Farmacocinética - Valaciclovir

Mestrado Integrado em Engenharia Informática e Computação

Métodos Numéricos

Grupo 29:

Miguel Norberto Costa Freitas - 201906159

Vasco Marques Lopes Teixeira - 201802112

Tiago Caldas da Silva - 201906045

Faculdade de Engenharia da Universidade do Porto

20 de Outubro de 2019

Índice

Introdução

O projeto que desenvolvemos tem como intuito prever o comportamento de um medicamento quando administrado e a sua concentração plasmática no corpo da pessoa a quem foi administrada. Usamos a concentração plasmática por não ser clinicamente

possível medir a concentração da substância nos órgãos e tecidos e esta tem uma relação linear com a concentração do fármaco no local de ação.

O medicamento em questão é o valaciclovir, um medicamento antiviral, e pretendemos fazer uma análise, usando o modelo bicompartimental e ao longo de 4 dias, à qualidade e precisão dos dados obtidos usando vários métodos numéricos, e discutir quais os mais ou menos eficazes e precisos, assim como as limitações de as previsões serem feitas com um sistema informático.

Breve descrição do Medicamento

Valaciclovir é um medicamento antiviral usado no tratamento de infecções causadas pelo vírus da herpes simples e vírus varicela-zoster, como a herpes zoster e a herpes genital. Embora não cure as infecções, diminui a dor e o prurido, ajuda as lesões a sarar e impede a formação de novas lesões.

Definições

mi - Massa no compartimento central do fármaco

mp - Massa no compartimento plasmático do fármaco

Ka - Constante cinética de absorção (h^{-1})

Ket - Constante cinética de eliminação total (h^{-1})

D(t) - Dose administrada como função do tempo (mg/h^{-1})

t - Tempo decorrido (h)

tmax - Instante (h) depois da administração, em que ocorre a concentração plasmática máxima (dado fornecido pelos laboratórios farmacêuticos).

Explicação da função de administração

A função de administração é a função que vai relatar a forma como a dose vai variar em função do tempo. No caso do *Valaciclovir*, um comprimido, vai resultar numa função em dente de serra.

Deste modo, no processo de definição da função, começamos por fazer o seguinte esboço:

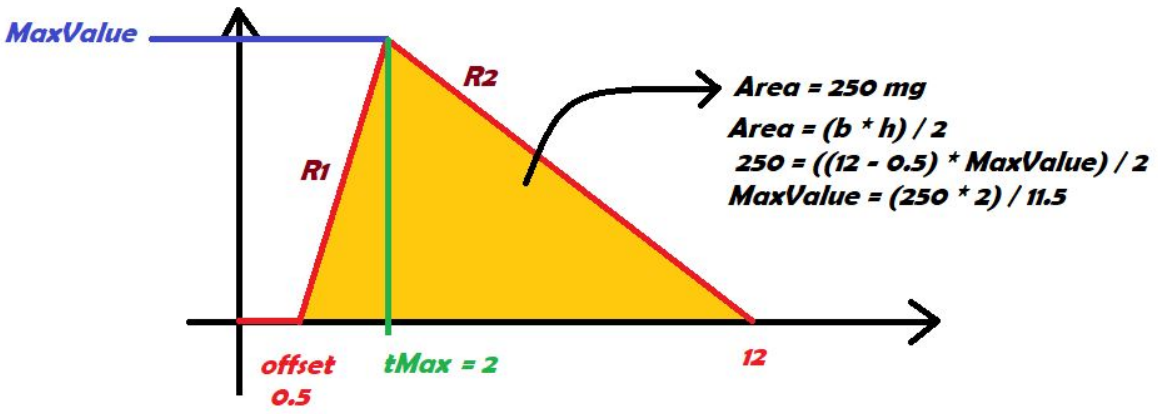

Figure 1.

Neste processo, partimos pela definição de um *offset* (tempo que o organismo demora a começar a absorver o antiviral) onde o valor da nossa função é nulo (pois nesse intervalo a dose administrada também irá ser nula). Dado que o medicamento em questão é um antiviral decidimos usar um offset de 0.5h.

Posto isto, sabendo que no intervalo [0, 12], o paciente toma 250 mg do comprimido, conseguimos descobrir qual é o valor máximo da nossa função através da área do dente de serra (como demonstra a figura).

