Problema de modelagem de um reator químico

Métodos Numéricos para Equações Diferenciais

Autores:
Lucas Rodrigues Estorck Pinto - 202310349511
Tainá Martins Macário - 202310070411

Professor:
Grazione Souza

Rio de Janeiro, Nova Friburgo

20 de setembro de 2025

Conteúdo

1 Introdução **2**

2 Metodologia **3**

3 Implementação computacional **5**

3.1 Estrutura do código, organização dos dados e implementação 5

3.2 Análise de complexibilidade e perfomance 6

3.3 Convergência, estabilidade e sensibilidade 7

3.4 Ambiente computacional e modularidade 8

4 Resultados e discussão **9**

5 Conclusão **15**

Referências Bibliográficas **16**

Apêndice A: Tabelas de análise de convergência **17**

Apêndice B: Código fonte **20**

1 Introdução

Este trabalho tem como objetivo aplicar o método de Runge-Kutta de 4° ordem no contexto da disciplina de Métodos Numéricos para Equações Diferenciais. O contexto de aplicação é a análise de um reator de batelada, onde a produção de penicilina por fermentação é controlada pelo sistema de equações mostrado nas Equações (1) e (2).

$$\frac{dg}{dt} = 13.1g - 13.94g^2 \tag{1}$$

$$\frac{df}{dt} = 1.71g \tag{2}$$

Onde g é a concentração adimensional da massa celular, f é a concentração adimensional de penicilina e t é o tempo adimensional no intervalo $0 \leq t \leq 1$. As condições iniciais para a solução do problema são:

$$g(0) = 0.03 \qquad\qquad f(0) = 0.0$$

Para a resolução, o algoritmo de Runge-Kutta de 4° ordem será implementado na linguagem de programação *Julia*, uma linguagem de alto nível e dinâmica, projetada para computação científica e numérica de alto desempenho. Ela combina características de linguagens interpretadas com a eficiência de linguagens compiladas. Dessa forma, *Julia* oferece a velocidade de uma linguagem compilada com a facilidade de uso do Python, a linguagem mais conhecida e implementada atualmente. [1]

Por fim, os resultados obtidos com a implementação serão discutidos e analisados. Espera-se, assim, avaliar a eficiência do método numérico aplicado e discutir sua aplicabilidade em sistemas reais, onde a resolução de equações diferenciais por métodos numéricos é necessária.

2 Metodologia

A metodologia utilizada para a resolução do problema baseou-se na aplicação do método de Runge-Kutta clássico de quarta ordem (RK4). Desenvolvido no início do século XX, este método representa uma classe de algoritmos empregados para a solução numérica de Equações Diferenciais Ordinárias (EDOs) e se destaca pelo equilíbrio entre precisão, estabilidade e custo computacional.

A forma geral de um método de Runge-Kutta é expressa pela Equação (3), onde s é o número de estágios, os coeficientes k_i são os incrementos/inclinações calculados em diferentes pontos do intervalo e b_i são os pesos que ponderam tais incrementos. [2]

$$y_{n+1} = y_n + \sum_{i=1}^{s} b_i k_i \tag{3}$$

O método clássico de quarta ordem é o mais difundido do grupo. Ele estima a solução de um Problema de Valor Inicial (PVI) utilizando uma média ponderada de quatro incrementos (k_1, k_2, k_3 e k_4), que representam a inclinação da função em diferentes pontos dentro do intervalo de tempo Δt. Para um PVI da forma $\frac{d\phi}{dt} = f(t, y)$, a iteração para encontrar o próximo valor ϕ_{n+1} a partir de um valor conhecido ϕ_n é dada pela Equação (4).

$$\phi^{n+1} = \phi^n + \tfrac{1}{6}\left(k_1 + 2k_2 + 2k_3 + k_4\right)\Delta t \tag{4}$$

Os incrementos são definidos por:

$$k_1 = \left(\frac{d\phi}{dt}\right)^n \tag{5}$$

$$k_2 = \frac{d\phi}{dt}\left(t^n + \tfrac{\Delta t}{2},\, \phi^n + k_1\tfrac{\Delta t}{2}\right) \tag{6}$$

$$k_3 = \frac{d\phi}{dt}\left(t^n + \tfrac{\Delta t}{2},\, \phi^n + k_2\tfrac{\Delta t}{2}\right) \tag{7}$$

$$k_4 = \frac{d\phi}{dt}\left(t^n + \Delta t,\, \phi^n + k_3\Delta t\right) \tag{8}$$

No contexto deste trabalho, o método RK4 foi aplicado ao sistema de equações diferenciais que descreve o reator de batelada. Como o modelo apresenta duas equações acopladas, relativas à concentração adimensional de células (g) e de penicilina (f), os incrementos k_1, k_2, k_3 e k_4 foram calculados separadamente para cada variável, mas em instantes intermediários de tempo idênticos. Para sistemas acoplados da forma:

$$\frac{dg}{dt} = f_g(t, g, f)$$
$$\frac{df}{dt} = f_f(t, g, f)$$

O método RK4 mantém a consistência temporal avaliando as derivadas de ambas as equações nos mesmos pontos intermediários, preservando assim o acoplamento entre as variáveis e garantindo que a ordem de precisão do método seja mantida para o sistema como um todo.

O passo de integração inicial foi definido como $\Delta t = 0{,}05$, conforme a especificação do problema. Posteriormente, foram realizados testes de convergência numérica, diminuindo-se gradualmente o valor de Δt para verificar se os resultados obtidos tendiam à solução de referência, comportamento esperado de um método de quarta ordem. Essa etapa permitiu avaliar a confiabilidade e a ordem de precisão efetiva do esquema implementado.

Além disso, foi conduzida uma análise de sensibilidade, em que os parâmetros cinéticos do modelo (coeficientes lineares e quadráticos) foram modificados em variações percentuais positivas e negativas. Esse procedimento teve como objetivo avaliar a influência de incertezas experimentais ou variações nos coeficientes sobre a dinâmica do reator, fornecendo uma compreensão mais ampla do comportamento do sistema.

Toda a formulação foi implementada na linguagem *Julia*, escolhida por sua eficiência em cálculos científicos e suporte a bibliotecas de visualização. O código foi estruturado de modo a resolver numericamente o sistema de EDOs, registrar os resultados em formato tabular, gerar gráficos de evolução temporal de $g(t)$ e $f(t)$ e automatizar a análise de convergência e sensibilidade. Essa integração entre modelagem matemática, implementação computacional e análise crítica assegurou que os resultados numéricos obtidos fossem não apenas consistentes, mas também interpretáveis no contexto da engenharia química.

3 Implementação computacional

A implementação numérica do problema foi realizada na linguagem *Julia*, versão 1.11.7+0. Esta linguagem se destaca pela combinação de sintaxe de alto nível e desempenho comparável ao de linguagens compiladas, como C e C++. Tal performance é viabilizada pelo modelo de compilador just-in-time (JIT) que utiliza a infraestrutura LLVM para gerar código nativo otimizado em tempo de execução [1].

A natureza fortemente tipada da linguagem, combinada com inferência automática de tipos, assegura que o código escrito em estilo conciso e expressivo seja convertido em rotinas eficientes, eliminando a necessidade de recorrer a extensões em C ou Fortran para obter desempenho adequado, como frequentemente observado em ambientes baseados em Python [3, 4].

3.1 Estrutura do código, organização dos dados e implementação

O código foi estruturado de forma modular, utilizando estruturas de dados (`struct`) para organizar os parâmetros do modelo e os resultados das simulações. A estrutura `ReactorParams` encapsula os coeficientes cinéticos do reator:

```
struct ReactorParams
    a1::Float64  # Coeficiente linear (13.1)
    a2::Float64  # Coeficiente quadrático (13.94)
    b1::Float64  # Coeficiente produção penicilina (1.71)
end
```

Esta abordagem garante type safety e facilita a parametrização para análises de sensibilidade, além de melhorar a legibilidade e manutenibilidade do código.

