
#define T 1.0 // Parâmetro de acoplamento

int N; // Número de sítios em cada dimensão
double h; // Passo de integração
double complex epsilon[NMAX][NMAX]; // Energias dos sítios

void derivadas(double t, double complex y[NMAX][NMAX], double complex dy[NMAX][NMAX]) {
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            int idx = i * N + j;
            double complex term1 = (i > 0 ? y[i-1][j] : 0) +
                                   (i < N-1 ? y[i+1][j] : 0) +
                                   (j > 0 ? y[i][j-1] : 0) +
                                   (j < N-1 ? y[i][j+1] : 0);
            dy[i][j] = -I * (epsilon[i][j] * y[i][j] - term1);
        }
    }
}

int main() {
  double complex y[NMAX][NMAX];
  double complex dy[NMAX][NMAX];
  double complex k1[NMAX][NMAX], k2[NMAX][NMAX], k3[NMAX][NMAX], k4[NMAX][NMAX];
```

```
double complex y1[NMAX][NMAX];
  double t_final = 100.0; // Tempo final
  int c100 = 0;
  h = 0.01; // Passo de integração
  N = 50; // Número de sítios em cada dimensão

  // Inicialização das energias dos sítios
  for (int i = 0; i < N; i++) {
      for (int j = 0; j < N; j++) {
          epsilon[i][j] = 12.0 * ((double)rand() / RAND_MAX - 0.5);
      }
  }

  // Condições iniciais
  for (int i = 0; i < N; i++) {
      for (int j = 0; j < N; j++) {
          y[i][j] = 0.0 + 0.0*I;
      }
  }
  y[N/2][N/2] = 1.0 + 0.0*I; // Função de onda localizada no sítio central

  FILE *file = fopen("sigma2d.dat", "w");

  for (double t = h; t <= t_final; t += h) {
      double norma = 0.0;
      double sigma = 0.0;
      double soma_i = 0.0;
      double soma_j = 0.0;
      double soma_i2 = 0.0;
      double soma_j2 = 0.0;
      c100++;
      // Calcula a norma e o desvio médio quadrático
    for (int i = 0; i < N; i++) {
      for (int j = 0; j < N; j++) {
        double abs_y = creal(y[i][j]) * creal(y[i][j]) + cimag(y[i][j]) * cimag(y[i][j]);
        norma += abs_y;
        soma_i += i * abs_y;
        soma_j += j * abs_y;
        soma_i2 += i * i * abs_y;
        soma_j2 += j * j * abs_y;
        }
     }
   double media_i = soma_i / norma;
   double media_j = soma_j / norma;
   sigma = (soma_i2 / norma - media_i * media_i) + (soma_j2 / norma - media_j * media_j);

     if (c100 > 5) {
         fprintf(file, "%.3f %17.6g %17.6g\n", t, sigma, norma);
         c100 = 0;
      }

      // Runge-Kutta 4ª ordem
      derivadas(t, y, dy);

   for (int i = 0; i < N; i++) {
      for (int j = 0; j < N; j++) {
           k1[i][j] = h * dy[i][j];
        }
   }
```

```
        for (int i = 0; i < N; i++) {
            for (int j = 0; j < N; j++) {
                y1[i][j] = y[i][j] + k1[i][j] / 2;
            }
        }
        derivadas(t + h / 2, y1, dy);

        for (int i = 0; i < N; i++) {
            for (int j = 0; j < N; j++) {
                k2[i][j] = h * dy[i][j];
            }
        }

        for (int i = 0; i < N; i++) {
            for (int j = 0; j < N; j++) {
                y1[i][j] = y[i][j] + k2[i][j] / 2;
            }
        }
        derivadas(t + h / 2, y1, dy);

        for (int i = 0; i < N; i++) {
            for (int j = 0; j < N; j++) {
                k3[i][j] = h * dy[i][j];
            }
        }

        for (int i = 0; i < N; i++) {
            for (int j = 0; j < N; j++) {
                y1[i][j] = y[i][j] + k3[i][j];
            }
        }
        derivadas(t + h, y1, dy);

        for (int i = 0; i < N; i++) {
            for (int j = 0; j < N; j++) {
                k4[i][j] = h * dy[i][j];
            }
        }

     for (int i = 0; i < N; i++) {
       for (int j = 0; j < N; j++) {
          y[i][j] += (k1[i][j] + 2.0 * k2[i][j] + 2.0 * k3[i][j] + k4[i][j]) / 6.0;
       }
     }
   }

    fclose(file);
    return 0;
}
```

O programa implementa a solução do modelo de Anderson 2D usando o método de Runge-Kutta de quarta ordem. Inicialmente, as energias dos sítios são aleatoriamente distribuídas no intervalo $[-6, 6]$, e a função de onda é centralizada. Os resultados são armazenados em um arquivo chamado 'sigma2d.dat', contendo o valor do desvio médio quadrático e da norma a

cada passo de tempo.

0.13 Modelo de Anderson 1D com Termo Não Linear Diagonal

Neste capítulo, vamos resolver a equação de Schrödinger dependente do tempo para o modelo de Anderson unidimensional (1D) com um efeito não linear diagonal. O termo diagonal ϵ_n será proporcional ao módulo quadrado da função de onda em cada sítio, ou seja, $\epsilon_n = A|f_n(t)|^2$, onde A é um parâmetro ajustável. Este modelo descreve a evolução de uma função de onda em uma rede com desordem, onde o termo não linear adiciona uma complexidade adicional ao problema. A equação de Schrödinger dependente do tempo é dada por:

$$i\frac{d}{dt}f_n(t) = -J\left(f_{n+1}(t) + f_{n-1}(t)\right) + \epsilon_n f_n(t),$$

onde $f_n(t)$ é a função de onda no sítio n, J é o parâmetro de hopping (aqui considerado como $J = 1$ para simplificação), e $\epsilon_n = A|f_n(t)|^2$ é o termo não linear diagonal. Vamos considerar uma cadeia de $N = 100$ sítios, com o estado inicial da função de onda localizado no centro da cadeia. A função de onda inicial será definida como:

$$f_n(0) = \begin{cases} 1, & \text{se } n = \frac{N}{2}, \\ 0, & \text{caso contrário.} \end{cases}$$

Para resolver a equação de Schrödinger numericamente, utilizamos o método de Runge-Kutta de 4ª ordem (RK4), que é um método numérico preciso e eficiente para integrar equações diferenciais. No método RK4, a função de onda em cada sítio n é atualizada ao longo do tempo usando a seguinte fórmula:

$$f_n(t + \Delta t) = f_n(t) + \frac{1}{6}\left(k_1 + 2k_2 + 2k_3 + k_4\right),$$

onde os incrementos k_1, k_2, k_3, e k_4 são definidos como:

$$k_1 = \Delta t \cdot f'(t),$$

$$k_2 = \Delta t \cdot f'\left(t + \frac{\Delta t}{2}\right),$$

$$k_3 = \Delta t \cdot f'\left(t + \frac{\Delta t}{2}\right),$$

$$k_4 = \Delta t \cdot f'(t + \Delta t).$$

A função $f'(t)$ é a derivada temporal da função de onda, que é calculada pela equação de Schrödinger. A derivada temporal da função de onda $f_n(t)$ é dada pela expressão:

$$\frac{d}{dt} f_n(t) = -i \left(J \left(f_{n+1}(t) + f_{n-1}(t) \right) + \epsilon_n f_n(t) \right),$$

onde $\epsilon_n = A|f_n(t)|^2$ é o termo não linear diagonal. A função de onda é então atualizada usando os incrementos do método RK4. A seguir, apresentamos a implementação do método RK4 em linguagem C, que resolve a equação de Schrödinger para este modelo com $N = 100$ sítios. O valor do parâmetro A é ajustável pelo usuário. Após a solução numérica podemos investigar a dinâmica eletrônica neste sistema através de quantidades física que medem a propagação (ou não) na presença de não-linearidade. Uma destas quantidades de interesse é a *probabilidade de retorno*, $P_{N/2}(t)$, que mede a probabilidade de a função de onda retornar ao sítio central da cadeia $N/2$ após um tempo t. Esta probabilidade é dada por:

$$P_{N/2}(t) = |f_{N/2}(t)|^2,$$

onde $f_{N/2}(t)$ é o valor da função de onda no sítio central $N/2$ em um tempo t. Para explorar o comportamento da probabilidade de retorno, realizamos

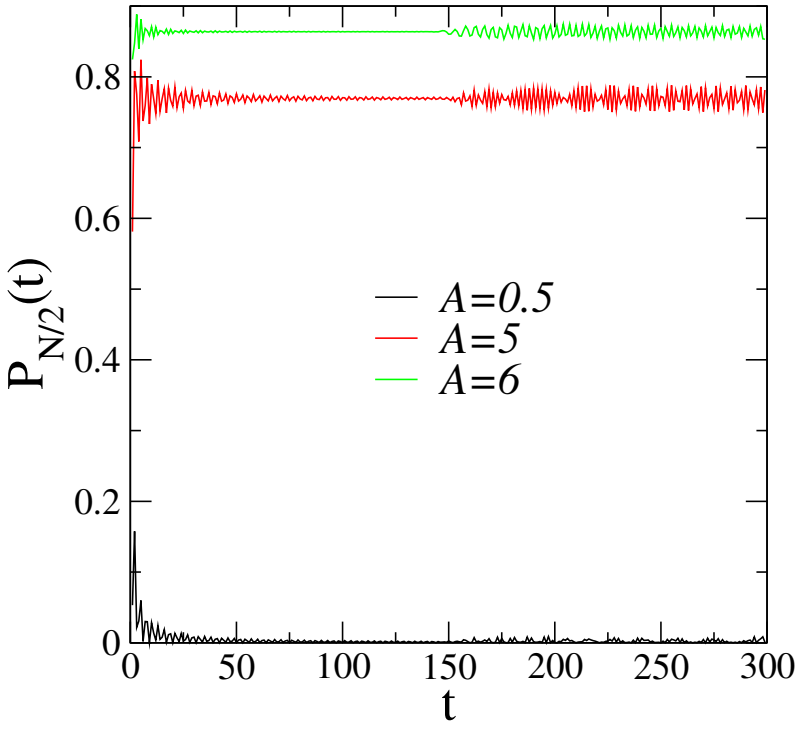

Figure 3: Probabilidade de retorno $P_{N/2}(t)$ para três valores de A: 0.5, 5, e 6.

simulações numéricas para três valores do parâmetro A, que controla a intensidade do termo não linear diagonal no modelo. Os valores considerados foram:

- $A = 0.5$,
- $A = 5$,
- $A = 6$.

O parâmetro A modifica diretamente o termo diagonal $\epsilon_n = A|f_n(t)|^2$, o que impacta a evolução da função de onda no tempo. Para cada valor de A, a função de onda foi inicializada com o estado localizado no centro da cadeia e sua evolução foi calculada usando o método de Runge-Kutta de 4ª ordem (RK4). A probabilidade de retorno foi registrada ao longo do tempo e será discutida em termos de seu comportamento para os diferentes regimes não lineares. Os gráficos resultantes de $P_{N/2}(t)$ para cada valor de A (ver fig. 3) mostram como a não linearidade afeta a dinâmica da função de onda.