De seguida, descobrimos o declive da reta *R1* (declive da função linear com expressão $y = m1 * t$), e da reta *R2* (declive da função afim com expressão $y = m2 * t + b$) através das seguintes expressões:

$$m1 = \frac{\frac{250 * 2}{11.5} - 0}{2 - 0.5} \qquad m2 = \frac{0 - \frac{250 * 2}{11.5}}{12 - 2}$$

e de forma a conseguirmos obter uma boa definição das funções descritas, efetuamos duas translações no eixo *xx* passando de $t = 0 \rightarrow t = 0.5$ na função representada pela reta *R1* e de $t = 0 \rightarrow t = 2$ na função representada pela reta *R2.*

Por fim, apenas nos faltava descobrir o valor de b para obtermos uma completa definição de ambas as funções e para podermos definir a equação da função de administração por ramos:

$$R2(12) = 0 = m2 * (t - 2) + b$$

$$b = -m2 * (12 - 2) = -m2 * 10$$

Assim, concluímos que a função iria ser definida por 3 ramos:

com $t \, \varepsilon \, [0, \, 0.5] \, com \, D(t) = 0$

com $t \, \varepsilon \, [0.5, 2] \, com \, D(t) = m1 * (x - 0.5)$

com $t \, \varepsilon \, [2, \ 12] \, com \, D(t) \ = \ m2 \ * (x \ - \ 2) \ + \ b$

```python
def d(t):
    if t <= 96:  # 4 days
        while t >= 12:
            t -= 12
        if t <= 0.5:
            return 0
        elif 0.5 < t <= 2:
            return (500 * (t - 0.5)) / 17.25
        elif 2 < t <= 12:
            return (-500 * (t - 2)) / 115 + 1000 / 23
    else:
        return 0
```

Figura 1 - Excerto de código de Python utilizado para obter a função de administração

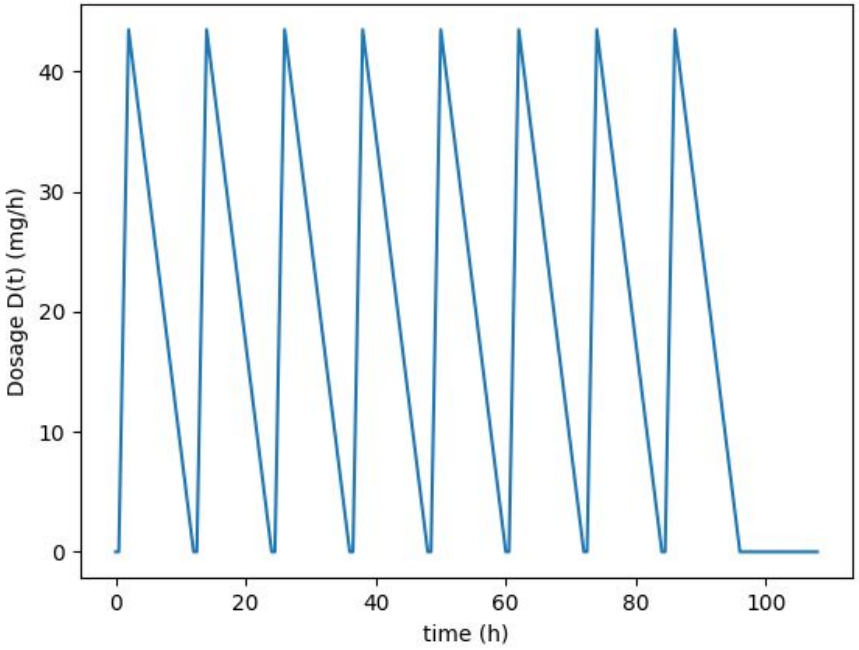

Figura 2 - Aproximação de uma função de administração em função do tempo(h) aplicada ao problema em questão.

Eficiência Computacional

As máquinas têm capacidade finita para guardar informação, logo conseguem apenas representar exatamente um número finito de números reais, cada um com um número fixo de dígitos (algarismos). Como a maioria dos problemas matemáticos utiliza o conjunto dos números reais, que é infinito e contínuo, isto leva a que se tenha de utilizar técnicas de arredondamento para os representar, conduzindo inevitavelmente a uma perda de precisão, especialmente quando isto ocorre em sucessão.

Corremos ainda o risco de incorrer em erros de truncatura quando estamos a trabalhar com métodos numéricos, visto que isto implica por vezes a substituição de um processo matemático infinito por uma aproximação finita, de modo a ser possível computá-lo.