O núcleo da implementação consiste na função `runge_kutta_4`, que resolve o sistema de equações diferenciais ordinárias através do método RK4 clássico. A implementação segue rigorosamente a formulação matemática apresentada na Equação (4), adaptada para sistemas acoplados.

Para um sistema de EDOs da forma:

$$\frac{dg}{dt} = f_g(t, g, f) = a_1 g - a_2 g^2 \quad \frac{df}{dt} \qquad = f_f(t, g, f) = b_1 g \tag{9}$$

A função central `runge_kutta_4` implementa o algoritmo através da pré-alocação, inicialização e cálculo dos coeficientes k para f. Abaixo, segue o trecho do código onde implementa-se a inicialização e o cálculo dos coeficientes:

```
# Número de passos
n_steps = Int(ceil(t_max / dt))
```

```
        # Inicializar vetores de resultados
        t = Vector{Float64}(undef, n_steps + 1)
        g = Vector{Float64}(undef, n_steps + 1)
        f = Vector{Float64}(undef, n_steps + 1)
```

A pré-alocação dos vetores de solução é uma otimização fundamental que evita realocações dinâmicas durante a execução, reduzindo significativamente o overhead computacional e melhorando a localização de memória. Complementarmente, o cálculo dos coeficientes:

```
    Cálculo dos coeficientes k para g
        k1_g = dg_dt(g_n, f_n, t_n, params)
        k2_g = dg_dt(g_n + k1_g * dt/2, f_n + k1_f * dt/2, t_n + dt/2, params)
        k3_g = dg_dt(g_n + k2_g * dt/2, f_n + k2_f * dt/2, t_n + dt/2, params)
        k4_g = dg_dt(g_n + k3_g * dt, f_n + k3_f * dt, t_n + dt, params)
    Cálculo dos coeficientes k para f
        k1_f = df_dt(g_n, f_n, t_n, params)
        k2_f = df_dt(g_n + k1_g * dt/2, f_n + k1_f * dt/2, t_n + dt/2, params)
        k3_f = df_dt(g_n + k2_g * dt/2, f_n + k2_f * dt/2, t_n + dt/2, params)
        k4_f = df_dt(g_n + k3_g * dt, f_n + k3_f * dt, t_n + dt, params)
```

É importante notar que, para sistemas acoplados, os coeficientes k_i de ambas as variáveis são calculados nos mesmos instantes intermediários de tempo, garantindo consistência no avanço temporal e preservando o acoplamento entre as equações. Este tratamento é essencial para manter a precisão de quarta ordem do método quando aplicado a sistemas de EDOs.

A fórmula de atualização implementa diretamente a Equação (4):

```
        g[i+1] = g_n + (dt/6) * (k1_g + 2k2_g + 2k3_g + k4_g)
        f[i+1] = f_n + (dt/6) * (k1_f + 2k2_f + 2k3_f + k4_f)
        t[i+1] = t_n + dt
```

3.2 Análise de complexibilidade e perfomance

O custo computacional do algoritmo implementado é $O(N)$, onde N representa o número de passos temporais. Cada iteração realiza um número fixo de operações aritméticas (16 avaliações de função para o sistema 2Œ2), resultando em complexidade linear em relação ao número total de pontos da discretização temporal.

A utilização de operações vetorizadas e a pré-alocação de memória contribuem significativamente para a eficiência computacional. Embora o código atual não implemente explicitamente macros de otimização de baixo nível (`@inbounds`, `@simd`), a infraestrutura LLVM do Julia realiza otimizações automáticas que resultam em performance próxima ao código compilado manualmente.

3.3 Convergência, estabilidade e sensibilidade

A validação numérica da implementação foi conduzida através da função analise_convergencia_completa, que verifica empiricamente a ordem de convergência teórica do método RK4. O procedimento compara soluções obtidas com diferentes passos de integração ($\Delta t = 0.05, 0.025, 0.01, 0.005, 0.001$) com uma solução de referência calculada usando $\Delta t = 5 \cdot 10^{-4}$.

O método RK4 é um integrador explícito de quarta ordem, cuja análise teórica de truncammento leva a erro local da ordem $O(h^5)$, implicando um erro global de acumulação da ordem $O(h^4)$ para problemas não rígidos com solução suficientemente suave. [5]. A verificação empírica confirma que, ao reduzir o passo de integração h por um fator r, o fator global diminui aproximadamente por um fator r^4, validando a implementação.

Este método possui região de estabilidade absoluta limitada, definida pela condição $|R(z)| < 1$, onde $R(z)$ é o polinômio de estabilidade associado com $z = \lambda h$ representa o produto do autovalor λ do sistema linearizado pelo passo de integração h. [6]

Para o sistema estudado, que apresenta comportamento não-rígido (autovalores com partes reais de magnitude moderada), o controle da estabilidade é garantido pela escolha adequada do passo temporal. A implementação permite ajuste fácil do passo h através do parâmetro dt, facilitando estudos de estabilidade.

A estrutura modular do código facilita a implementação de análises de sensibilidade, conduzidas através da função `analise_sensibilidade_coeficientes`. Esta função varia sistematicamente cada parâmetro cinético (a_1, a_2 e b_1) em $\pm 5\%$ e $\pm 10\%$, avaliando o impacto sobre as concentrações finais:

```
            for var in variacoes
                fator = 1 + var/100
                a1_novo = a1_orig * fator
                params_mod = ReactorParams(a1_novo, a2_orig, b1_orig)

                t, g, f = runge_kutta_4(g0, f0, dt_base, t_max, params_mod)
                label = var == 0 ? "Original (a=13.1)" : @sprintf("a=%.2f (%+d%%)", a1_novo,
                ↪  var)
                resultado = SimulationResult(t, g, f, params_mod, dt_base, label)
                push!(resultados_a1, resultado)
            end
            resultados_sensibilidade["a1"] = resultados_a1
```

Esta abordagem permite quantificar a robustez da solução frente a incertezas experimentais nos parâmetros cinéticos, fornecendo informações valiosas para a validação do modelo e análise de confiabilidade dos resultados.

3.4 Ambiente computacional e modularidade

A execução foi realizada em um MacBook Air equipado com processador Apple M4 e sistema operacional macOS Sequoia. Esta configuração moderna permite explorar adequadamente as otimizações de baixo nível fornecidas pelo compilador LLVM, resultando em tempos de execução reduzidos sem comprometer a precisão numérica.

O código foi estruturado com foco na reproducibilidade, incluindo geração automática de relatórios tabulares, gráficos de alta qualidade (resolução 300 DPI) e documentação abrangente dos parâmetros utilizados. Todos os resultados são salvos em formatos padrão (PNG para gráficos, TXT para dados tabulares), facilitando a integração com sistemas de documentação LaTeX.

A arquitetura do código permite fácil extensão para métodos de integração mais sofisticados. Para problemas rígidos (*stiff*), onde o RK4 pode exigir passos extremamente pequenos, a estrutura modular facilita a substituição do núcleo integrador por métodos implícitos ou semi-implícitos, como BDF (Backward Differentiation Formulas) ou métodos Rosenbrock, disponíveis no ecossistema Julia através da biblioteca `DifferentialEquations.jl`.

Esta flexibilidade arquitetural garante que o código desenvolvido possa servir como base para estudos mais avançados, mantendo consistência na interface de dados e nos procedimentos de análise implementados.

4 Resultados e discussão

A resolução numérica do sistema de equações diferenciais ordinárias que governa a dinâmica do reator de batelada revelou comportamentos distintos para as concentrações adimensionais de células $g(t)$ e penicilina $f(t)$ ao longo da operação, conforme apresentado nas Figuras (1) e (2).