```
#include <stdio.h>
#include <stdlib.h>
```

```
#include <math.h>
#include <complex.h>

#define N 200
#define J 1.0
#define DT 0.01
#define STEPS 20000

double A;  // Parâmetro ajustável

// Função para calcular o termo epsilon_n = A * |f_n(t)|^2
double epsilon(complex double f_n) {
    return A * pow(cabs(f_n), 2);
}

// Função para a derivada temporal da função de onda
void derivada(complex double f[], complex double dfdt[]) {
    for (int n = 0; n < N; n++) {
        complex double hopping = 0.0;
        if (n > 0) hopping += f[n - 1];
        if (n < N - 1) hopping += f[n + 1];
        dfdt[n] = -I * (J * hopping + epsilon(f[n]) * f[n]);
    }
}

// Método RK4 para evoluir a função de onda
void rk4(complex double f[], double dt) {
    complex double k1[N], k2[N], k3[N], k4[N];
    complex double f_temp[N];

    derivada(f, k1);

    for (int n = 0; n < N; n++) f_temp[n] = f[n] + 0.5 * dt * k1[n];
    derivada(f_temp, k2);

    for (int n = 0; n < N; n++) f_temp[n] = f[n] + 0.5 * dt * k2[n];
    derivada(f_temp, k3);

    for (int n = 0; n < N; n++) f_temp[n] = f[n] + dt * k3[n];
    derivada(f_temp, k4);

    for (int n = 0; n < N; n++) {
        f[n] = f[n] + (dt / 6.0) * (k1[n] + 2.0 * k2[n] + 2.0 * k3[n] + k4[n]);
    }
}

int main() {
    // Inicializar a função de onda
    complex double f[N] = {0.0 + 0.0 * I};
    f[N / 2] = 1.0 + 0.0 * I;  // Estado inicial localizado no centro

    // Leitura do parâmetro A
    printf("Digite o valor do parâmetro A: ");
    scanf("%lf", &A);

       char filename[20];
            sprintf(filename, "retorno_%f.dat", A);
            FILE *fp = fopen(filename, "w");
    // Evolução temporal
```

```
    for (int step = 1; step < STEPS; step++) {
        rk4(f, DT);

        // Salvar os dados da função de onda para análise (opcional)
        if (step % 100 == 0) {


                fprintf(fp, "%f %f\n", step*DT, pow(cabs(f[N/2]),2.));


        }
    }

    return 0;
}
```

Nesta seção, descrevemos a resolução numérica da equação de Schrödinger dependente do tempo para o modelo de Anderson 1D com um termo diagonal não linear. O método de Runge-Kutta de 4ª ordem foi utilizado para integrar a equação diferencial, e o código em C apresentado permite a simulação da evolução temporal da função de onda. O parâmetro A pode ser ajustado para explorar diferentes regimes não lineares.

0.14 Modelo de Anderson 1d : solução usando o método de Taylor

O método de Taylor para a solução da equação de Schrödinger é baseado na expansão de Taylor do operador de evolução temporal. Considerando o problema da equação de Schrödinger 1D com *hopping* $J = 1$:

$$i\hbar\frac{df_n}{dt} = \epsilon_n f_n + f_{n+1} + f_{n-1}, \quad n = 1, 2, 3, \ldots \tag{3}$$

A expansão de Taylor para o operador de evolução temporal $U(t) = e^{-\frac{iHt}{\hbar}}$ é dada por:

$$U(t) = e^{-\frac{iHt}{\hbar}} \approx 1 + \sum_{k=1}^{N_0} \left(-\frac{iHt}{\hbar}\right)^k \frac{1}{k!}, \tag{4}$$

onde H é o Hamiltoniano do modelo de Anderson 1D e N_0 é a ordem de truncamento da série (geralmente, $N_0 > 10$). Para o estado inicial

$|\psi(0)\rangle = \sum_n f_n(0)\,|n\rangle$, o estado $|\psi(t)\rangle$ pode ser expresso como:

$$|\psi(t)\rangle = e^{-\frac{iHt}{\hbar}}\,|\psi(0)\rangle = \left\{1 + \sum_{k=1}^{N_0}\left(-\frac{iHt}{\hbar}\right)^k \frac{1}{k!}\right\}|\psi(0)\rangle\,. \tag{5}$$

Para $t = \Delta t < 1$, o estado $|\psi(\Delta t)\rangle$ é obtido pela soma das potências do Hamiltoniano aplicadas ao estado $|\psi(0)\rangle$:

$$\sum_k \left(-\frac{iH\Delta t}{\hbar}\right)^k \frac{1}{k!}\,|\psi(0)\rangle = -\frac{iH\Delta t}{\hbar}\,|\psi(0)\rangle \tag{6}$$

$$+ \quad \left[\left(\frac{-i\Delta t}{\hbar}\right)^2 \frac{H^2}{2!} + \left(\frac{-i\Delta t}{\hbar}\right)^3 \frac{H^3}{3!} + \cdots\right]|\psi(0)\rangle\,, \tag{7}$$

Para calcular $H^k\,|\psi(0)\rangle = H^k \sum_n f_n(0)\,|n\rangle$, usamos:

$$H^1 \sum_n f_n(0)\,|n\rangle = \sum_n C_n^1\,|n\rangle = \sum_n \{\epsilon_n f_n(0) + [f_{n+1}(0) + f_{n-1}(0)]\}\,|n\rangle\,, \tag{8}$$

logo temos que :

$$C_n^1 = \epsilon_n f_n(0) + f_{n+1}(0) + f_{n-1}(0), \tag{9}$$

Lembrando que $H^2 \sum_n f_n(0)\,|n\rangle = \sum_n C_n^2\,|n\rangle = H[H\sum_n f_n(0)\,|n\rangle]$ temos que:

$$C_n^2 = \epsilon_n C_n^1 + C_{n+1}^1 + C_{n-1}^1. \tag{10}$$

Seguindo o mesmo raciocínio, podemos demonstrar que:

$$C_n^k = \epsilon_n C_n^{k-1} + C_{n+1}^{k-1} + C_{n-1}^{k-1}. \tag{11}$$

Para encontrar a solução numérica em tempo t_m, aplicamos o método sucessivamente. Recomenda-se $N_0 \approx 12$ e $\Delta t = 0.05$, pois são adequados

para manter a conservação da norma da função de onda. Este formalismo é aplicável em dimensões superiores (2D, 3D) e é geralmente mais rápido que o método de Runge-Kutta de quarta ordem, especialmente em sistemas de maior dimensão. A partir da solução, podemos calcular medidas do grau de localização, como participação ξ, desvio médio quadrático σ e entropia de Shannon $S(t)$. Estas quantidades são definidas como:

$$\xi(t) = (\sum_n |f_n(t)|^4)^{-1}, \tag{12}$$

$$\sigma(t) = \sqrt{\sum_n (n - \langle n(t) \rangle)^2 |f_n(t)|^2}, \tag{13}$$

$$S(t) = -\sum_n |f_n(t)|^2 \log\left(|f_n(t)|^2\right). \tag{14}$$

Abaixo está o código em C para implementar o modelo de Anderson 1D, calculando a evolução temporal da função de onda usando o método de Taylor:

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#define N 100       // Número de sites
#define N0 12       // Ordem de truncamento
#define DELTA_T 0.05 // Intervalo de tempo

// Função para inicializar as energias aleatoriamente
void inicializar_energies(double epsilon[]) {
 for (int i = 0; i < N; i++) {
  epsilon[i] = ((double)rand() / RAND_MAX) * 2.0 - 1.0; // Valores entre -1 e 1
  }
}

// Função para calcular a evolução temporal usando o método de Taylor
void evoluir_temporal(double epsilon[], double f[], double f_temp[]) {
    for (int k = 1; k <= N0; k++) {
        // Calcular H^k f
        for (int n = 0; n < N; n++) {
            f_temp[n] = epsilon[n] * f[n];
            if (n > 0) f_temp[n] += f[n-1];
            if (n < N-1) f_temp[n] += f[n+1];
        }
        // Atualizar f
        for (int n = 0; n < N; n++) {
            f[n] = f_temp[n];
        }
    }
}
```

```
// Função principal
int main() {
    double epsilon[N], f[N], f_temp[N];
    // Inicializar as energias e a função de onda
    inicializar_energies(epsilon);
    for (int i = 0; i < N; i++) {
        f[i] = 1.0 / sqrt(N); // Função de onda inicial
    }

    // Evolução temporal
    evoluir_temporal(epsilon, f, f_temp);

    // Salvar os resultados em um arquivo
    FILE *fp = fopen("resultados.txt", "w");
    for (int i = 0; i < N; i++) {
        fprintf(fp, "%d %f\n", i, f[i]);
    }
    fclose(fp);

    return 0;
}
```

Este código realiza a evolução temporal da função de onda para o modelo de Anderson 1D usando o método de Taylor. Ele inicializa as energias aleatoriamente, aplica o método de Taylor para atualizar a função de onda e salva os resultados em um arquivo de texto. Ajuste os parâmetros conforme necessário para obter resultados mais precisos.

0.15 Método de Adams de 2ª e 4ª Ordem

Nesta seção, abordaremos o método de Adams-Bashforth, com foco na sua dedução para a 2ª ordem, e aplicaremos esse método para resolver a equação diferencial $y'(t) = -y + t^2 + 1$, uma EDO comum em problemas de dinâmica e física matemática. O método de Adams-Bashforth é um método multistep, ou seja, ele utiliza valores anteriores da solução e da função derivada $y'(t)$ para avançar no tempo. A equação diferencial a ser resolvida é:

$$y'(t) = -y + t^2 + 1$$

com uma condição inicial $y(0) = y_0$. Iremos aplicar tanto o método de Adams-Bashforth de 2ª ordem quanto de 4ª ordem.

Método de Adams-Bashforth de 2ª Ordem

O método de Adams-Bashforth é deduzido utilizando a interpolação de diferenças finitas. Para deduzirmos o método de 2ª ordem, considere que queremos avançar de t_n para t_{n+1}, dado que temos os valores de $y(t_n)$ e $y(t_{n-1})$, bem como as respectivas derivadas $f_n = y'(t_n)$ e $f_{n-1} = y'(t_{n-1})$. Para encontrar a fórmula de Adams-Bashforth, começamos pela expansão de Taylor da derivada de y no ponto t_{n+1}, que pode ser aproximada pela combinação linear das derivadas avaliadas nos pontos t_n e t_{n-1}. Partimos da seguinte ideia:

$$y'(t) = f(t, y(t))$$

que representa a taxa de variação de y no tempo. Queremos calcular $y(t_{n+1})$, o valor da função no próximo passo de tempo $t_{n+1} = t_n + \Delta t$. Expansão de Taylor nos dá a fórmula básica de avanço no tempo:

$$y_{n+1} = y_n + \Delta t \cdot y'(t_n)$$

Essa é a forma mais simples, o método de Euler, mas o método de Adams-Bashforth melhora essa aproximação ao usar também o valor da derivada no ponto anterior t_{n-1}. Utilizamos uma interpolação polinomial para combinar os valores anteriores de $f(t, y)$. No caso de 2ª ordem, a fórmula de Adams-Bashforth é derivada ao integrar o polinômio de Lagrange que interpola $f(t)$ sobre o intervalo $[t_{n-1}, t_n]$. Isso resulta na fórmula:

$$y_{n+1} = y_n + \frac{\Delta t}{2}(3f_n - f_{n-1})$$

Aqui, $f_n = f(t_n, y_n)$ e $f_{n-1} = f(t_{n-1}, y_{n-1})$. Essa fórmula é muito mais precisa do que o método de Euler, pois leva em consideração a variação de $f(t)$ entre dois pontos. O método de Adams-Bashforth de 2ª ordem (assim como outros métodos multistep) não é auto-inicializável, ou seja, ele não pode ser usado diretamente desde o primeiro passo de tempo. Isso ocorre porque ele depende de dois valores anteriores de y e de $f(t, y)$. Para y_1, o valor de $y(t_1)$, precisamos de um método de passo simples, como o método de Euler ou o método de Runge-Kutta, para fornecer o primeiro valor da solução. Ou seja, o primeiro passo de integração deve ser feito com outro método. No caso desta implementação, usaremos o método de Euler para calcular y_1, e, a partir daí, aplicamos a fórmula de Adams-Bashforth de 2ª ordem para os

próximos passos.

```
#include <stdio.h>

#define N 100   // Número de passos de tempo
#define DELTA_T 0.1  // Passo de tempo

// Função que representa a derivada dy/dt = -y + t^2 + 1
double f(double t, double y) {
    return -y + t * t + 1;
}

// Função para resolver a EDO usando o método de Adams-Bashforth de 2ª ordem
void adams_bashforth_2(double y0) {
    double y = y0;  // Valor inicial de y
    double t = 0.0; // Tempo inicial
    double y_prev;  // Valor anterior de y
    double f_prev, f_curr;  // Derivadas anteriores e atuais

    FILE *file = fopen("adams2_ordem.dat", "w");
    fprintf(file, "# Tempo\tValor de y\n");

    // Primeiro passo usando método de Euler para iniciar o cálculo
    f_curr = f(t, y);
    y_prev = y;
    t += DELTA_T;
    y = y_prev + DELTA_T * f_curr; // Método de Euler
    fprintf(file, "%lf\t%lf\n", t, y);

    // Aplicando Adams-Bashforth de 2ª ordem para os próximos passos
    for (int i = 1; i < N; i++) {
        t += DELTA_T;
        f_prev = f_curr;          // Atualiza f_{n-1}
        f_curr = f(t, y);         // Calcula f_n
        y = y + DELTA_T / 2 * (3 * f_curr - f_prev); // Adams-Bashforth 2ª ordem
        fprintf(file, "%lf\t%lf\n", t, y);
    }

    fclose(file);
}

int main() {
    double y_inicial = 0.5; // Condição inicial y(0) = 0.5
    adams_bashforth_2(y_inicial);

    return 0;
}
```