Metodologia

O objetivo final deste trabalho era mapear numericamente o comportamento temporal da concentração no plasma sanguíneo de um fármaco, no nosso caso o já acima mencionado Valaciclovir,usando o modelo bicompartimental.

Este modelo bicompartimental opera segundo a premissa de que após ocorrer a administração do fármaco no paciente, este passa diretamente para um compartimento central, a partir do qual vai passando para o compartimento plasmático, segundo uma velocidade dada por Ka, a constante cinética de absorção. O fármaco é posteriormente eliminado do sistema a uma taxa dada pela constante cinética de eliminação total.

Este modelo pode então ser traduzido pelo seguinte sistema de equações diferenciais de 1ª ordem, em que Mi representa a massa de fármaco presente no compartimento central e Mp a massa de fármaco no compartimento plasmático.

$$\begin{cases} \frac{dm_i}{dt} = D(t) - K_a M_i \\ \frac{dm_p}{dt} = K_a M_i - K_{et} m_p \end{cases}$$

Temos portanto duas equações mas três variáveis (Ka, Mi e Mp), pelo que nos é providenciada uma terceira equação para encontrarmos o valor de Ka, para sermos capazes de modelar o comportamento de Mi e Mp:

$$K_a e^{-K_a t_{max}} - K_e e^{-K_e t_{max}} = 0$$

Isolamento de raízes

Antes de aplicar os métodos para achar os zeros da função acima, tivemos ainda de realizar um passo extra de isolamento das raízes, pois alguns métodos (método da bisseção e método da corda) são métodos intervalares que operam com um dado intervalo e depois vão alterando os extremos desse intervalo para chegar mais perto da solução. Este passo também nos foi útil para o uso de métodos não intervalares (método de Newton e

Picard-Peano) pois permitiu-nos obter *guesses* iniciais mais precisos, reduzindo o número de iterações necessárias para obter um resultado.

Utilizamos o programa Maxima para fazer um gráfico da função para os valores atribuídos ao nosso grupo (Tmax = 2h, Ke = 0.064 * 60 h^{-1}), obteve-se:

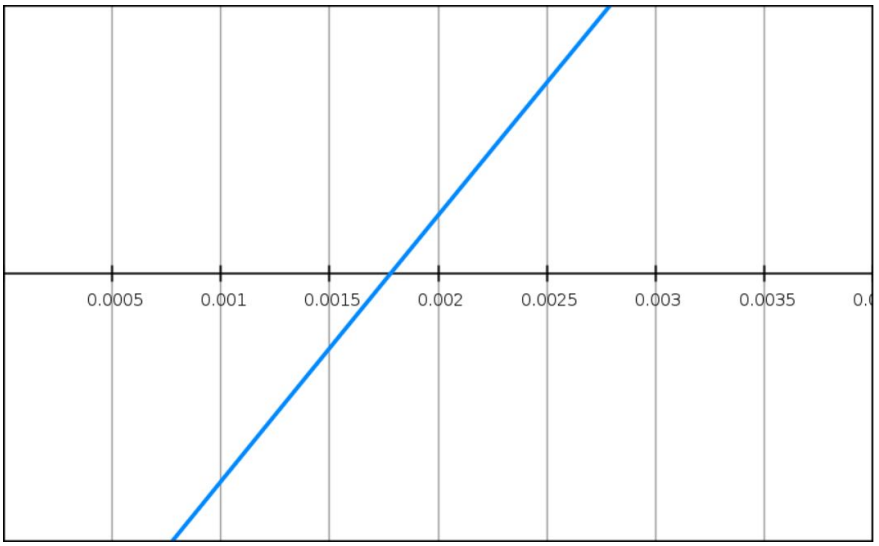

Figura 3 - Aproximação do intervalo próximo ao primeiro zero da função

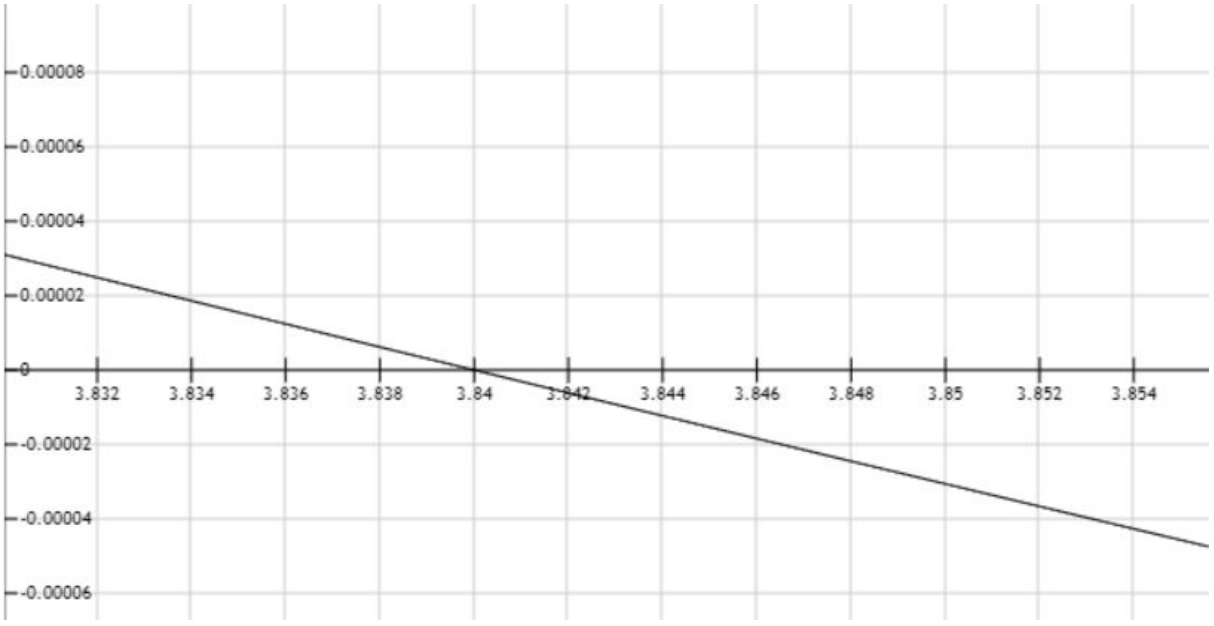

Figura 4 -- Aproximação do intervalo próximo ao segundo zero da função

Métodos para achar zeros reais de uma função real

Método da Bisseção

Este trata-se do método mais simples e menos pesado em termos de poder computacional para executar, pelo que é frequentemente utilizado para obter uma primeira aproximação da solução. Neste método, reduz-se o intervalo a partir do ponto médio do

intervalo e se o produto do valor da função no ponto médio com o valor da função no extremo inferior do intervalo for negativo, passa a ser o ponto máximo do intervalo. Para determinar os zeros da função, aplicou-se o método da bissecção aos intervalos: [0.001, 0.002] e [3.8, 3.9].

Decidiu-se utilizar o critério da precisão absoluta para decidir a paragem do processo, isto é, parar quando a diferença entre os extremos do intervalo que contém a raiz for menor que um dado valor pré-definido, que foi no nosso caso 10^{-4}.

```
def bisection_method(a, b, function, error):
    counter = 0
    while abs(a - b) > error:
        middle = (a + b) / 2
        if function(a) * function(middle) < 0:
            b = middle
        else:
            a = middle
        counter += 1
    return a, counter
```

Figura 5 - Implementação do método da bisseção em Python

As vantagens do método da Bissecção:

➔ Converge sempre (desde que exista raiz no intervalo inicial);

➔ Possibilidade de prever um majorante para o erro cometido ao fim de um certo número de iterações; - custo computacional de cada iteração muito baixo.

As desvantagens do método da Bissecção:

➔ A maior desvantagem reside no facto da sua convergência ser muito lenta (muitas iterações) quando comparada com a dos outros métodos. A ordem de convergência do método da bissecção é linear (P = 1) e a constante de convergência é igual a 1/2 (C = 1/2).

Método da Corda /Falsa Posição

A seguir foi implementado o método da corda que pode ser encarado como uma melhoria do método da bisseção. Em vez de se determinar um ponto médio do intervalo [a,b], determina-se um ponto P que resulta da interseção da reta secante que passa pelos pontos (a, f(a)) e (b, f(b)) com o eixo dos *xx*.

É de destacar que as sucessivas iterações neste algoritmo não provocam efeitos de cancelamento subtrativo, ou seja, o efeito da perda de dígitos significativos na subtração de números quase iguais, pois f(b) e f(a) têm sempre sinais contrários.
O critério de paragem optado foi novamente o critério de precisão absoluta com E = 10^{-4} e aplicou-se então o método da corda aos intervalos: [0.001, 0.002] e [3.8, 3.9].