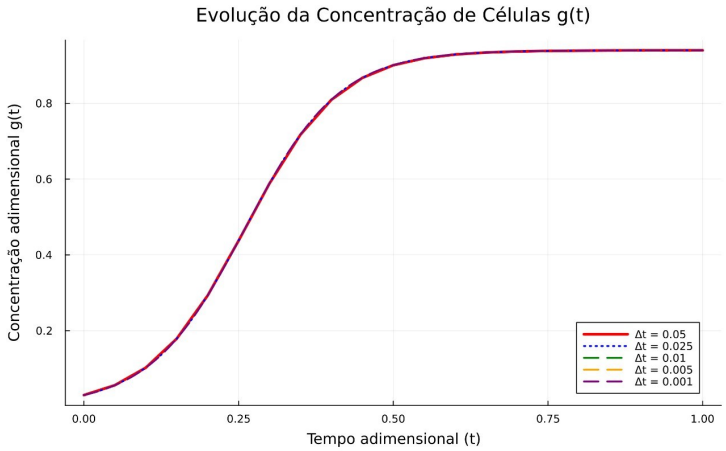

Figura 1: Evolução da concentração de células g(t)

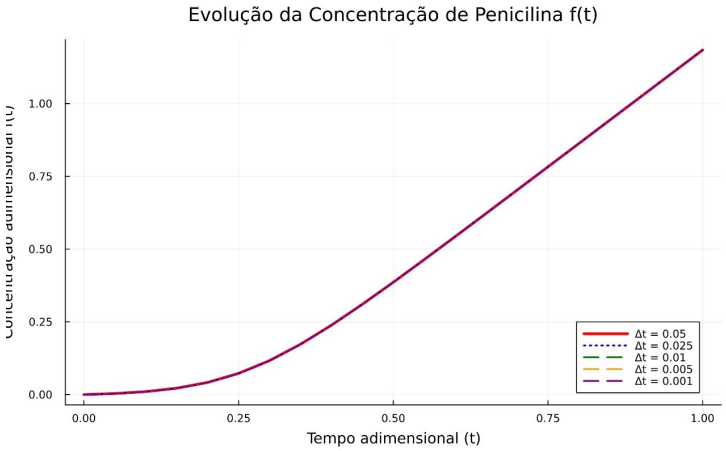

Figura 2: Evolução da concentração de células g(t)

A análise da Figura (1) revela que a concentração adimensional de células $g(t)$ apresenta uma dinâmica característica de crescimento logístico com saturação. Partindo da condição inicial $g(0) = 0.03$, a concentração celular experimenta uma fase de crescimento acelerado até $t \approx 0.4$, momento em que a taxa de crescimento diminui progressivamente devido ao efeito inibitório do termo quadrático $a_2 g^2$ na equação diferencial.

O sistema atinge um regime de semi-equilíbrio em $t \approx 0.7$ com a concentração final convergindo para $g(1) = 0.9397$, considerando $\Delta t = 0.001$ como referência. Este comportamento é consistente com o modelo cinético proposto, onde o termo linear $a_1 g = 13.1g$ representa o crescimento celular e o termo quadrático $a_2 g^2 = 13.94g^2$ modela a limitação por recursos ou inibição por produtos

metabólicos.

A capacidade de suporte efetiva do sistema pode ser estimada por meio de $\frac{dg}{dt} = 0$, resultando em $g_{eq} = \frac{a_1}{a_2} = \frac{13.1}{13.94} = 0.940$, valor que chega extremamente próximo do resultado numérico obtido, com um erro de 0.0319%, apenas.

O comportamento da concentração de penicilina $f(t)$, ilustrado na Figura (2), apresenta características fundamentalmente diferentes do comportamento de $g(t)$. A partir da condição inicial $f(0) = 0.0$, a concentração de penicilina exibe crescimento monotônico aproximadamente linear para $t > 0.5$, atingindo $f(1) = 1.1844$ ao final da simulação.

Este comportamento é explicado pela estrutura da equação $\frac{df}{dt} = b_1 g = 1.71g$ que estabelece que a taxa de produção de penicilina é diretamente proporcional à concentração de células. Durante a fase de crescimento celular exponencial ($t < 0.3$) a produção de penicilina acelera progressivamente. Após $t = 0.5$, quando a concentração celular se aproxima do equílibrio, a taxa de produção de penicilina se torna aproximadamente constante, resultando no comportamento linear observado.

A representação no plano de fase g x t, apresentado na Figura (3) fornece algumas explicações adicionais sobre a dinâmica do sistema acoplado. A trajetória inicia na origem modificada $(0.03, 0)$ e evolui em direção ao ponto $(0.940, 1.184)$, descrevendo uma curva monotônica que reflete o acoplamento unidirecional entre as variáveis.

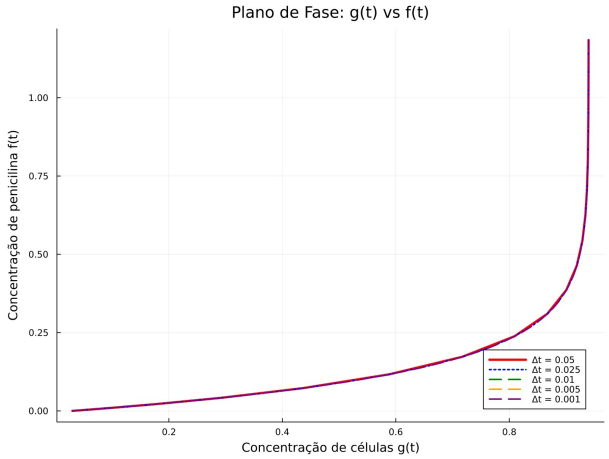

Figura 3: Plano de fase g x f

A concavidade da curva no plano de fase mostra que a produção inicial de penicilin aé relativamente baixa devido à pequena população celular, mas acelera significativamente durante a fase de crescimento exponencial das células. A transição para um regime de crescimento linear de $f(t)$ corresponde à região onde a trajetória se aproxima de uma reta no plano de fase.

A análise de convergência, conduzida variando o passo de integração de $\Delta t = 0.001$ até $\Delta t = 0.05$ é exibida nas Tabelas (3), (4), (5), (6) e (7), no Apêndice A. As Tabelas (1) e (2) exibem os resultados de forma resumida, resultados estes que demonstram convergência consistente para ambas as variáveis.

Δt	g(1)
0.050	0.93968222
0.025	0.93968341
0.010	0.93968347
0.005	0.93968347
0.001	0.93968347

Tabela 1: Concentração de células $g(t)$ em $t = 1$ com variação de Δt

Δt	f(1)
0.050	1.18425817
0.025	1.18443113
0.010	1.18444499
0.005	1.18444537
0.001	1.18444540

Tabela 2: Concentração de células $f(t)$ em $t = 1$ com variação de Δt

Os erros relativos entre $\Delta t = 0.05$ e $\Delta t = 0.001$ são da ordem de $1.3 \cdot 10^{-6}$ para $g(t)$ e de $1.6 \cdot 10^{-4}$ para $f(t)$, confirmando a alta precisão do método implementado. A convergência observada é de consistente com a ordem teóriaca $O(h^4)$ do método RK4, o que valida a implementação computacional.

A variação do coeficiente linear a_1 produz impactos significativos na dinâmica de ambas as variáveis, o que pode ser visto nas Figuras (4) e (5). A análise de ambas revela que o aumento em a_1 eleva diretamente a capacidade de suporte para a concentração de células. Para um aumento de 10% de a_1 em $g(t)$, observa-se que $g(1) = 1.040$, enquanto para uma diminuição de 10% do coeficiente, obtem-se $g(1) = 0.846$. A relação é aproximadamente linear, conforme espera-se na análise de equilíbrio $g_{eq} = \frac{a_1}{a_2}$.

Da mesma forma, na concentração de penicilina, o efeito é amplificado devido ao acoplamento. Para um aumento de 10% em a_1, tem-se $f(1) = 1.311$, representando um aumento de 10.7% em relação ao caso base. Paralelamente, para uma redução de 10% em a_1, tem-se $f(1) = 1.047$, com uma redução de 11.6%.