0.16 Método de Adams-Bashforth de 4ª Ordem

Agora, sem nos aprofundarmos em uma dedução completa, aplicaremos diretamente o método de Adams-Bashforth de 4ª ordem, uma técnica multistep

explícita amplamente utilizada para resolver equações diferenciais ordinárias (EDOs). Esse método utiliza uma combinação linear de quatro valores anteriores de $f(t, y)$ (a função que define a EDO) para calcular y_{n+1}, a solução no próximo passo de tempo. A fórmula é dada por:

$$y_{n+1} = y_n + \frac{\Delta t}{24} \left(55 f_n - 59 f_{n-1} + 37 f_{n-2} - 9 f_{n-3}\right),$$

onde: - y_n é o valor da solução no instante t_n, - $f_n = f(t_n, y_n)$ é o valor da função derivada em t_n, - Δt é o tamanho do passo de tempo, - $f_{n-1}, f_{n-2}, f_{n-3}$ são os valores de f calculados nos três passos de tempo anteriores.

Este método é mais preciso do que métodos de ordens inferiores, como o Adams-Bashforth de 2ª ordem, pois leva em consideração mais valores anteriores de $f(t, y)$, permitindo uma melhor estimativa de y_{n+1}. A precisão aumenta significativamente à medida que mais informações sobre o comportamento passado da solução são incorporadas na estimativa. No entanto, o método de Adams-Bashforth de 4ª ordem não é auto-inicializável. Isso significa que, para dar início ao processo, os três primeiros valores de y (ou seja, y_1, y_2 e y_3) precisam ser calculados com um método de passo simples, como o método de Euler ou o método de Runge-Kutta de 4ª ordem. Uma vez obtidos esses valores iniciais, o método de Adams-Bashforth pode ser aplicado para avançar a solução no tempo. Essa abordagem torna o método de Adams-Bashforth muito eficiente para problemas que exigem precisão em longas integrações temporais, pois a fórmula de quarta ordem equilibra bem a precisão com o esforço computacional.

```
#include <stdio.h>

#define N 100   // Número de passos de tempo
#define DELTA_T 0.1  // Passo de tempo

// Função que representa a derivada dy/dt = -y + t^2 + 1
double f(double t, double y) {
    return -y + t * t + 1;
}

// Função para resolver a EDO usando o método de Adams-Bashforth de 4ª ordem
void adams_bashforth_4(double y0) {
    double y = y0;  // Valor inicial de y
    double t = 0.0; // Tempo inicial
    double y_prev[3];  // Valores anteriores de y
    double f_prev[4];  // Derivadas anteriores e atual

    FILE *file = fopen("adams4_ordem.dat", "w");
```

```
    fprintf(file, "# Tempo\tValor de y\n");

    // Primeiros três passos usando método de Euler para iniciar o cálculo
    f_prev[0] = f(t, y); // f_0
    y_prev[0] = y;
    t += DELTA_T;
    y = y_prev[0] + DELTA_T * f_prev[0]; // Euler
    f_prev[1] = f(t, y); // f_1
    y_prev[1] = y;
    t += DELTA_T;
    y = y_prev[1] + DELTA_T * f_prev[1]; // Euler
    f_prev[2] = f(t, y); // f_2
    y_prev[2] = y;
    t += DELTA_T;
    y = y_prev[2] + DELTA_T * f_prev[2]; // Euler
    f_prev[3] = f(t, y); // f_3
    fprintf(file, "%lf\t%lf\n", t, y);

    // Aplicando Adams-Bashforth de 4ª ordem para os próximos passos
  for (int i = 3; i < N; i++) {
     t += DELTA_T;
     for (int j = 3; j > 0; j--) {
        f_prev[j] = f_prev[j - 1];  // Atualiza f_{n-1}, f_{n-2}, f_{n-3}
      }
      f_prev[0] = f(t, y);            // Calcula f_n
      y = y + DELTA_T / 24 * (55 * f_prev[0] - 59 * f_prev[1] + 37 * f_prev[2] - 9 * f_prev[3]);
      fprintf(file, "%lf\t%lf\n", t, y);
  }

    fclose(file);
}

int main() {
    double y_inicial = 0.5; // Condição inicial y(0) = 0.5
    adams_bashforth_4(y_inicial);

    return 0;
}
```

0.17 Solução para o Modelo de Anderson com termo diagonal não-linear usando Adams-Bashforth de 4ª Ordem

Nesta seção, resolvemos a equação diferencial não linear dada por:

$$i\frac{d}{dt}f_n(t) = -J\left(f_{n+1}(t) + f_{n-1}(t)\right) + \epsilon_n f_n(t),$$

onde $\epsilon_n = A|f_n|^2$. Para isso vamos aplicar o método de Adams-Bashforth de 4ª ordem. Inicialmente, utilizamos o método Runge-Kutta de 4ª ordem (RK4) para calcular os valores iniciais. Para cada equação neste sistema o

método RK4 será aplicado como segue:

$$
\begin{aligned}
k_1^n &= \Delta t \cdot \frac{-J}{i}\left(f_{n+1}(t) + f_{n-1}(t)\right) + \frac{\epsilon_n}{i} f_n(t) \\
k_2^n &= \Delta t \cdot \frac{-J}{i}\left(f_{n+1}(t + \frac{\Delta t}{2}) + f_{n-1}(t + \frac{\Delta t}{2})\right) + \frac{\epsilon_n}{i}\left(f_n(t) + \frac{k_1}{2}\right) \\
k_3^n &= \Delta t \cdot \frac{-J}{i}\left(f_{n+1}(t + \frac{\Delta t}{2}) + f_{n-1}(t + \frac{\Delta t}{2})\right) + \frac{\epsilon_n}{i}\left(f_n(t) + \frac{k_2}{2}\right) \\
k_4^n &= \Delta t \cdot \frac{-J}{i}\left(f_{n+1}(t + \Delta t) + f_{n-1}(t + \Delta t)\right) + \frac{\epsilon_n}{i}\left(f_n(t) + k_3\right)
\end{aligned}
$$

A solução no tempo $t + \Delta t$ é dada por :

$$f_n(t + \Delta t) = f_n(t) + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4)$$

Para resolver uma equação diferencial utilizando o método de Adams-Bashforth de 4ª ordem, é necessário ter os valores da função nos quatro primeiros instantes de tempo. Como o método de Adams-Bashforth é um método de passo múltiplo, ele depende de valores prévios para calcular o próximo ponto da integração. Para obter esses valores iniciais, utilizamos o método de Runge-Kutta de 4ª ordem (RK4), que é um método de passo único e fornece uma estimativa precisa para os primeiros quatro pontos da solução. Assim, começamos aplicando o RK4 para calcular as soluções nos primeiros quatro instantes: $t = \Delta t$, $t = 2\Delta t$, $t = 3\Delta t$ e $t = 4\Delta t$. Esses valores iniciais são armazenados e, a partir daí, passamos a utilizar o método de Adams-Bashforth de 4ª ordem para os passos subsequentes. O método de Adams-Bashforth usa as soluções prévias para prever o valor da função no próximo instante de tempo, permitindo que o processo de integração prossiga de forma eficiente. Esse esquema combinado — RK4 para os primeiros passos e Adams-Bashforth para os seguintes — é frequentemente utilizado porque o RK4 fornece uma base inicial precisa, enquanto o Adams-Bashforth oferece uma solução eficiente para passos subsequentes, aproveitando a informação dos pontos anteriores. O código a seguir implementa esse algoritmo, começando com RK4 e continuando com Adams-Bashforth de 4ª ordem.

```
#include <stdio.h>
#include <stdlib.h>
```

```
#include <math.h>
#include <complex.h>
#include <time.h>

#define NMAX1   10000 // tamanho máximo da cadeia
#define Npassos 20000 // número de passos
#define J 1.0     // Constante de acoplamento
#define A 1.0     // Amplitude do termo estocástico
#define DT 0.001   // Passo de tempo
#define EPSILON 1e-6 // Tolerância para verificar o retorno ao meio
double complex f[NMAX1+1];
double complex k1[NMAX1+1],k2[NMAX1+1];
double complex  df[NMAX1+1],k3[NMAX1+1],k4[NMAX1+1],f_temp[NMAX1+1];
double complex derv[NMAX1+1][5];
double t,prob_retorno,norma,epsilon;
int N,i,j,middle,n,i100;

// Função que calcula a derivada de f_n(t) para o método RK4
void derivada() {
    for (i = 0; i<=N-1; i++) { // Evita os limites para não acessar posições inválidas
        epsilon=A*pow(cabs(f_temp[i]),2.);
       if (i==0) df[i] = -I * (-J * (f_temp[i+1]) + epsilon*f_temp[i]);
       if (i>0) df[i] = -I * (-J * (f_temp[i+1]+f_temp[i-1])+epsilon*f_temp[i]);
    }
}

// Função principal
int main() {
    // Definir variáveis
    N=100;
    t=0.;
    middle=N/2;
    FILE *outfile;

    // Inicializar gerador de números aleatórios
   // srand(time(NULL));

    // Inicializar condições iniciais
    for (int n = 0; n <= N; n++) {
        f[n] = 0.0 + 0.0*I; // Coloca o elétron inicialmente no meio
        f_temp[n]=0.0 + 0.0*I;
    }
    f[middle]=f_temp[middle]= 1.0 + 0.0*I; // Colocar um valor inicial no meio

       derivada();
    // Abrir arquivo para salvar os resultados
    outfile = fopen("probabilidade_retorno.dat", "w");

    // Calculando os 4 primeiros passos com RK4
    t=0.;
    for (n = 0; n < 4; n++) {
     t=t+DT;
        // Calcular k1, k2, k3, k4




        for (i = 0; i <= N-1; i++) {
```

```
           k1[i]=df[i];
            f_temp[i]=f[i]+0.5*DT*k1[i];

        }


            derivada();

        for (i = 0; i <= N-1; i++) {

            k2[i]=df[i];
            f_temp[i]=f[i]+0.5*DT*k2[i];

        }

          derivada();
        for (i = 0; i <= N-1; i++) {
            k3[i]=df[i];
            f_temp[i] =f[i]+ DT*k3[i];

        }
             derivada();
           norma=0.;
        for (i = 0; i <= N-1; i++){

            k4[i]=df[i];


           f_temp[i]=f[i]+(DT/ 6.0)*(k1[i] + 2*k2[i] + 2*k3[i] + k4[i]);
           f[i]=f_temp[i];
           norma=norma+pow(cabs(f[i]),2.);
        }
 printf( "%f %f\n", t,norma);
             derivada();

                 for (i = 0; i <= N-1; i++){
                  derv[i][n]=df[i];

                 }

    }


 i100=0;
    // Usar Adams-Bashforth de 4ª ordem para passos subsequentes
  for (n = 4; n < Npassos; n++) {
        // Previsão de f_n usando Adams-Bashforth
i100=i100+1;
         t=t+DT;
   for (i = 0;i <= N-1; i++){


      f_temp[i] =f[i]+(DT/24.0)*(55.*derv[i][n-1]-59.*derv[i][n-2]+37.*derv[i][n-3]-9.*derv[i][n-4]);

    }

        derivada();
        norma=0.;
         for (i = 0; i<=N-1; i++){
```

```
          derv[i][n-4]=derv[i][n-3];
          derv[i][n-3]=derv[i][n-2];
          derv[i][n-2]=derv[i][n-1];
          derv[i][n-1]=df[i];
          f[i]=f_temp[i];
        norma=norma+pow(cabs(f[i]),2.);
          }



          // Calcular a probabilidade de retorno ao meio
          prob_retorno = pow(cabs(f[middle]),2.);
        if (i100>1000){

    fprintf(outfile, "%f %f %f\n", t, prob_retorno,norma);

    i100=0;
        }

    }

    fclose(outfile);
    return 0;
}
```