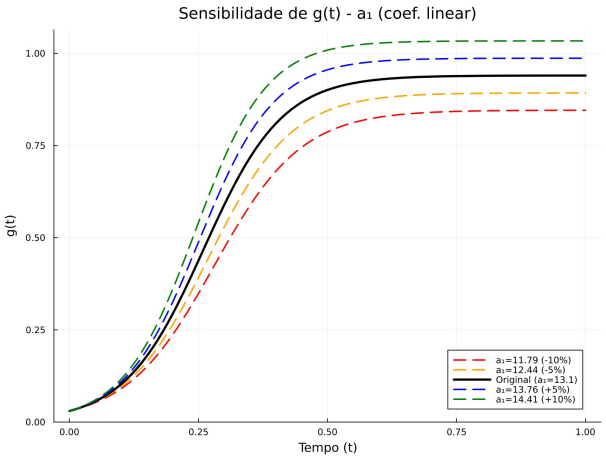

Figura 4: Sensibilidade de $g(t)$ ao coeficiente linear a_1

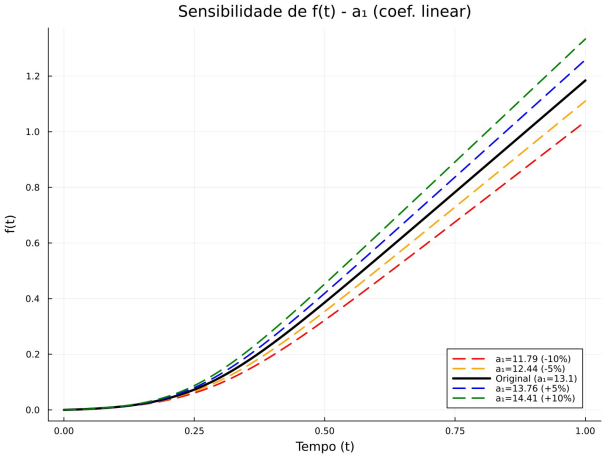

Figura 5: Sensibilidade de $f(t)$ ao coeficiente linear a_1

Já em relação ao coeficiente quadrático a_2, as Figuras (6) e (7) mostram o comportamento, que é inverso ao observado para a_1. Em relação a concentração de células, as reduções em a_2 aumentam a capacidade de suporte. Para 10% de redução, $g(1) = 1.044$, enquanto para 10% de aumento, $g(1) = 0.855$.

Da mesma maneira ao coeficiente a_1, a concentração de penicilina segue a mesma tendência da concentração de células devido ao acoplamento, então com variações de $\pm 10\%$ em a_2, há mudanças de aproximadamente $\pm 11\%$ em $f(1)$.

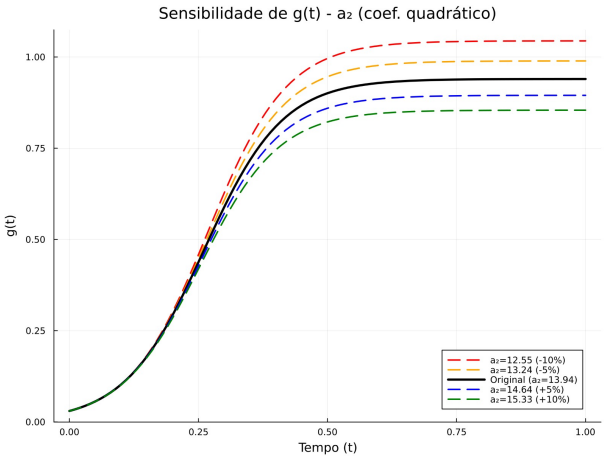

Figura 6: Sensibilidade de $g(t)$ ao coeficiente quadrático a_2

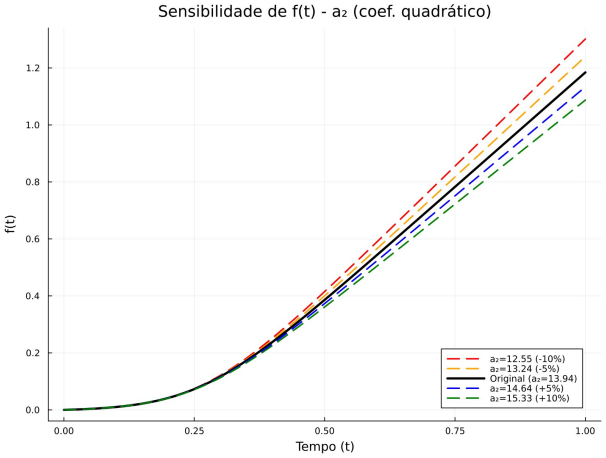

Figura 7: Sensibilidade de $f(t)$ ao coeficiente quadrático a_2

Por fim, nas Figuras (8) e (9) tem-se a variação do coeficiente de produção b_1. As curvas de $g(t)$ são quase idênticas para todas as variações de b_1 testadas. Esta situação ocorre pois b_1 não aparece na equação diferencial que rege $g(t)$, confirmando que a produção de penicilina não afeta retroativamente o crescimento celular no modelo proposto.

Já na concentração de penicilina, vê-se um efeito diretamente proporcional, o que é esperado pela relação $\frac{df}{dt} = b_1 g$. Variações de $\pm 10\%$ em b_1 resultam em mudanças correspondentes de $\pm 10\%$ em $f(1)$, demonstrando linearidade perfeita.

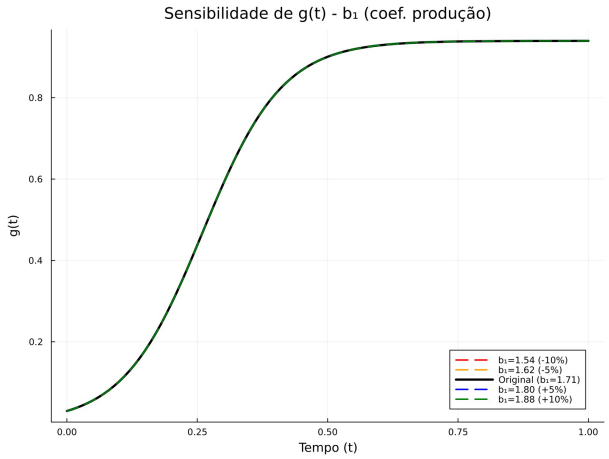

Figura 8: Sensibilidade de $g(t)$ ao coeficiente linear b_1

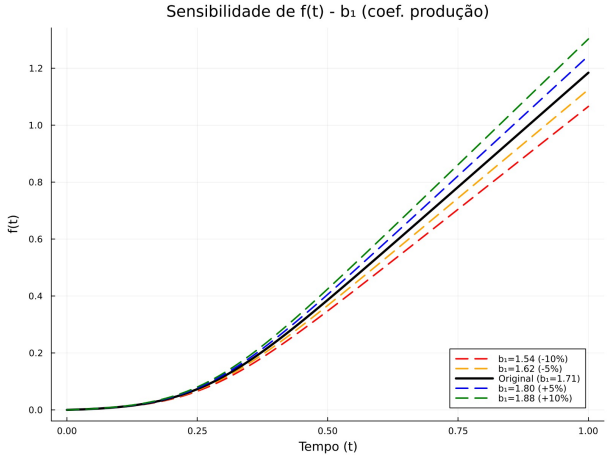

Figura 9: Sensibilidade de $f(t)$ ao coeficiente linear b_1

A implementação do método RK4 demonstrou excelente estabilidade numérica e convergência consistente com a teoria, o que valida a abordagem computacional aqui adotada. Os resultados revelaram um comportamento dinâmico distinto para as duas variáveis do sistema, sendo elas: crescimento logístico saturado para a concentração celular e crescimento linear para a produção da penicilina após o regime transiente inicial.

A análise de sensibilidade quantificou a influência de cada parâmetro cinético, evidenciando que as variações de $\pm 10\%$ nos coeficientes resultam em desvios controláveis de aproximadamente 11% na produção final de penicilina. Particularmente relevante é a constatação da não-influência do coeficiente de produção b_1 na dinâmica celular, o que confirma o desacoplamento unidirecional do modelo proposto.

5 Conclusão

O presente trabalho demonstrou com sucesso a aplicação do método de Runge-Kutta de quarta ordem na resolução numérica de um sistema de equações diferenciais ordinárias que modela a dinâmica de um reator de batelada para produção de penicilina. A implementação computacional em linguagem Julia comprovou-se eficaz, proporcionando resultados numericamente consistentes e computacionalmente eficientes.

A análise de convergência confirmou empiricamente a ordem teórica $O(h^4)$ do método RK4, com erros relativos da ordem de $1.3 \cdot 10^{-6}$ para a concentração celular e $1.6 \cdot 10^{-4}$ para a concentração de penicilina quando comparados entre $\Delta t = 0.001$ e $\Delta t = 0.05$. Esta verificação valida a implementação computacional e a adequação do método para o problema.

A estabilidade numérica observada e a complexidade computacional linear $O(N)$ validam tanto a implementação quanto a adequação do método para o problema estudado.

Os resultados revelaram comportamentos fisicamente coerentes: crescimento logístico saturado para a concentração celular, convergindo para $g_{eq} = 0.940$, possuindo uma concordância excelente com a previsão teórica $\frac{a1}{a2}$, e crescimento linear para a penicilina após regime transiente. O desacoplamento unidirecional do modelo foi confirmado pela independência da dinâmica celular em relação ao coeficiente b_1.

A análise paramétrica quantificou que variações de $\pm 10\%$ nos coeficientes a_1 e a_2 resultam em mudanças de aproximadamente $\pm 11\%$ na produção final de penicilina. O coeficiente b_1 apresentou relação diretamente proporcional com a produção, sem afetar o crescimento celular.

Os resultados indicam que $t = 0.7$ representa um marco adequado entre crescimento celular (com 95% do valor final) e produção de penicilina para definição de tempos de ciclo. A robustez do processo frente a variações paramétricas ($\pm 11\%$ de variação para $\pm 10\%$ nos coeficientes) e a independência entre b_1 e crescimento celular sugerem estratégias de otimização que podem melhorar a eficiência produtiva sem comprometer a fase de crescimento.

As limitações incluem a hipótese de sistema não-rígido e parâmetros constantes, que podem não se aplicar a modelos mais complexos.

Trabalhos futuros podem incluir: implementação de métodos implícitos para sistemas rígidos (BDF, Rosenbrock); incorporação de efeitos dinâmicos de temperatura, pH e concentração de substrato; desenvolvimento de algoritmos de otimização automática para maximização da produtividade; análise de incertezas probabilísticas utilizando métodos de Monte Carlo; e extensão para reatores contínuos e fed-batch com controle em tempo real.

A estrutura modular desenvolvida facilita essas extensões, posicionando este trabalho como base sólida para investigações mais avançadas em modelagem e otimização de bioreatores.

Em síntese, o trabalho demonstrou a aplicabilidade de métodos numéricos clássicos na solução de problemas de engenharia química, fornecendo validação teórica e conclusões operacionais relevantes para otimização de processos biotecnológicos.

Referências Bibliográficas

[1] J. Bezanson, A. Edelman, S. Karpinski, and V. B. Shah, "Julia: A fresh approach to numerical computing," *SIAM Review*, vol. 59, no. 1, pp. 65–98, 2017.

[2] S. C. Chapra and R. P. Canale, *Métodos Numéricos para Engenharia*. Porto Alegre: AMGH Editora, 7 ed., 2016.

[3] "Para que serve a linguagem julia?." `https://www.datacamp.com/pt/blog/what-is-julia-used-for`, 2024. Accessed: 2025-09-19.

[4] "Julia vs python - which one is best fit for your business?." `https://eluminoustechnologies.com/blog/julia-vs-python/`, 2025. Accessed: 2025-09-19.

[5] J. C. Butcher, *Numerical Methods for Ordinary Differential Equations*. John Wiley & Sons, 3 ed., 2016.

[6] E. Hairer, S. P. Nørsett, and G. Wanner, *Solving Ordinary Differential Equations I: Nonstiff Problems*. Springer, 1993. Classical reference on RK methods and stability analysis.

Apêndice A: Tabelas de análise de convergência

t	g(t)	f(t)
0.0000	0.03000000	0.00000000
0.0500	0.05609750	0.00357042
0.1000	0.10234364	0.01016446
0.1500	0.17899199	0.02194005
0.2000	0.29296190	0.04184897
0.2500	0.43774353	0.07293434
0.3000	0.58892539	0.11689063
0.3500	0.71767463	0.17298491
0.4000	0.80961352	0.23854627
0.4500	0.86732904	0.31044706
0.5000	0.90068113	0.38616634
0.5500	0.91903849	0.46403921
0.6000	0.92887253	0.54308151
0.6500	0.93406426	0.62274571
0.7000	0.93678404	0.70273696
0.7500	0.93820306	0.78289921
0.8000	0.93894186	0.86315057
0.8500	0.93932608	0.94344830
0.9000	0.93952579	1.02377014
0.9500	0.93962956	1.10410452
1.0000	0.93968347	1.18444540

Tabela 3: Resultados para $\Delta t = 0.0010$, 1000 passos, $g(0) = 0.03$, $f(0) = 0.0$ e coeficientes $a_1 = 13.1, a_2 = 13.94$ e $b_1 = 1.71$

t	g(t)	f(t)
0.0000	0.03000000	0.00000000
0.0500	0.05609749	0.00357042
0.1000	0.10234363	0.01016446
0.1500	0.17899197	0.02194004
0.2000	0.29296186	0.04184897
0.2500	0.43774348	0.07293433
0.3000	0.58892534	0.11689061
0.3500	0.71767459	0.17298489
0.4000	0.80961348	0.23854625
0.4500	0.86732902	0.31044704
0.5000	0.90068112	0.38616632
0.5500	0.91903848	0.46403919
0.6000	0.92887252	0.54308148
0.6500	0.93406426	0.62274568
0.7000	0.93678404	0.70273694
0.7500	0.93820306	0.78289918
0.8000	0.93894186	0.86315054
0.8500	0.93932608	0.94344828
0.9000	0.93952579	1.02377012
0.9500	0.93962956	1.10410449
1.0000	0.93968347	1.18444537

Tabela 4: Resultados para $\Delta t = 0.0050$, 200 passos, $g(0) = 0.03$, $f(0) = 0.0$ e coeficientes $a_1 = 13.1, a_2 = 13.94$ e $b_1 = 1.71$

t	g(t)	f(t)
0.0000	0.03000000	0.00000000
0.0500	0.05609743	0.00357041
0.1000	0.10234345	0.01016444
0.1500	0.17899159	0.02193999
0.2000	0.29296126	0.04184887
0.2500	0.43774275	0.07293418
0.3000	0.58892463	0.11689040
0.3500	0.71767399	0.17298462
0.4000	0.80961303	0.23854594
0.4500	0.86732868	0.31044670
0.5000	0.90068088	0.38616596
0.5500	0.91903832	0.46403882
0.6000	0.92887242	0.54308111
0.6500	0.93406420	0.62274530
0.7000	0.93678400	0.70273656
0.7500	0.93820304	0.78289880
0.8000	0.93894185	0.86315016
0.8500	0.93932608	0.94344789
0.9000	0.93952579	1.02376973
0.9500	0.93962956	1.10410410
1.0000	0.93968347	1.18444499

Tabela 5: Resultados para $\Delta t = 0.0100$, 100 passos, $g(0) = 0.03$, $f(0) = 0.0$ e coeficientes $a_1 = 13.1, a_2 = 13.94$ e $b_1 = 1.71$

t	g(t)	f(t)
0.0000	0.03000000	0.00000000
0.0500	0.05609544	0.00357016
0.1000	0.10233712	0.01016361
0.1500	0.17897820	0.02193813
0.2000	0.29293988	0.04184544
0.2500	0.43771647	0.07292873
0.3000	0.58889898	0.11688271
0.3500	0.71765233	0.17297497
0.4000	0.80959576	0.23853493
0.4500	0.86731545	0.31043480
0.5000	0.90067128	0.38615344
0.5500	0.91903175	0.46402585
0.6000	0.92886814	0.54306782
0.6500	0.93406151	0.62273180
0.7000	0.93678236	0.70272291
0.7500	0.93820206	0.78288507
0.8000	0.93894128	0.86313637
0.8500	0.93932575	0.94343407
0.9000	0.93952560	1.02375590
0.9500	0.93962945	1.10409026
1.0000	0.93968341	1.18443113