O método de Adams é uma família de métodos numéricos utilizados para a resolução de equações diferenciais ordinárias (EDOs). Ele se baseia na aproximação da solução de uma EDO por meio de uma série de passos sucessivos, utilizando valores anteriores da função e suas derivadas para melhorar a precisão da solução. Este método é geralmente dividido em duas categorias: o *método preditor-corretor* e o *método de Adams-Bashforth-Moulton*, que combina uma etapa preditiva (explícita) com uma etapa corretiva (implícita).

No entanto, embora o método de Adams seja eficiente para uma ampla gama de EDOs, ele pode não ser tão preciso em alguns casos específicos, como o modelo de Anderson em sistemas desordenados. No contexto desse modelo, as flutuações caóticas da desordem e a presença de interações locais podem exigir uma abordagem mais robusta para manter a precisão. A introdução de correções é uma prática comum nesses casos.

No caso apresentado no último exemplo mesmo, o modelo de Anderson, que descreve a localização de elétrons em sistemas desordenados, existem desafios adicionais na sua resolução numérica devido à natureza complexa da desordem e das interações no sistema. Nesses casos, o método de Adams pode falhar em fornecer resultados precisos se utilizado de forma direta, especialmente para sistemas de grande escala e longos tempos de integração. Para superar esses problemas, pode ser necessário aplicar fórmulas de correção ao método de Adams, especialmente para equações que envolvem acoplamen-

tos desordenados ou não-lineares. A seguir, apresentamos as fórmulas de correção para o método de Adams de quarta ordem. Essas fórmulas combinam o método de predição com um passo corretivo que refina a solução obtida. Após a predição obtida pelo método de Adams-Bashforth , o método de Adams-Moulton (corretor) de 4ª ordem é aplicado para refinar a estimativa:

$$y_{n+1} = y_n + \frac{h}{24}\left(9f_{n+1}^{(p)} + 19f_n - 5f_{n-1} + f_{n-2}\right), \tag{15}$$

onde y_{n+1} é a estimativa corrigida, obtida pela utilização do valor predito $f_{n+1}^{(p)} = f(t_{n+1}, y_{n+1}^{(p)})$. Esse esquema preditor-corretor permite uma maior precisão na solução de EDOs, especialmente em sistemas mais complexos, como o modelo de Anderson. No entanto, para equações onde a desordem é dominante, pode ser necessário ajustar os parâmetros do método para garantir a convergência e a precisão dos resultados. O método de Adams é uma ferramenta poderosa para a solução de EDOs, mas, em sistemas complexos como o modelo de Anderson, pode ser necessário aplicar correções para garantir a precisão. A utilização do esquema preditor-corretor com as fórmulas de 4ª ordem fornecidas aqui é uma estratégia eficaz para melhorar os resultados numéricos nesses casos. O código a seguir mostra o esquema preditor-corretor implementado para a equação não-linear.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <complex.h>
#include <time.h>

#define NMAX1   10000 // tamanho máximo da cadeia
#define Npassos 20000 // número de passos
#define J 1.0     // Constante de acoplamento
#define A 1.0     // Amplitude do termo estocástico
#define DT 0.001   // Passo de tempo
#define EPSILON 1e-6 // Tolerância para verificar o retorno ao meio
double complex f[NMAX1+1];
double complex k1[NMAX1+1],k2[NMAX1+1];
double complex  df[NMAX1+1],k3[NMAX1+1],k4[NMAX1+1],f_temp[NMAX1+1];
double complex derv[NMAX1+1][5];
double t,prob_retorno,norma,epsilon;
int N,i,j,middle,n,i100;

// Função que calcula a derivada de f_n(t) para o método RK4
void derivada() {
    for (i = 0; i<=N-1; i++) { // Evita os limites para não acessar posições inválidas
        epsilon=A*pow(cabs(f_temp[i]),2.);
```

```
       if (i==0) df[i] = -I * (-J * (f_temp[i+1]) + epsilon*f_temp[i]);
       if (i>0) df[i] = -I * (-J * (f_temp[i+1]+f_temp[i-1])+epsilon*f_temp[i]);
    }
}

// Função principal
int main() {
    // Definir variáveis
    N=100;
    t=0.;
    middle=N/2;
    FILE *outfile;

    // Inicializar gerador de números aleatórios
   // srand(time(NULL));

    // Inicializar condições iniciais
    for (int n = 0; n <= N; n++) {
        f[n] = 0.0 + 0.0*I; // Coloca o elétron inicialmente no meio
        f_temp[n]=0.0 + 0.0*I;
    }
    f[middle]=f_temp[middle]= 1.0 + 0.0*I; // Colocar um valor inicial no meio

       derivada();
    // Abrir arquivo para salvar os resultados
    outfile = fopen("probabilidade_retorno.dat", "w");

    // Calculando os 4 primeiros passos com RK4
    t=0.;
    for (n = 0; n < 4; n++) {
     t=t+DT;
        // Calcular k1, k2, k3, k4




        for (i = 0; i <= N-1; i++) {

           k1[i]=df[i];
            f_temp[i]=f[i]+0.5*DT*k1[i];

        }


            derivada();

        for (i = 0; i <= N-1; i++) {

            k2[i]=df[i];
            f_temp[i]=f[i]+0.5*DT*k2[i];

        }

          derivada();
        for (i = 0; i <= N-1; i++) {
            k3[i]=df[i];
            f_temp[i] =f[i]+ DT*k3[i];

        }
```

```
            derivada();
           norma=0.;
        for (i = 0; i <= N-1; i++){

            k4[i]=df[i];


           f_temp[i]=f[i]+(DT/ 6.0)*(k1[i] + 2*k2[i] + 2*k3[i] + k4[i]);
           f[i]=f_temp[i];
           norma=norma+pow(cabs(f[i]),2.);
        }
 printf( "%f %f\n", t,norma);
            derivada();

                for (i = 0; i <= N-1; i++){
                 derv[i][n]=df[i];

                }

    }


 i100=0;
    // Usar Adams-Bashforth de 4ª ordem para passos subsequentes
  for (n = 4; n < Npassos; n++) {

i100=i100+1;
         t=t+DT;

         // // Previsão de f_n usando Adams-Bashforth
   for (i = 0;i <= N-1; i++){


      f_temp[i] =f[i]+(DT/24.0)*(55.*derv[i][n-1]-59.*derv[i][n-2]+37.*derv[i][n-3]-9.*derv[i][n-4]);

    }

        derivada();



        // correção usando  Adams-Moulton

        for (i = 0;i <= N-1; i++){
         f_temp[i]=f[i]+(DT/24.)*(9.*df[i]+19.*derv[i][n-1]-5.*derv[i][n-2]+derv[i][n-3]);
        }

         derivada();

        norma=0.;
         for (i = 0; i<=N-1; i++){
         derv[i][n-4]=derv[i][n-3];
         derv[i][n-3]=derv[i][n-2];
         derv[i][n-2]=derv[i][n-1];
         derv[i][n-1]=df[i];
         f[i]=f_temp[i];
       norma=norma+pow(cabs(f[i]),2.);
         }
```

```
        // Calcular a probabilidade de retorno ao meio
        prob_retorno = pow(cabs(f[middle]),2.);
      if (i100>1000){

    fprintf(outfile, "%f %f %f\n", t, prob_retorno,norma);

    i100=0;
        }

    }

    fclose(outfile);
    return 0;
}
```

Se faz importante salientar que existem métodos de Adams de alta ordem (como os de 6ª, 8ª, 10ª ordem, etc. A principal vantagem dos métodos de Adams de alta ordem é a precisão elevada que podem atingir com relativamente poucos passos, tornando-os ideais para problemas onde é importante minimizar o erro de truncamento. À medida que a ordem do método aumenta, a aproximação numérica da solução converge mais rapidamente para a solução exata, o que significa que, para o mesmo intervalo de tempo e passos de integração, métodos de ordem mais alta podem oferecer soluções mais precisas do que aqueles de ordem inferior. Além disso, para sistemas onde a solução varia suavemente ao longo do tempo, os métodos de Adams de alta ordem permitem resolver problemas de longo prazo sem a necessidade de passos de tempo muito pequenos, o que reduz o número total de iterações e, consequentemente, o tempo computacional. Apesar das vantagens em termos de precisão, os métodos de Adams de alta ordem têm algumas desvantagens. Uma delas é a necessidade de mais valores iniciais para iniciar a integração. Por exemplo, para usar um método de 6ª ordem, são necessários os valores da função e suas derivadas em seis pontos anteriores; para um método de 10ª ordem, são necessários dez pontos, e assim por diante. Isso significa que, antes de aplicar um método de alta ordem, é necessário calcular esses valores iniciais com um método de passo único, como o Runge-Kutta, o que pode aumentar o custo inicial da simulação. Outro problema associado aos métodos de alta ordem é a sua instabilidade em sistemas com alta sensibilidade a condições iniciais ou onde há oscilações rápidas na solução. Métodos de ordem muito alta podem ser menos estáveis em sistemas rígidos (*stiff systems*), onde há escalas de tempo muito distintas entre diferentes componentes da equação, exigindo o uso de estratégias adaptativas ou mesmo de métodos especializados para garantir a estabilidade. Os métodos de Adams, sendo métodos de passo múltiplo, necessitam de um número correspondente de pontos iniciais para iniciar o processo de integração. Para métodos de 6ª ordem, por exemplo, são necessários seis

valores iniciais da função; para métodos de 10ª ordem, são necessários dez. Isso significa que, antes de aplicar um método de Adams de alta ordem, devemos usar um método de passo único, como o Runge-Kutta de 4ª ordem (RK4) ou um método de ordem mais alta, para calcular os primeiros pontos da solução. Esse esquema híbrido, onde métodos de passo único como RK4 são usados para iniciar a integração, e os métodos de Adams assumem a partir de então, é comum. O RK4 é utilizado para calcular as soluções nos primeiros instantes de tempo, e essas soluções são então utilizadas para alimentar o método de Adams nas iterações seguintes, proporcionando uma combinação eficiente de precisão inicial e alta eficiência nos passos subsequentes. Os métodos de Adams de alta ordem são ferramentas poderosas para a solução de equações diferenciais, especialmente em problemas onde a solução se comporta de maneira suave e a precisão é essencial. No entanto, sua aplicação deve ser feita com cautela, considerando as necessidades de valores iniciais e os potenciais problemas de instabilidade. A escolha de um método de alta ordem depende tanto das características da equação diferencial quanto dos requisitos de precisão e eficiência computacional. Métodos híbridos, que começam com Runge-Kutta e depois utilizam Adams, são amplamente utilizados para balancear essas necessidades.

0.18 Solução Numérica do Sistema Massa-Mola com Resistência do Ar Utilizando Diferenças Finitas de 2ª Ordem

Neste capítulo, abordaremos a solução numérica de uma equação diferencial que descreve o comportamento de um sistema massa-mola com resistência do ar. A equação diferencial é dada por:

$$m\frac{d^2x}{dt^2} = -kx - \beta v$$

onde m é a massa, k é a constante da mola, v é a velocidade, e β é o coeficiente de resistência do ar. Dividindo ambos os lados da equação por m, temos:

$$\frac{d^2x}{dt^2} + \frac{\beta}{m}\frac{dx}{dt} + \frac{k}{m}x = 0$$

Nosso objetivo é resolver essa equação numericamente utilizando o método de diferenças finitas de 2ª ordem. O método de diferenças finitas é uma técnica numérica para aproximar as derivadas de funções. No caso de derivadas de segunda ordem, podemos utilizar a seguinte aproximação para a segunda derivada:

$$\frac{d^2x}{dt^2} \approx \frac{x_{n+1} - 2x_n + x_{n-1}}{\Delta t^2}$$

Além disso, a derivada de primeira ordem $\frac{dx}{dt}$ pode ser aproximada por:

$$\frac{dx}{dt} \approx \frac{x_{n+1} - x_{n-1}}{2\Delta t}$$

Substituindo essas aproximações na equação diferencial, obtemos:

$$\frac{x_{n+1} - 2x_n + x_{n-1}}{\Delta t^2} + \frac{\beta}{m}\frac{x_{n+1} - x_{n-1}}{2\Delta t} + \frac{k}{m}x_n = 0$$

Multiplicando toda a equação por Δt^2, reescrevemos como:

$$x_{n+1} - 2x_n + x_{n-1} + \frac{\beta\Delta t}{2m}(x_{n+1} - x_{n-1}) + \frac{k\Delta t^2}{m}x_n = 0$$

Isolando x_{n+1}:

$$x_{n+1} = \frac{2x_n - x_{n-1} + \frac{\beta\Delta t}{2m}x_{n-1} - \frac{k\Delta t^2}{m}x_n}{1 + \frac{\beta\Delta t}{2m}}$$

Essa é a fórmula que utilizaremos para calcular os valores sucessivos de x. Como estamos lidando com uma equação diferencial de segunda ordem, precisamos especificar tanto a posição inicial $x(0)$ quanto a velocidade inicial

$v(0)$. O primeiro passo de integração, x_1, pode ser calculado usando o método de Euler:

$$x_1 = x_0 + v_0 \Delta t$$

A seguir, apresentamos a implementação do método de diferenças finitas de 2ª ordem para resolver a equação diferencial do sistema massa-mola com resistência do ar.

```
#include <stdio.h>
#include <math.h>

#define N 1000      // Número de passos de tempo
#define DELTA_T 0.01 // Passo de tempo
#define M 1.0       // Massa
#define K 1.0       // Constante da mola
#define BETA 0.2    // Coeficiente de resistência do ar

// Função para resolver a EDO da massa-mola usando diferenças finitas de 2ª ordem
void diferencas_finitas_2a_ordem(double x0, double v0) {
    double x[N];  // Posições
    double t = 0.0; // Tempo inicial

    FILE *file = fopen("massa_mola_dif_fin_2a_ordem.dat", "w");
    fprintf(file, "# Tempo\tPosição\n");

    // Condições iniciais
    x[0] = x0;
    fprintf(file, "%lf\t%lf\n", t, x[0]);

    // Calcula x_1 usando o método de Euler para iniciar
    x[1] = x[0] + DELTA_T * v0;
    t += DELTA_T;
    fprintf(file, "%lf\t%lf\n", t, x[1]);

    // Aplicando diferenças finitas de 2ª ordem para os próximos passos
    for (int i = 1; i < N - 1; i++) {
        t += DELTA_T;
        x[i + 1] = (2 * x[i] - x[i - 1] + (BETA * DELTA_T / (2 * M)) * x[i - 1] - (K * DELTA_T * DELTA
                   / (1 + (BETA * DELTA_T / (2 * M)));
        fprintf(file, "%lf\t%lf\n", t, x[i + 1]);
    }

    fclose(file);
}

int main() {
    double x_inicial = 1.0;  // Posição inicial x(0) = 1
    double v_inicial = 0.0;  // Velocidade inicial v(0) = 0

    diferencas_finitas_2a_ordem(x_inicial, v_inicial);
```

```
    return 0;
}
```

No código apresentado:

- A função `diferencas_finitas_2a_ordem()` implementa a solução da equação diferencial usando o método de diferenças finitas de 2ª ordem.
- O vetor x armazena as posições calculadas em cada passo de tempo.
- As condições iniciais x_0 e v_0 são passadas como parâmetros para a função. O primeiro valor de x_1 é calculado utilizando o método de Euler.
- Para os demais valores de x, a fórmula de diferenças finitas de 2ª ordem é utilizada iterativamente.
- Os resultados são armazenados no arquivo `massa_mola_dif_fin_2a_ordem.dat`, contendo os valores da posição x para cada instante de tempo t.

O método de diferenças finitas de 2ª ordem é uma técnica numérica eficiente para resolver equações diferenciais de segunda ordem, como a equação do sistema massa-mola com resistência do ar. A precisão desse método é superior à do método de Euler, especialmente para passos de tempo menores. O código em C apresentado pode ser utilizado para explorar diferentes condições de massa, constante da mola, coeficiente de resistência do ar e outros parâmetros para estudar o comportamento oscilatório de sistemas físicos amortecidos.

0.19 Solução Numérica para o Sistema Massa-Mola com Força Estocástica: método de Euler-Maruyama

Neste capítulo, abordaremos a solução numérica da equação diferencial para um sistema massa-mola com uma força estocástica. Vamos consider a equação diferencial:

$$m\frac{d^2x}{dt^2} = -kx + f(t), \tag{16}$$

onde $f(t)$ é uma função estocástica dada por:

$$f(t) = Ar(t), \tag{17}$$

e $r(t)$ é uma variável aleatória gaussiana com média zero e variância 1. Para resolver essa equação com uma força estocástica, utilizaremos o método de Euler-Maruyama, que é uma extensão do método de Euler para equações diferenciais estocásticas. O método de Euler-Maruyama é uma abordagem para resolver equações diferenciais estocásticas da forma:

$$\frac{dX(t)}{dt} = a(X(t), t) + b(X(t), t)r(t), \tag{18}$$

onde $a(X(t), t)$ é o termo determinístico e $b(X(t), t)r(t)$ é o termo estocástico, com $r(t)$ representando o termo de ruído branco gaussiano. Para o nosso problema, a equação pode ser reescrita como um sistema de equações diferenciais de primeira ordem:

$$\frac{dx(t)}{dt} = v(t), \tag{19}$$

$$\frac{dv(t)}{dt} = -\frac{k}{m}x(t) + \frac{f(t)}{m}. \tag{20}$$

O método de Euler-Maruyama para resolver esse sistema é dado por:

$$x_{i+1} = x_i + v_i \Delta t, \tag{21}$$

$$v_{i+1} = v_i + \left(-\frac{k}{m}x_i\right)\Delta t + \frac{Ar_i}{m}\sqrt{\Delta t}, \tag{22}$$

A seguir, apresentamos um programa em C que usa o método de Euler-Maruyama para resolver a equação diferencial com uma força estocástica. O programa gera variáveis aleatórias gaussianas para simular o termo estocástico e atualiza a posição e a velocidade da massa.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#define M 1.0          // Massa
#define K 1.0          // Constante da mola
#define A 0.001          // Amplitude da força estocástica
#define DT 0.01        // Passo de tempo
#define N 1000         // Número de passos

// Função para gerar variáveis aleatórias gaussianas
double gaussian_random() {
    double u1 = ((double)rand() / RAND_MAX);
    double u2 = ((double)rand() / RAND_MAX);
    return sqrt(-2 * log(u1)) * cos(2 * M_PI * u2);
}

int main() {
    double x = 0.01;   // Posição inicial
    double v = 0.0;   // Velocidade inicial
    double f;

    int i;

    // Gerar e resolver
    FILE *file = fopen("solucao.dat", "w");
    for (i = 0; i < N; i++) {
        f = A * gaussian_random();
        x = x + v * DT;
        v = v + (- (K / M) * x ) * DT + (f / M)*sqrt(DT);
        fprintf(file, "%lf %lf %lf\n", ((double)i)*DT, x, v);
    }
    fclose(file);

    return 0;
}
```

Nesta seção, apresentamos a solução numérica da equação diferencial para um sistema massa-mola com uma força estocástica utilizando o método de Euler-Maruyama. Este método é apropriado para lidar com termos estocásticos e inclui o termo de correção $\sqrt{\Delta t}$ para garantir a precisão das simulações. O código fornecido em C ilustra a aplicação prática do método, gerando uma série de valores para a posição e a velocidade da massa. Este método é útil para analisar sistemas dinâmicos com características estocásticas e pode ser aplicado em diversas áreas da física e engenharia.

0.20 Solução Numérica para o Sistema Massa-Mola com Força Estocástica : Método de Runge Kutta adaptado

Esta seção descreve a solução numérica da mesma equação diferencial com força estocástica sendo que agora vamos usar um método de Runge-Kutta de 2^a ordem adaptado para este tipo de problema. A equação diferencial estocástica é dada por:

$$\frac{d^2x}{dt^2} = -kx + f(t)$$

onde $f(t) = A \cdot r(t)$, com $r(t)$ sendo uma função estocástica gerada a partir de números gaussianos com média zero e variância 1. A equação será resolvida utilizando o método estocástico de Runge-Kutta de $2^{\underline{a}}$ ordem (SRK2). Primeiro, reescrevemos a equação de segunda ordem como um sistema de duas equações diferenciais de primeira ordem. Definimos:

$$v = \frac{dx}{dt}$$

Assim, temos o sistema:

$$\frac{dx}{dt} = v$$

$$\frac{dv}{dt} = -kx + A \cdot r(t)$$

onde $r(t) \sim N(0, 1)$, ou seja, números aleatórios gaussianos com média zero e variância 1. O método de Runge-Kutta estocástico de $2^{\underline{a}}$ ordem é descrito da seguinte forma. Dado um intervalo de tempo Δt, queremos calcular $x(t)$ e $v(t)$ em passos discretos de tempo. Para cada passo $t_n \to t_{n+1} = t_n + \Delta t$, o esquema é dado por:

$$
\begin{aligned}
k_1 &= v_n \\
l_1 &= -kx_n + A \cdot r_n \\
k_2 &= v_n + l_1 \frac{\Delta t}{2} \\
l_2 &= -k\left(x_n + k_1 \frac{\Delta t}{2}\right) + A \cdot r_{n+1}
\end{aligned}
$$

Então, as atualizações para x e v são dadas por:

$$
\begin{aligned}
x_{n+1} &= x_n + k_1 \Delta t + \frac{1}{2} \Delta W_n \\
v_{n+1} &= v_n + l_1 \Delta t + \frac{1}{2} \Delta W_n
\end{aligned}
$$

Aqui, ΔW_n é o incremento de Wiener, calculado como:

$$
\Delta W_n = \sqrt{\Delta t} \cdot r_n
$$

O código a seguir implementa o algoritmo SRK2 descrito anteriormente.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>

// Definir constantes globais
#define N 1000   // Número de passos
#define A 1.0    // Amplitude do termo estocástico
#define k 1.0    // Constante da equação
#define DT 0.01  // Passo de tempo

// Função para gerar números aleatórios gaussianos
double gaussrand() {
    static int hasSpare = 0;
    static double spare;
    if(hasSpare) {
        hasSpare = 0;
        return spare;
```