Tabela 6: Resultados para $\Delta t = 0.0250$, 40 passos, $g(0) = 0.03$, $f(0) = 0.0$ e coeficientes $a_1 = 13.1, a_2 = 13.94$ e $b_1 = 1.71$

t	g(t)	f(t)
0.0000	0.03000000	0.00000000
0.0500	0.05607190	0.00356722
0.1000	0.10226165	0.01015390
0.1500	0.17881616	0.02191602
0.2000	0.29267641	0.04180476
0.2500	0.43738777	0.07286383
0.3000	0.58857636	0.11679015
0.3500	0.71737472	0.17285811
0.4000	0.80935935	0.23840190
0.4500	0.86711749	0.31029131
0.5000	0.90051646	0.38600195
0.5500	0.91891989	0.46386783
0.6000	0.92879248	0.54290474
0.6500	0.93401279	0.62256509
0.7000	0.93675209	0.70255376
0.7500	0.93818375	0.78271435
0.8000	0.93893042	0.86296468
0.8500	0.93931940	0.94326179
0.9000	0.93952194	1.02358326
0.9500	0.93962736	1.10391742
1.0000	0.93968222	1.18425817

Tabela 7: Resultados para $\Delta t = 0.0500$, 20 passos, $g(0) = 0.03$, $f(0) = 0.0$ e coeficientes $a_1 = 13.1, a_2 = 13.94$ e $b_1 = 1.71$

Apêndice B: Código fonte

```julia
using Plots, Printf, Statistics, Dates

# Estrutura para armazenar parâmetros do modelo
struct ReactorParams
    a1::Float64  # Coeficiente linear (13.1)
    a2::Float64  # Coeficiente quadrático (13.94)
    b1::Float64  # Coeficiente produção penicilina (1.71)
end

# Parâmetros originais conforme especificação do problema
PARAMS_ORIGINAL = ReactorParams(13.1, 13.94, 1.71)

function dg_dt(g, f, t, params::ReactorParams)
    return params.a1 * g - params.a2 * g^2
end

function df_dt(g, f, t, params::ReactorParams)
    return params.b1 * g
end

# Implementação do método Runge-Kutta clássico de 4 ordem
function runge_kutta_4(g0, f0, dt, t_max, params::ReactorParams)
    """
    Implementa o método Runge-Kutta clássico de 4ł ordem para sistema de EDOs

    Parâmetros:
    - g0, f0: condições iniciais
    - dt: incremento de tempo
    - t_max: tempo máximo de simulação
    - params: parâmetros do modelo

    Retorna:
    - t, g, f: vetores com soluções numéricas
    """

    # Número de passos
    n_steps = Int(ceil(t_max / dt))

    # Inicializar vetores de resultados
    t = Vector{Float64}(undef, n_steps + 1)
    g = Vector{Float64}(undef, n_steps + 1)
    f = Vector{Float64}(undef, n_steps + 1)

    # Condições iniciais
    t[1] = 0.0
    g[1] = g0
    f[1] = f0

    # Loop principal do método RK4
```

```
    for i in 1:n_steps
        # Valores atuais
        t_n = t[i]
        g_n = g[i]
        f_n = f[i]

        # Cálculo dos coeficientes k para g
        k1_g = dg_dt(g_n, f_n, t_n, params)
        k2_g = dg_dt(g_n + k1_g * dt/2, f_n, t_n + dt/2, params)
        k3_g = dg_dt(g_n + k2_g * dt/2, f_n, t_n + dt/2, params)
        k4_g = dg_dt(g_n + k3_g * dt, f_n, t_n + dt, params)

        # Cálculo dos coeficientes k para f
        k1_f = df_dt(g_n, f_n, t_n, params)
        k2_f = df_dt(g_n + k1_g * dt/2, f_n, t_n + dt/2, params)
        k3_f = df_dt(g_n + k2_g * dt/2, f_n, t_n + dt/2, params)
        k4_f = df_dt(g_n + k3_g * dt, f_n, t_n + dt, params)

        # Aplicação da fórmula RK4
        g[i+1] = g_n + (dt/6) * (k1_g + 2*k2_g + 2*k3_g + k4_g)
        f[i+1] = f_n + (dt/6) * (k1_f + 2*k2_f + 2*k3_f + k4_f)
        t[i+1] = t_n + dt
    end

    return t, g, f
end

# Estrutura para armazenar resultados
struct SimulationResult
    t::Vector{Float64}
    g::Vector{Float64}
    f::Vector{Float64}
    params::ReactorParams
    dt::Float64
    label::String
end

# Função para salvar resultados em formato tabular
function salvar_resultados_tabela(t, g, f, dt, filename)
    """
    Salva os resultados em formato de tabela para o relatório
    """
    open(filename, "w") do file
        write(file, "="^70 * "\n\n")

        write(file, "PARÂMETROS DA SIMULAÇÃO:\n")
        write(file, @sprintf(" Incremento de tempo (t): %.4f\n", dt))
        write(file, @sprintf(" Tempo máximo (t_max): %.1f\n", t[end]))
        write(file, @sprintf(" Número de passos: %d\n", length(t)-1))
        write(file, " Condições iniciais: g(0) = 0.03, f(0) = 0.0\n")
        write(file, " Coeficientes: a = 13.1, a = 13.94, b = 1.71\n\n")

```

```
        write(file, "RESULTADOS NUMÉRICOS:\n")
        write(file, @sprintf("%-12s %-15s %-15s\n", "t", "g(t)", "f(t)"))
        write(file, "-"^45 * "\n")

        # Mostrar todos os pontos para dt principal, ou pontos selecionados para outros
        step = dt == 0.05 ? 1 : max(1, Int(round(length(t)/20)))

        for i in 1:step:length(t)
            write(file, @sprintf("%-12.4f %-15.8f %-15.8f\n", t[i], g[i], f[i]))
        end

        # Sempre mostrar o último ponto
        if step > 1
            write(file, @sprintf("%-12.4f %-15.8f %-15.8f\n", t[end], g[end], f[end]))
        end

                write(file, "\n")
        write(file, @sprintf("VALORES FINAIS (t = %.1f):\n", t[end]))
        write(file, @sprintf(" g(%.1f) = %.8f\n", t[end], g[end]))
        write(file, @sprintf(" f(%.1f) = %.8f\n", t[end], f[end]))
    end
    println("Resultados salvos em: $filename")
end

# Função para análise de convergência conforme especificação
function analise_convergencia_completa(g0, f0, t_max)
    """
    Realiza análise de convergência variando t conforme requisito 2
    Incremento inicial: t = 0.05, depois testa convergência
    """
    println("\n" * "="^60)
    println("ANÁLISE DE CONVERGÊNCIA NUMÉRICA")
    println("="^60)

    # Valores de t para teste de convergência (começando com 0.05 conforme especificação)
    dt_values = [0.05, 0.025, 0.01, 0.005, 0.001]
    resultados = SimulationResult[]

    println("Testando convergência com diferentes valores de t:")
    println("-"^60)

    # Solução de referência com t muito pequeno
    println("Calculando solução de referência (t = 0.0005)...")
    t_ref, g_ref, f_ref = runge_kutta_4(g0, f0, 0.0005, t_max, PARAMS_ORIGINAL)

    # Arquivo para resultados de convergência
    open("convergencia_numerica.txt", "w") do file
        write(file, "ANÁLISE DE CONVERGÊNCIA NUMÉRICA\n")
        write(file, "="^70 * "\n\n")

        write(file, @sprintf("%-8s %-12s %-12s %-12s %-12s %-10s\n",
              "t", "g(1)", "f(1)", "Erro g", "Erro f", "Ordem"))
```

```
        write(file, "-"^75 * "\n")

        erro_anterior_g = 0.0

        for (idx, dt) in enumerate(dt_values)
            println(@sprintf("Simulando com t = %.4f...", dt))

            t, g, f = runge_kutta_4(g0, f0, dt, t_max, PARAMS_ORIGINAL)

            # Calcular erros em relação à solução de referência
            erro_g = abs(g[end] - g_ref[end])
            erro_f = abs(f[end] - f_ref[end])

            # Calcular ordem de convergência
            ordem = idx > 1 && erro_anterior_g > 0 ? log2(erro_anterior_g / erro_g) : 0.0

            write(file, @sprintf("%-8.4f %-12.8f %-12.8f %-12.2e %-12.2e %-10.2f\n",
                  dt, g[end], f[end], erro_g, erro_f, ordem))

            println(@sprintf("  t = %.4f: g(1) = %.8f, f(1) = %.8f, Erro_g = %.2e",
                    dt, g[end], f[end], erro_g))

            # Salvar resultados tabulares para cada t
            filename = @sprintf("resultados_dt_%.4f.txt", dt)
            salvar_resultados_tabela(t, g, f, dt, filename)

            # Armazenar para plotagem
            resultado = SimulationResult(t, g, f, PARAMS_ORIGINAL, dt, "t = $dt")
            push!(resultados, resultado)

            erro_anterior_g = erro_g
        end

        write(file, "\nNOTAS:\n")
        write(file, " Solução de referência calculada com t = 0.0005\n")
        write(file, " Ordem de convergência teórica do RK4: 4\n")
        write(file, " Erro = |valor_calculado - valor_referência|\n")
    end

    return resultados
end

# Função para plotar resultados principais (conforme requisito 3)
function plotar_resultados_principais(resultados)
    """
    Gera os gráficos principais dos resultados conforme especificação
    """
    println("\nGerando gráficos dos resultados...")

    # Configurar tema dos gráficos
    gr(size=(800, 600), dpi=300)