```
    }
    hasSpare = 1;
    double u, v, s;
    do {
        u = (rand() / ((double) RAND_MAX)) * 2.0 - 1.0;
        v = (rand() / ((double) RAND_MAX)) * 2.0 - 1.0;
        s = u * u + v * v;
    } while( s >= 1.0 || s == 0.0 );
    s = sqrt(-2.0 * log(s) / s);
    spare = v * s;
    return u * s;
}

int main() {
    // Definir variáveis
    double x = 1.0, v = 0.0;  // Condições iniciais
    double t;
    double k1, l1, k2, l2;
    double r_n, r_n1;
    FILE *outfile;

    // Inicializar gerador de números aleatórios
    srand(time(NULL));

    // Abrir arquivo para salvar os resultados
    outfile = fopen("resultados.dat", "w");
        t=DT;
    // Loop principal
    for (int n = 1; n < N; n++) {
        // Gerar números aleatórios gaussianos para r_n e r_n1
        r_n = gaussrand();
        r_n1 = gaussrand();

        // Calcular os k's e l's
        k1 = v;
        l1 = -k * x + A * r_n;
        k2 = v + l1 * (DT / 2.0);
        l2 = -k * (x + k1 * (DT / 2.0)) + A * r_n1;

        // Atualizar x e v
        x = x + k1 * DT + 0.5 * sqrt(DT) * r_n;
        v = v + l1 * DT + 0.5 * sqrt(DT) * r_n;

        // Salvar os valores no arquivo
        fprintf(outfile, "%f %f %f\n",t,x, v);

        // Atualizar o tempo
        t += DT;
    }

    // Fechar o arquivo
    fclose(outfile);

    return 0;
}
```

O método estocástico de Runge-Kutta de 2ª ordem (SRK2) foi implementado para resolver a equação diferencial estocástica de segunda ordem

$\frac{d^2x}{dt^2} = -kx + f(t)$, onde $f(t)$ é uma função aleatória gaussiana. O código em C foi apresentado e realiza a simulação numérica, salvando os resultados em um arquivo.

0.21 Método de Verlet Velocity

O método de Verlet Velocity é uma abordagem numérica para resolver equações diferenciais como aquelas encontradas em sistemas de partículas e simulações moleculares. Para ilustrar este método, consideraremos o problema clássico da massa-mola, descrito pela equação diferencial:

$$\frac{d^2x}{dt^2} = -kx, \tag{23}$$

onde $x(t)$ é a posição da massa em função do tempo t e k é a constante da mola. Para resolver essa equação usando o método de Verlet Velocity, seguiremos os seguintes passos: Discretizamos o tempo em intervalos Δt, onde o tempo t_n é dado por $t_n = n\Delta t$. A posição x_n e a velocidade v_n da massa no instante t_n são calculadas a partir de:

$$x_{n+1} = x_n + v_n\Delta t + \frac{1}{2}a_n(\Delta t)^2, \tag{24}$$

$$v_{n+1/2} = v_n + \frac{1}{2}(a_n + a_{n+1})\Delta t, \tag{25}$$

onde a_n é a aceleração da massa no instante t_n, dada por:

$$a_n = -kx_n. \tag{26}$$

O método de Verlet Velocity pode ser descrito pelo seguinte algoritmo:

1. **Inicialização:** Defina as condições iniciais para a posição x_0, a velocidade v_0 e o passo de tempo Δt. Calcule a aceleração inicial $a_0 = -kx_0$.
2. **Loop de Iteração:**

(a) Calcule a nova posição usando:

$$x_{n+1} = x_n + v_n \Delta t + \frac{1}{2} a_n (\Delta t)^2. \tag{27}$$

(b) Calcule a nova aceleração $a_{n+1} = -k x_{n+1}$.

(c) Atualize a velocidade:

$$v_{n+1/2} = v_n + \frac{1}{2}(a_n + a_{n+1})\Delta t. \tag{28}$$

(d) Avance para o próximo passo de tempo $n \to n + 1$.

3. **Repetição:** Repita o loop de iteração até que o tempo desejado seja alcançado.

Consideramos uma massa m conectada a uma mola com constante k. Suponha que a massa está inicialmente em repouso com uma posição inicial $x_0 = 1$ e a constante da mola $k = 1$. O passo de tempo é escolhido como $\Delta t = 0.1$. Podemos implementar o método de Verlet Velocity com os seguintes valores iniciais:

- $x_0 = 1$
- $v_0 = 0$
- $a_0 = -k x_0 = -1$
- $\Delta t = 0.1$

A implementação do algoritmo em c é a seguinte:

```
#include <stdio.h>
#include <math.h>

#define K 1.0      // Constante da mola
#define DT 0.1     // Passo de tempo
#define N 1000     // Número de passos de tempo

int main() {
    // Variáveis iniciais
    double x0 = 1.0;          // Posição inicial
    double v0 = 0.0;          // Velocidade inicial
    double a0 = -K * x0;      // Aceleração inicial
```

```
    double x = x0;            // Posição no tempo atual
    double v = v0;            // Velocidade no tempo atual
    double a = a0;            // Aceleração no tempo atual
    double x_new;             // Nova posição
    double a_new;             // Nova aceleração

    // Abre o arquivo para escrever os resultados
    FILE *file = fopen("verlet_velocity.dat", "w");
    if (file == NULL) {
        printf("Erro ao abrir o arquivo.\n");
        return 1;
    }


    // Loop de iteração
    for (int i = 0; i < N; i++) {
        // Calcula a nova posição
        x_new = x + v * DT + 0.5 * a * DT * DT;

        // Calcula a nova aceleração
        a_new = -K * x_new;

        // Atualiza a velocidade
        v += 0.5 * (a + a_new) * DT;

        // Avança para o próximo passo de tempo
        x = x_new;
        a = a_new;

        // Escreve os resultados no arquivo
        fprintf(file, "%f\t%f\t%f\n", (i + 1) * DT, x, v);
    }

    // Fecha o arquivo
    fclose(file);

    return 0;
}
```

O método de Verlet Velocity é eficaz para a simulação de sistemas dinâmicos onde as forças podem ser calculadas a partir da posição. A precisão do método depende do tamanho do passo de tempo Δt. Passos menores geralmente resultam em uma maior precisão, mas aumentam o custo computacional. Este método é particularmente útil para simulações de partículas e sistemas físicos onde a conservação da energia é importante.

0.22 Simulação da Dinâmica de uma Partícula no Potencial de Morse

Nesta seção apresentamos a simulação do movimento de uma partícula sujeita ao potencial de Morse, que é frequentemente usado para descrever a

energia potencial de moléculas diatômicas. A equação de movimento da partícula é resolvida numericamente usando o método de Verlet Velocity. O potencial de Morse é dado por:

$$V(r) = D_e \left(1 - e^{-a(r-r_e)}\right)^2$$

onde:

- $V(r)$ é o potencial em função da distância r entre os átomos,
- D_e é a profundidade do poço de potencial,
- r_e é a distância de equilíbrio entre os átomos,
- a é um parâmetro que controla a largura do potencial.

A força atuando sobre a partícula é obtida pela derivada negativa do potencial:

$$F(r) = -\frac{dV(r)}{dr} = 2aD_e \left(1 - e^{-a(r-r_e)}\right) e^{-a(r-r_e)}$$

De acordo com a segunda lei de Newton $F = m\frac{d^2r}{dt^2}$, a equação de movimento é:

$$m\frac{d^2r}{dt^2} = 2aD_e \left(1 - e^{-a(r-r_e)}\right) e^{-a(r-r_e)}$$

Reescrevendo a equação de movimento, obtemos:

$$\frac{d^2r}{dt^2} = \frac{2aD_e}{m} \left(1 - e^{-a(r-r_e)}\right) e^{-a(r-r_e)}$$

O método de Verlet Velocity é utilizado para resolver a equação de movimento numericamente. As equações do método são:

$$r(t + \Delta t) = r(t) + v(t)\Delta t + \frac{1}{2}a(t)(\Delta t)^2$$

$$v(t + \Delta t) = v(t) + \frac{1}{2}[a(t) + a(t + \Delta t)]\Delta t$$

onde $a(t)$ é a aceleração no instante t, e Δt é o passo de tempo. Utilizamos o método de Verlet Velocity para simular o movimento da partícula ao longo do tempo. A seguir estão as condições iniciais e parâmetros usados:

- Posição inicial $r(0) = 0.001$
- Velocidade inicial $v(0) = 0.0$
- Passo de tempo $\Delta t = 0.0001$
- Número de passos de tempo steps = 100000
- Parâmetros do potencial: $D_e = 0.05$, $a = 0.05$, $r_e = 1.0$, e $m = 1.0$

O código em C a seguir realiza a simulação e calcula a energia total da partícula em cada passo de tempo:

```
#include <stdio.h>
#include <math.h>

// Parâmetros do potencial de Morse
double D_e;
double a;
double r_e;
double m;

// Variáveis globais para a simulação
double r;   // Posição inicial
double v;   // Velocidade inicial
double dt;  // Passo de tempo
int steps;  // Número de passos de tempo

double accel;       // Aceleração inicial
double r_next;      // Posição no próximo passo
double v_half;      // Velocidade no meio do passo
double accel_next;  // Aceleração no próximo passo
double E_kinetic;   // Energia cinética
double E_potential; // Energia potencial
double E_total;     // Energia total
```

```
// Função que calcula a força F(r) no potencial de Morse
double force(double r) {
    return 2. * a * D_e * (1 - exp(-a * (r - r_e))) * exp(-a * (r - r_e));
}

// Função que calcula a energia potencial no potencial de Morse
double potential_energy(double r) {
    return D_e * pow(1 - exp(-a * (r - r_e)), 2);
}

// Função que calcula a energia cinética
double kinetic_energy(double v) {
    return 0.5 * m * v * v;
}

int main() {
    // Inicializa parâmetros e variáveis
    r = 0.001;   // Posição inicial
    v = 0.0;     // Velocidade inicial
    dt = 0.0001; // Passo de tempo
    steps = 100000; // Número de passos de tempo
    D_e = 0.05;
    a = 0.05;
    r_e = 1.0;
    m = 1.0;

    // Calcula a aceleração inicial
    accel = force(r) / m;

    // Método de Verlet Velocity
    for (int i = 0; i < steps; i++) {
        // Atualiza a posição
        r_next = r + v * dt + 0.5 * accel * dt * dt;

        // Atualiza a velocidade no meio do passo
        v_half = v + 0.5 * accel * dt;

        // Calcula a nova aceleração
        accel_next = force(r_next) / m;

        // Atualiza a velocidade no final do passo
        v = v_half + 0.5 * accel_next * dt;

        // Atualiza a posição e aceleração para o próximo passo
        r = r_next;
        accel = accel_next;

        // Calcula a energia total (cinética + potencial) para monitoramento
        E_kinetic = kinetic_energy(v);
        E_potential = potential_energy(r);
        E_total = E_kinetic + E_potential;

        // Salva a posição, tempo e energia em um arquivo ou imprime na tela
        printf("%lf\t%f\t%f\n", dt * (double)i, r, E_total);
    }

    return 0;
}
```

O código realiza a simulação do movimento da partícula e calcula a energia total para cada passo de tempo. O método de Verlet Velocity é eficiente para simular sistemas dinâmicos e permite monitorar a energia total ao longo do tempo, o que é útil para verificar a conservação da energia no sistema.

0.23 Simulação de uma Partícula no Potencial de Lennard-Jones

Nesta seção, implementaremos uma simulação numérica de uma partícula de massa $m = 1$ sujeita ao potencial de Lennard-Jones, utilizando o método de Verlet Velocity para a evolução temporal. O potencial de Lennard-Jones é amplamente utilizado para modelar interações entre partículas, especialmente em simulações de dinâmica molecular. O potencial de Lennard-Jones é descrito pela equação:

$$V(r) = 4\epsilon \left[\left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6} \right],$$

onde:

- ϵ é a profundidade do poço de potencial, indicando a intensidade da interação;
- σ é a distância onde o potencial é zero;
- r é a distância entre as partículas.

O termo $\left(\frac{\sigma}{r}\right)^{12}$ representa a repulsão a curta distância devido à sobreposição das nuvens eletrônicas, enquanto o termo $\left(\frac{\sigma}{r}\right)^{6}$ modela a atração de longo alcance (forças de van der Waals). A força $F(r)$ atuando sobre a partícula é obtida pela derivada negativa do potencial:

$$F(r) = -\frac{dV(r)}{dr} = 24\epsilon \left[2\left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6} \right] \frac{1}{r}.$$

A energia total de uma partícula sujeita ao potencial de Lennard-Jones é dada pela soma da energia cinética e da energia potencial:

$$E_{\text{total}} = E_{\text{cinética}} + E_{\text{potencial}},$$

onde a energia cinética $E_{\text{cinética}}$ é expressa como:

$$E_{\text{cinética}} = \frac{1}{2}mv^2,$$

e a energia potencial $E_{\text{potencial}}$ é dada pela função $V(r)$. O método de Verlet Velocity é uma técnica comum para integrar equações de movimento. Neste método, as equações de atualização da posição r e velocidade v são expressas como:

$$r(t + \Delta t) = r(t) + v(t)\Delta t + \frac{1}{2}a(t)\Delta t^2,$$

onde $a(t)$ é a aceleração no instante t. A velocidade é atualizada da seguinte maneira:

$$v(t + \Delta t) = v(t) + \frac{1}{2}\left[a(t) + a(t + \Delta t)\right]\Delta t.$$

A seguir, apresentamos o código em C que implementa a simulação de uma partícula de massa $m = 1$ no potencial de Lennard-Jones. O programa grava o tempo, a posição, a energia cinética, a energia potencial e a energia total em um arquivo chamado `LJ.dat`. Além disso, ele grava o potencial de Lennard-Jones em função da distância em um arquivo separado, `LJ_potential.dat`, para que o potencial possa ser visualizado posteriormente.

```
#include <stdio.h>
#include <math.h>

// Parâmetros do potencial de Lennard-Jones
double epsilon = 1.0; // Profundidade do poço de potencial
double sigma = 3.0;   // Distância onde o potencial é zero
double m = 1.0;       // Massa da partícula

// Variáveis globais para a simulação
```

```
double r;   // Posição inicial
double v;   // Velocidade inicial
double dt;  // Passo de tempo
int steps;  // Número de passos de tempo

double accel;       // Aceleração inicial
double r_next;      // Posição no próximo passo
double v_half;      // Velocidade no meio do passo
double accel_next;  // Aceleração no próximo passo
double E_kinetic;   // Energia cinética
double E_potential; // Energia potencial
double E_total;     // Energia total

// Função que calcula a força F(r) no potencial de Lennard-Jones
double force(double r) {
    double inv_r = sigma / r;
    double inv_r6 = pow(inv_r, 6);
    double inv_r12 = inv_r6 * inv_r6;
    return 24 * epsilon * (2 * inv_r12 - inv_r6) / r;
}

// Função que calcula a energia potencial no potencial de Lennard-Jones
double potential_energy(double r) {
    double inv_r = sigma / r;
    double inv_r6 = pow(inv_r, 6);
    double inv_r12 = inv_r6 * inv_r6;
    return 4 * epsilon * (inv_r12 - inv_r6);
}

// Função que calcula a energia cinética
double kinetic_energy(double v) {
    return 0.5 * m * v * v;
}

int main() {
    // Abrir arquivo para escrever os resultados da simulação
    FILE *file = fopen("LJ.dat", "w");
    if (file == NULL) {
        printf("Erro ao abrir o arquivo!\n");
        return 1;
    }

    // Abrir arquivo para gravar o potencial de Lennard-Jones
    FILE *file_potential = fopen("LJ_potential.dat", "w");
    if (file_potential == NULL) {
        printf("Erro ao abrir o arquivo para o potencial!\n");
        return 1;
    }

    // Definindo as condições iniciais
    r = 3.3;   // Posição inicial (não deve ser muito próxima de 0 para evitar singularidade)
    v = 0.;   // Velocidade inicial
    dt = 0.001; // Passo de tempo
    steps = 100000; // Número de passos de tempo

    // Calcula a aceleração inicial
    accel = force(r) / m;

    // Escrevendo o potencial de Lennard-Jones no intervalo de r
```

```
    double r_min = 3.05;
    double r_max = 5.0;
    int potential_points = 1000;
    double dr = (r_max - r_min) / potential_points;

    for (double r_value = r_min; r_value <= r_max; r_value += dr) {
        double potential = potential_energy(r_value);
        fprintf(file_potential, "%lf\t%lf\n", r_value, potential);
    }

    // Método de Verlet Velocity
    for (int i = 0; i < steps; i++) {
        // Atualiza a posição
        r_next = r + v * dt + 0.5 * accel * dt * dt;

        // Atualiza a velocidade no meio do passo
        v_half = v + 0.5 * accel * dt;

        // Calcula a nova aceleração
        accel_next = force(r_next) / m;

        // Atualiza a velocidade no final do passo
        v = v_half + 0.5 * accel_next * dt;

        // Atualiza a posição e aceleração para o próximo passo
        r = r_next;
        accel = accel_next;

        // Calcula as energias
        E_kinetic = kinetic_energy(v);
        E_potential = potential_energy(r);
        E_total = E_kinetic + E_potential;

        // Escreve o tempo, posição e energias no arquivo
        fprintf(file, "%lf\t%lf\t%lf\t%lf\t%lf\n", dt * i, r, E_kinetic, E_potential, E_total);
    }

    // Fechar os arquivos
    fclose(file);
    fclose(file_potential);

    return 0;
}
```

Este programa utiliza o método de Verlet Velocity para calcular a posição e a velocidade da partícula a cada passo de tempo, e também grava os dados do potencial de Lennard-Jones em função de r em um arquivo separado. O potencial pode ser plotado posteriormente usando ferramentas como Gnuplot ou xmgrace (por exemplo).

0.24 Simulação de uma Cadeia com Potencial de Morse: Energia e Largura do Pulso

Neste capítulo, simulamos uma cadeia composta por N massas m_i acopladas através do potencial de Morse. O método de integração numérica utilizado será o *Verlet Velocity*, e focaremos em calcular a energia total da cadeia (energia cinética e potencial) e a evolução da largura do pulso de energia ao longo do tempo. Além disso, os dados serão gravados em um arquivo chamado `cadeiamorse.dat`. O potencial de Morse entre duas massas i e $i+1$ é dado por:

$$V(r) = D_e \left(1 - e^{-a(r-r_e)}\right)^2,$$

onde D_e é a profundidade do poço de potencial, a é a largura do poço, r é a distância entre as massas e r_e é a distância de equilíbrio. A força derivada desse potencial é obtida pela derivada de $V(r)$ em relação à posição r:

$$F(r) = 2aD_e \left(1 - e^{-a(r-r_e)}\right) e^{-a(r-r_e)}.$$

Consideramos que todas as massas estão inicialmente em repouso e na posição de equilíbrio, exceto pela massa central $m_{N/2}$, que é deslocada ligeiramente por uma quantidade 0.01 da posição de equilíbrio. Assim, as condições iniciais para a posição e velocidade de cada massa i são:

$$r_i(0) = r_e \quad \text{para} \quad i \neq N/2, \quad r_{N/2}(0) = r_e + 0.01,$$

$$v_i(0) = 0 \quad \text{para todos os} \quad i.$$

A energia total da cadeia é composta pela energia cinética E_{cin} e a energia potencial E_{pot}. A energia cinética para uma massa m_i é dada por:

$$E_{\text{cin}} = \frac{1}{2} m_i v_i^2,$$

e a energia potencial entre duas massas vizinhas i e $i+1$ é:

$$E_{\text{pot}} = D_e \left(1 - e^{-a(r_i - r_e)}\right)^2.$$

Para calcular a largura do pulso de energia ao longo da cadeia, definimos a "largura" como a extensão onde a energia se concentra. Vamos usar o desvio padrão da distribuição de energia em torno do centro da cadeia como uma medida da largura do pulso. O seguinte código C implementa essas condições e calcula as energias e a largura do pulso ao longo do tempo, gravando os resultados no arquivo `cadeiamorse.dat`:

```
#include <stdio.h>
#include <math.h>

#define NMAX 10000

// Parâmetros do potencial de Morse
double De = 1.0;
double a = 1.0;
double re = 1.0;

// Número de massas na cadeia
int N;

// Arrays para armazenar massas, posições, velocidades, acelerações e energias
double m[NMAX];
double r[NMAX];
double v[NMAX];
double a_current[NMAX];
double a_next[NMAX];

// Parâmetros de tempo e simulação
double dt;
int steps;

// Energia total e largura do pulso
double E_total, width;

// Função que calcula a força a partir do potencial de Morse
double force_morse(double r_ij) {
    double term = exp(-a * (r_ij - re));
    return 2 * a * De * (1 - term) * term;
}
```

```
// Função para calcular a energia potencial entre duas massas
double potential_morse(double r_ij) {
    double term = exp(-a * (r_ij - re));
    return De * pow(1 - term, 2);
}

// Inicialização das posições e velocidades
void initialize(double *r, double *v, int N) {
    for (int i = 0; i < N; i++) {
        r[i] = re;
        v[i] = 0.0;
        m[i] = 1.0 ;  // Massas constantes
    }
    // Desloca a massa central levemente
    r[N/2] = re + 0.01;
}

// Calcula as forças e atualiza as acelerações
void calculate_forces(double *r, double *a_current, int N) {
    for (int i = 1; i < N - 1; i++) {
        double r_ij = r[i] - r[i - 1];
        double r_ik = r[i + 1] - r[i];

        double F_left = force_morse(fabs(r_ij));
        double F_right = force_morse(fabs(r_ik));

        a_current[i] = (F_right - F_left) / m[i];
    }
}

// Calcula a energia total (cinética e potencial) e a largura do pulso
void calculate_energy_and_width(double *r, double *v, int N, double *E_total, double *width) {
    double E_cin = 0.0, E_pot = 0.0;
    double mean_energy = 0.0, mean_position = 0.0, variance = 0.0;

    // Calcula a energia cinética e potencial
    for (int i = 1; i < N - 1; i++) {
        E_cin += 0.5 * m[i] * v[i] * v[i];
        E_pot += potential_morse(fabs(r[i] - r[i - 1]));
    }

    // Energia total
    *E_total = E_cin + E_pot;

    // Calcula a largura do pulso de energia
    for (int i = 1; i < N - 1; i++) {
        double energy_density = 0.5 * m[i] * v[i] * v[i] + potential_morse(fabs(r[i] - r[i - 1]));
        mean_energy += energy_density;
        mean_position += energy_density * i;
    }

    mean_position /= mean_energy;

    for (int i = 1; i < N - 1; i++) {
        double energy_density = 0.5 * m[i] * v[i] * v[i] + potential_morse(fabs(r[i] - r[i - 1]));
        variance += energy_density * pow(i - mean_position, 2);
    }

    *width = sqrt(variance / mean_energy);
```

```
}

int main() {
    // Definindo N e o passo de tempo
    N = 100;
    dt = 0.001;
    steps = 100000;

    FILE *file = fopen("cadeiamorse.dat", "w");

    // Inicializa as posições e velocidades
    initialize(r, v, N);

    // Calcula a aceleração inicial
    calculate_forces(r, a_current, N);

    // Simulação usando Verlet Velocity
    for (int step = 0; step < steps; step++) {
        for (int i = 1; i < N - 1; i++) {
            double r_next = r[i] + v[i] * dt + 0.5 * a_current[i] * dt * dt;
            double v_half = v[i] + 0.5 * a_current[i] * dt;