```

```
    # 1. Gráfico da evolução temporal de g(t)
    p1 = plot(title="Evolução da Concentração de Células g(t)",
              xlabel="Tempo adimensional (t)",
              ylabel="Concentração adimensional g(t)",
              legend=:bottomright,
              grid=true,
              linewidth=2)

    # 2. Gráfico da evolução temporal de f(t)
    p2 = plot(title="Evolução da Concentração de Penicilina f(t)",
              xlabel="Tempo adimensional (t)",
              ylabel="Concentração adimensional f(t)",
              legend=:bottomright,
              grid=true,
              linewidth=2)

    # 3. Plano de fase g vs f
    p3 = plot(title="Plano de Fase: g(t) vs f(t)",
              xlabel="Concentração de células g(t)",
              ylabel="Concentração de penicilina f(t)",
              legend=:bottomright,
              grid=true,
              linewidth=2)

    # Cores para diferentes t
    cores = [:red, :blue, :green, :orange, :purple]

    for (i, res) in enumerate(resultados)
        cor = cores[min(i, length(cores))]
        estilo = res.dt == 0.05 ? :solid : (res.dt <= 0.01 ? :dash : :dot)
        largura = res.dt == 0.05 ? 3 : 2

        plot!(p1, res.t, res.g,
              label=res.label,
              color=cor,
              linestyle=estilo,
              linewidth=largura)

        plot!(p2, res.t, res.f,
              label=res.label,
              color=cor,
              linestyle=estilo,
              linewidth=largura)

        plot!(p3, res.g, res.f,
              label=res.label,
              color=cor,
              linestyle=estilo,
              linewidth=largura)
    end

    # Salvar gráficos individuais
```

```
    savefig(p1, "evolucao_g_tempo.png")
    savefig(p2, "evolucao_f_tempo.png")
    savefig(p3, "plano_fase.png")

    # Gráfico combinado
    p_combined = plot(p1, p2, layout=(2,1), size=(800, 1000))
    savefig(p_combined, "resultados_principais.png")

    println("Gráficos principais salvos:")
    println(" evolucao_g_tempo.png")
    println(" evolucao_f_tempo.png")
    println(" plano_fase.png")
    println(" resultados_principais.png")

    return p1, p2, p3
end

# Função para análise de sensibilidade (requisito 4)
function analise_sensibilidade_coeficientes(g0, f0, t_max, dt_base)
    """
    Análise de sensibilidade quando da variação de coeficientes das equações
    Conforme requisito 4 da especificação
    """
    println("\n" * "="^60)
    println("ANÁLISE DE SENSIBILIDADE DOS COEFICIENTES")
    println("="^60)

    # Variações percentuais dos coeficientes
    variacoes = [-10, -5, 0, 5, 10]  # em %
    resultados_sensibilidade = Dict{String, Vector{SimulationResult}}()

    # Coeficientes originais
    a1_orig, a2_orig, b1_orig = 13.1, 13.94, 1.71

    # Análise do coeficiente a = 13.1
    println("Analisando sensibilidade do coeficiente a = 13.1...")
    resultados_a1 = SimulationResult[]

    for var in variacoes
        fator = 1 + var/100
        a1_novo = a1_orig * fator
        params_mod = ReactorParams(a1_novo, a2_orig, b1_orig)

        t, g, f = runge_kutta_4(g0, f0, dt_base, t_max, params_mod)
        label = var == 0 ? "Original (a=13.1)" : @sprintf("a=%.2f (%+d%%)", a1_novo, var)
        resultado = SimulationResult(t, g, f, params_mod, dt_base, label)
        push!(resultados_a1, resultado)
    end
    resultados_sensibilidade["a1"] = resultados_a1

    # Análise do coeficiente a = 13.94
    println("Analisando sensibilidade do coeficiente a = 13.94...")
```

```
    resultados_a2 = SimulationResult[]

    for var in variacoes
        fator = 1 + var/100
        a2_novo = a2_orig * fator
        params_mod = ReactorParams(a1_orig, a2_novo, b1_orig)

        t, g, f = runge_kutta_4(g0, f0, dt_base, t_max, params_mod)
        label = var == 0 ? "Original (a=13.94)" : @sprintf("a=%.2f (%+d%%)", a2_novo, var)
        resultado = SimulationResult(t, g, f, params_mod, dt_base, label)
        push!(resultados_a2, resultado)
    end
    resultados_sensibilidade["a2"] = resultados_a2

    # Análise do coeficiente b = 1.71
    println("Analisando sensibilidade do coeficiente b = 1.71...")
    resultados_b1 = SimulationResult[]

    for var in variacoes
        fator = 1 + var/100
        b1_novo = b1_orig * fator
        params_mod = ReactorParams(a1_orig, a2_orig, b1_novo)

        t, g, f = runge_kutta_4(g0, f0, dt_base, t_max, params_mod)
        label = var == 0 ? "Original (b=1.71)" : @sprintf("b=%.2f (%+d%%)", b1_novo, var)
        resultado = SimulationResult(t, g, f, params_mod, dt_base, label)
        push!(resultados_b1, resultado)
    end
    resultados_sensibilidade["b1"] = resultados_b1

    return resultados_sensibilidade
end

# Função para plotar análise de sensibilidade
function plotar_sensibilidade(resultados_sensibilidade)
    """
    Gera gráficos da análise de sensibilidade
    """
    println("Gerando gráficos de análise de sensibilidade...")

    cores = [:red, :orange, :black, :blue, :green]

    # Gráficos para cada coeficiente
    plots_g = []
    plots_f = []

    coef_nomes = ["a (coef. linear)", "a (coef. quadrático)", "b (coef. produção)"]
    coef_keys = ["a1", "a2", "b1"]

    for (idx, (key, nome)) in enumerate(zip(coef_keys, coef_nomes))
        resultados = resultados_sensibilidade[key]

```

```
        # Gráfico g(t)
        p_g = plot(title="Sensibilidade de g(t) - $nome",
                   xlabel="Tempo (t)", ylabel="g(t)",
                   legend=:bottomright, grid=true)

        # Gráfico f(t)
        p_f = plot(title="Sensibilidade de f(t) - $nome",
                   xlabel="Tempo (t)", ylabel="f(t)",
                   legend=:bottomright, grid=true)

        for (i, res) in enumerate(resultados)
            cor = cores[i]
            estilo = occursin("Original", res.label) ? :solid : :dash
            largura = occursin("Original", res.label) ? 3 : 2

            plot!(p_g, res.t, res.g,
                  label=res.label, color=cor,
                  linestyle=estilo, linewidth=largura)

            plot!(p_f, res.t, res.f,
                  label=res.label, color=cor,
                  linestyle=estilo, linewidth=largura)
        end

        push!(plots_g, p_g)
        push!(plots_f, p_f)

        # Salvar gráficos individuais
        savefig(p_g, "sensibilidade_$(key)_g.png")
        savefig(p_f, "sensibilidade_$(key)_f.png")
    end

    # Gráficos combinados
    p_combined_g = plot(plots_g..., layout=(3,1), size=(800, 1200))
    p_combined_f = plot(plots_f..., layout=(3,1), size=(800, 1200))

    savefig(p_combined_g, "sensibilidade_completa_g.png")
    savefig(p_combined_f, "sensibilidade_completa_f.png")

    println("Gráficos de sensibilidade salvos:")
    for key in coef_keys
        println(" sensibilidade_$(key)_g.png")
        println(" sensibilidade_$(key)_f.png")
    end
    println(" sensibilidade_completa_g.png")
    println(" sensibilidade_completa_f.png")

    return plots_g, plots_f
end

# Função para análise quantitativa da sensibilidade
function relatorio_sensibilidade(resultados_sensibilidade)
```

```
    """
    Gera relatório detalhado da análise de sensibilidade
    """
    open("analise_sensibilidade_completa.txt", "w") do file
        write(file, "ANÁLISE DE SENSIBILIDADE DOS COEFICIENTES\n")
        write(file, "="^70 * "\n\n")

        write(file, "VALORES ORIGINAIS:\n")
        write(file, " a = 13.1 (coeficiente linear)\n")
        write(file, " a = 13.94 (coeficiente quadrático)\n")
        write(file, " b = 1.71 (coeficiente de produção)\n\n")

        # Valor original de referência
        res_original = findfirst(r -> occursin("Original", r.label),
                                 resultados_sensibilidade["a1"])
        g_ref = resultados_sensibilidade["a1"][res_original].g[end]
        f_ref = resultados_sensibilidade["a1"][res_original].f[end]

        write(file, @sprintf("VALORES DE REFERÊNCIA (t=1):\n"))
        write(file, @sprintf(" g(1) = %.8f\n", g_ref))
        write(file, @sprintf(" f(1) = %.8f\n\n", f_ref))

        # Análise para cada coeficiente
        coef_nomes = Dict("a1" => "a (coeficiente linear)",
                          "a2" => "a (coeficiente quadrático)",
                          "b1" => "b (coeficiente de produção)")

        for (key, nome) in coef_nomes
            write(file, "SENSIBILIDADE DO $nome:\n")
            write(file, "-"^50 * "\n")
            write(file, @sprintf("%-15s %-12s %-12s %-10s %-10s\n",
                  "Variação", "g(1)", "f(1)", "g(%)", "f(%)"))
            write(file, "-"^65 * "\n")

            for res in resultados_sensibilidade[key]
                if !occursin("Original", res.label)
                    delta_g_pct = ((res.g[end] - g_ref) / g_ref) * 100
                    delta_f_pct = ((res.f[end] - f_ref) / f_ref) * 100

                    # Extrair variação do label
                    if occursin("(+", res.label)
                        variacao = split(split(res.label, "(+")[2], "%")[1] * "%"
                        variacao = "+" * variacao
                    elseif occursin("(-", res.label)
                        variacao = split(split(res.label, "(-")[2], "%")[1] * "%"
                        variacao = "-" * variacao
                    else
                        variacao = "Original"
                    end

                    write(file, @sprintf("%-15s %-12.8f %-12.8f %-10.2f %-10.2f\n",
                          variacao, res.g[end], res.f[end], delta_g_pct, delta_f_pct))
```

```
                end
            end
            write(file, "\n")
        end

        write(file, "INTERPRETAÇÃO DOS RESULTADOS:\n")
        write(file, " g(%) e f(%): variação percentual em relação ao caso original\n")
        write(file, " Valores positivos indicam aumento da concentração\n")
        write(file, " Valores negativos indicam diminuição da concentração\n")
    end

    println("Relatório de sensibilidade salvo: analise_sensibilidade_completa.txt")
end

# FUNÇÃO PRINCIPAL
function main()
    """
    Função principal que executa toda a simulação conforme especificação
    """
    println("Iniciando simulação completa do reator de batelada...")
    println("Linguagem: Julia")
    println("Data: $(Dates.now())")

    # PARÂMETROS DO PROBLEMA (conforme especificação)
    g0 = 0.03     # g(0) = 0.03
    f0 = 0.0      # f(0) = 0.0
    t_max = 1.0   # 0  t  1
    dt_inicial = 0.05  # t inicial = 0.05 (conforme requisito 2)

    # 1. ANÁLISE DE CONVERGÊNCIA (requisitos 2 e 3)
    println("\n1 EXECUTANDO ANÁLISE DE CONVERGÊNCIA...")
    resultados_convergencia = analise_convergencia_completa(g0, f0, t_max)

    # 2. PLOTAR RESULTADOS PRINCIPAIS (requisito 3)
    println("\n2 GERANDO GRÁFICOS DOS RESULTADOS...")
    plotar_resultados_principais(resultados_convergencia)

    # 3. ANÁLISE DE SENSIBILIDADE (requisito 4)
    println("\n3 EXECUTANDO ANÁLISE DE SENSIBILIDADE...")
    resultados_sensibilidade = analise_sensibilidade_coeficientes(g0, f0, t_max, 0.01)

    # 4. PLOTAR SENSIBILIDADE
    println("\n4 GERANDO GRÁFICOS DE SENSIBILIDADE...")
    plotar_sensibilidade(resultados_sensibilidade)

    # 5. RELATÓRIO DE SENSIBILIDADE
    println("\n5GERANDO RELATÓRIO DE SENSIBILIDADE...")
    relatorio_sensibilidade(resultados_sensibilidade)

    # 6. RESUMO FINAL
    println("\n" * "="^80)
    println("SIMULAÇÃO CONCLUÍDA COM SUCESSO!")
```

```
    println("="^80)
    println("\n ARQUIVOS GERADOS:")
    println("\n TABELAS DE RESULTADOS:")
    println(" convergencia_numerica.txt - Análise de convergência")
    println(" resultados_dt_0.0500.txt - Resultados com t=0.05 (principal)")
    println(" resultados_dt_*.txt - Resultados para outros valores de t")
    println(" analise_sensibilidade_completa.txt - Análise detalhada")

    println("\n GRÁFICOS PRINCIPAIS:")
    println(" evolucao_g_tempo.png - Evolução de g(t)")
    println(" evolucao_f_tempo.png - Evolução de f(t)")
    println(" plano_fase.png - Plano de fase g vs f")
    println(" resultados_principais.png - Gráficos combinados")

    println("\nGRÁFICOS DE SENSIBILIDADE:")
    println(" sensibilidade_a1_g.png, sensibilidade_a1_f.png")
    println(" sensibilidade_a2_g.png, sensibilidade_a2_f.png")
    println(" sensibilidade_b1_g.png, sensibilidade_b1_f.png")
    println(" sensibilidade_completa_g.png, sensibilidade_completa_f.png")

    # Resultados finais com t = 0.05
    resultado_principal = resultados_convergencia[1]  # t = 0.05
    println("\n RESULTADOS FINAIS (t = 0.05):")
    println(@sprintf(" g(1) = %.8f", resultado_principal.g[end]))
    println(@sprintf(" f(1) = %.8f", resultado_principal.f[end]))

end

main()
```