            // Calcula a nova força após atualizar a posição
            calculate_forces(r, a_next, N);

            v[i] = v_half + 0.5 * a_next[i] * dt;
            r[i] = r_next;
            a_current[i] = a_next[i];
        }

        // A cada passo, calcula a energia total e a largura do pulso
        if (step % 100 == 0) {
            calculate_energy_and_width(r, v, N, &E_total, &width);
            fprintf(file, "%f %f %f\n", step * dt, width, E_total);
        }
    }

    fclose(file);
    return 0;
}
```

Simulamos a cadeia de massas acopladas com o potencial de Morse, calculando a energia total e a largura do pulso de energia ao longo do tempo. Os resultados foram salvos no arquivo `cadeiamorse.dat`, o que permite a análise posterior das dinâmicas da cadeia. Este exemplo ilustra como variações iniciais pequenas podem se propagar ao longo da cadeia, resultando em uma interessante dinâmica de energia.

0.25 Simulação de uma Cadeia harmônica com Método de Verlet Velocity

Nesta seção, iremos resolver numericamente a evolução temporal de uma cadeia massa-mola usando o método de Verlet Velocity. O sistema é composto por N massas conectadas por molas, e a força restauradora em cada massa é dada por $-kx$, onde k é a constante elástica da mola e x é o deslocamento da posição de equilíbrio. A energia potencial associada a cada mola é $U(x) = \frac{kx^2}{2}$. Para simplificação, consideraremos $k = 1$ e $m = 1$, e utilizaremos a seguinte configuração inicial:

- Todas as massas estão em repouso com posição inicial $x_i = 0$.
- A massa central ($i = N/2$) tem uma velocidade inicial $v_{N/2} = 0.1$.

O método de Verlet Velocity é utilizado para a integração temporal das equações de movimento, sendo uma técnica de segunda ordem que permite calcular as novas posições e velocidades de cada massa a partir do estado anterior do sistema. O algoritmo segue os seguintes passos:

1. Atualizamos as posições utilizando:

$$x_i(t + \Delta t) = x_i(t) + v_i(t)\Delta t + \frac{1}{2}a_i(t)\Delta t^2$$

2. Calculamos as novas acelerações baseadas nas novas posições.

3. Atualizamos as velocidades:

$$v_i(t + \Delta t) = v_i(t) + \frac{1}{2}(a_i(t) + a_i(t + \Delta t))\Delta t$$

O código abaixo implementa este algoritmo para uma cadeia de $N = 2000$ massas, com passo de tempo $\Delta t = 0.01$ e um total de 50000 passos de tempo:

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#define NMAX 10000   // Tamanho máximo da cadeia
#define dt 0.01      // Passo de tempo
#define T 50000      // Número total de passos de tempo

int N;
```

```
double x[NMAX];   // Posições das massas
double v[NMAX];   // Velocidades das massas
double a[NMAX];   // Acelerações das massas
double a_antiga[NMAX];
double m[NMAX];   // Massas das partículas
double F[NMAX], sum_F, mean_i, sigma;
double energia_potencial, energia_cinetica, energia, energia_total;
int i, t;

// Função que inicializa as massas aleatoriamente no intervalo [1, 3]
void inicializar_massas() {
    for (i = 0; i < N; i++) {
        m[i] = 1.0 + 2.0 * ((double)rand() / RAND_MAX); // [1, 3]
    }
}

// Função que calcula as forças de interação (aceleração)
void calcular_aceleracoes() {
    for (i = 0; i < N; i++) {
        if (i == 0) {
            a[i] = -(x[i] - x[i + 1]) / m[i];  // Massa na extremidade esquerda
        } else if (i == N - 1) {
            a[i] = -(x[i] - x[i - 1]) / m[i];  // Massa na extremidade direita
        } else {
            a[i] = -(2 * x[i] - x[i - 1] - x[i + 1]) / m[i];  // Massa no meio
        }
    }
}

// Função para calcular a energia total do sistema
double calcular_energia_total() {
    energia = 0.0;
    for (i = 0; i < N; i++) {
        energia_potencial = 0.0;
        if (i > 0) {
            energia_potencial = 0.5 * pow(x[i] - x[i - 1], 2);
        }
        energia_cinetica = 0.5 * m[i] * v[i] * v[i];
        energia += energia_potencial + energia_cinetica;
    }
    return energia;
}

// Função para calcular a largura do pulso de energia (sigma)
double largura_pulso() {
    sum_F = mean_i = sigma = 0.0;
    for (i = 0; i < N; i++) {
        energia_potencial = 0.0;
        if (i > 0) {
            energia_potencial = 0.5 * pow(x[i] - x[i - 1], 2);
        }
        energia_cinetica = 0.5 * m[i] * v[i] * v[i];
        F[i] = energia_potencial + energia_cinetica;
        sum_F += F[i];
    }
    for (i = 0; i < N; i++) {
        mean_i += i * F[i];
    }
    mean_i /= sum_F;
```

```
    for (i = 0; i < N; i++) {
        sigma += (i - mean_i) * (i - mean_i) * F[i];
    }
    sigma /= sum_F;
    return sigma;
}

int main() {
    N = 2000;  // Definimos o valor de N dentro do main

    FILE *saida;
    saida = fopen("cadeiaharmonica.dat", "w");
    if (saida == NULL) {
        printf("Erro ao abrir o arquivo cadeiaharmonica.dat\n");
        return 1;
    }

    // Inicializar as massas aleatoriamente
    inicializar_massas();

    // Condições iniciais: todas as massas em repouso exceto a central
    for (i = 0; i < N; i++) {
        x[i] = 0.0;
        v[i] = 0.0;
    }
    v[N / 2] = 0.1;  // Massa central com velocidade inicial 0.1

    // Primeira etapa do algoritmo de Verlet Velocity
    calcular_aceleracoes();  // Inicializar as acelerações

    // Laço no tempo
    for (t = 0; t < T; t++) {
        // Atualizar posições
        for (i = 0; i < N; i++) {
            x[i] += v[i] * dt + 0.5 * a[i] * dt * dt;
        }

        // Guardar acelerações antigas
        for (i = 0; i < N; i++) {
            a_antiga[i] = a[i];
        }

        // Recalcular novas acelerações
        calcular_aceleracoes();

        // Atualizar velocidades
        for (i = 0; i < N; i++) {
            v[i] += 0.5 * (a_antiga[i] + a[i]) * dt;
        }

        // A cada 100 passos, calcular e escrever os resultados no arquivo
        if (t % 100 == 0) {
            energia_total = calcular_energia_total();
            sigma = largura_pulso();
            if (t > 0) {
                fprintf(saida, "%f %f %f\n", t * dt, sigma, energia_total);
            }
        }
    }
```

```
    fclose(saida);
    return 0;
}
```

Este programa simula a dinâmica de uma cadeia unidimensional de partículas conectadas, onde cada partícula possui uma massa variável aleatoriamente distribuída no intervalo $[1, 3]$. A simulação utiliza um modelo de cadeia harmônica com interação entre vizinhos imediatos, onde a força restauradora é proporcional à deformação do sistema. Um pequeno destaque para as principais etapas deste programa pode ser encontrado abaixo:

- **Inicialização das Massas:** As massas das partículas são geradas aleatoriamente e armazenadas em um vetor $m[i]$ com valores entre 1 e 3.
- **Cálculo das Acelerações:** A função `calcular_aceleracoes` determina as acelerações das partículas com base nas forças restauradoras, considerando as massas variáveis.
- **Energia do Sistema:** A energia total, calculada pela função `calcular_energia_t` inclui a energia potencial e cinética.
- **Largura do Pulso de Energia (sigma):** A função `largura_pulso` mede a dispersão da energia ao longo da cadeia, avaliando a distribuição da energia em função das posições das partículas.
- **Simulação:** O método de Verlet Velocity é utilizado para atualizar posições e velocidades das partículas. A cada 100 passos de tempo, a energia total e a largura do pulso de energia são salvas em um arquivo de saída.
- **Energia Total:** Representa a soma da energia cinética e potencial do sistema. A variação da energia ao longo do tempo indica a estabilidade do sistema e a redistribuição da energia entre as partículas.
- **Largura do Pulso de Energia (sigma):** Mede a dispersão da energia ao longo da cadeia. Valores altos de sigma indicam maior variabilidade na distribuição de energia, enquanto valores baixos sugerem uma distribuição mais uniforme.

A análise dos resultados no arquivo `cadeiaharmonica.dat` permite observar como a energia total e a largura do pulso de energia evoluem ao longo

do tempo. Isso pode revelar informações sobre a dinâmica do sistema e o impacto das massas variáveis na sua evolução. Comparar simulações com diferentes distribuições de massa fornece insights sobre a sensibilidade do sistema a variações nas propriedades das partículas.

0.26 Considerações Finais

Ao final deste livro, é possível refletir sobre a jornada pela física computacional, uma disciplina que, ao integrar teoria e prática, nos permite enfrentar desafios complexos em diversos campos da física. Desde o início, o foco foi mostrar como os métodos computacionais podem ser aplicados para compreender sistemas que, de outra forma, seriam intratáveis por meio de abordagens puramente analíticas.

Este livro abordou uma ampla gama de tópicos, partindo de métodos numéricos fundamentais, como a resolução de equações diferenciais e a análise de sistemas dinâmicos simples, até técnicas avançadas de simulação e análise de fenômenos físicos mais complexos. Ao longo deste percurso, foi possível ver como a física computacional se tornou uma ferramenta indispensável na pesquisa moderna, não apenas para resolver problemas teóricos, mas também para entender o comportamento de sistemas físicos reais em áreas como a mecânica quântica, a termodinâmica e a teoria do caos.

A física computacional é, sem dúvida, uma ponte entre o mundo abstrato da matemática e o mundo prático dos experimentos. Através da implementação e análise de algoritmos numéricos, como o método de Runge-Kutta e suas variantes, os leitores foram capacitados a explorar sistemas que variam de sistemas de partículas a modelos de campo, desde problemas determinísticos até sistemas estocásticos. Essas habilidades práticas são fundamentais para qualquer físico ou cientista que pretenda trabalhar na fronteira entre teoria e experimentação, onde a capacidade de simular fenômenos complexos pode abrir novas portas para a descoberta científica.

Os exemplos práticos fornecidos ao longo dos capítulos não tinham apenas o objetivo de ilustrar os conceitos teóricos, mas também de permitir que os leitores adquirissem uma experiência prática valiosa. A implementação desses métodos em linguagens de programação como C reflete a realidade de muitos projetos de pesquisa, onde a eficiência computacional e a precisão dos resultados são de suma importância. A experiência adquirida na construção de códigos, na resolução de problemas e na análise de dados é algo que transcende os exemplos apresentados aqui, preparando os leitores para enfrentar novos desafios em suas próprias investigações.

Além disso, o livro buscou mostrar a versatilidade da física computacional em diversas áreas da ciência, desde a análise de dados experimentais até a modelagem de fenômenos naturais. A importância da experimentação numérica, combinada com a compreensão profunda dos métodos utilizados, foi enfatizada para incentivar os leitores a se tornarem não apenas usuários de ferramentas computacionais, mas criadores de novas abordagens para problemas ainda não resolvidos.

Em conclusão, a física computacional é uma disciplina dinâmica e em con-

stante evolução. A contínua exploração e aplicação das técnicas discutidas neste livro abrirá novas oportunidades para a resolução de problemas desafiadores em diversas áreas do conhecimento. À medida que os computadores se tornam mais poderosos e os algoritmos mais sofisticados, a física computacional continuará a expandir suas fronteiras, contribuindo de maneira decisiva para o avanço do conhecimento científico e tecnológico. Assim, o que foi explorado aqui é apenas o começo de uma jornada que os leitores poderão continuar a trilhar em suas próprias investigações, desenvolvendo novos métodos e abordagens que, em última instância, ajudarão a desvendar os mistérios do universo.