As vantagens/desvantagens do método da Corda:

Como o método da Corda corresponde a um simples melhoramento do método da bisseção, as vantagens e desvantagens deste aplicam-se aqui.

```
def rope_method(a, b, function, error):
    counter = 0
    while abs(a - b) > error:
        w = (a * function(b) - b * function(a)) / (function(b) - function(a))
        if function(a) * function(w) < 0:
            b = w
        else:
            a = w
        counter += 1
    return a, counter
```

Figura 6 - Implementação do método da corda/ falsa posição em Python

Método de Newton

Este método consiste em substituir o gráfico da função pela tangente no ponto considerado, usando o zero desta como nova aproximação à raiz em cada sucessiva iteração. Como primeira aproximação à raiz foi escolhido o ponto intermédio de um dos intervalos usados nos métodos anteriores, 0.0015, e 3.9 um dos extremos dos intervalos.

```python
def newtons_method(guess, function, diff, error):
    new_x = guess
    old_x = guess + 10
    counter = 0
    while abs(new_x - old_x) > error:
        old_x = new_x
        new_x = old_x - function(old_x) / diff(old_x)
        counter += 1
    return new_x, counter
```

Figura 7 - Implementação do método de Newton em Python

As vantagens do método de Newton:

➔ Quando converge, tem convergência quadrática, o que significa que necessita poucas iterações para chegar à raíz.

➔ Necessita apenas de um ponto, para estimativa inicial, logo não requer o passo inicial de isolamento das raízes.

As desvantagens do método de Newton:

➔ Exige uma boa aproximação inicial; caso contrário pode divergir.

➔ Exige o cálculo da derivada em cada iteração, o que pode ser lento ou mesmo impossível.

➔ Exige que a derivada (no denominador) nunca se anule. Note-se que, mesmo para valores da derivada próximos de zero, a intersecção da tangente com o eixo dos XX é um ponto muito afastado.

Método de Picard-Peano

Por último, foi tentado o método de Picard Peano.

Para o método de Picard-peano foi necessário fazer um isolamento da variável da função (x). Deste modo, a função $g(ka)$ definida teria de passar pelo critério de convergência $|g'(guess)| < 1$. Com isto foram definidas duas funções para utilizar no método (uma para cada zero) que foram:

```
def function_g(ka):
    return - m.log(0.0017739836113215 41 / ka) / 2


def function_g2(ka):
    return (ket * m.exp(-ket * tMax)) / m.exp(-ka * tMax)
```

Figura 8 - Funções de critério de convergência

```
def picard_peano_method(guess, function, error):
    old_x = guess + 10
    new_x = guess
    counter = 0
    while abs(new_x - old_x) > error:
        old_x = new_x
        new_x = function(old_x)
        counter += 1
    return new_x, counter
```

Figura 9 - Implementação do método de Picard Peano em Python

As vantagens do método de Picard-Peano:

➔Relativamente ao método de Newton, acaba por ser melhor em certas situações pois evita o cálculo da derivada, o que em certas circunstâncias se apresenta impraticável.

Resultados obtidos e algumas observações

Método	Mínimo/Guess	Máximo	Erro	Nº Iterações	Resultado
Bisseção	0.001	0.002	10^{-4}	4	0.00175
Corda/Falsa Posição	0.001	0.002	10^{-4}	6	0.0017803113563309232
Newton	0.0015	-	10^{-4}	2	0.0017803113562813225
Picard-Peano	0.0015	-	10^{-4}	2	0.0017803014712598417

Tabela 1 - Resultados da aplicação dos métodos de encontrar zeros de funções no primeiro intervalo [0.001,0.002]

Método	Mínimo/Guess	Máximo	Erro	Nº Iterações	Resultado
Bisseção	3.8	3.9	10^{-4}	10	3.839941406249999
Corda/Falsa Posição	3.8	3.9	10^{-4}	10	3.8399999999999994
Newton	3.9	-	10^{-4}	3	3.8399999999377896
Picard-Peano	3.9	-	10^{-4}	5	3.84000222572088

Tabela 2 - Resultados da aplicação dos métodos de encontrar zeros de funções no primeiro intervalo [3.8,3.9]..

Dado termos encontrado duas raízes para o valor da constante, decidimos aceitar como valor “real” 3.84, dado que o outro valor se apresenta demasiado baixo e levaria a que basicamente nenhuma substância fosse absorvida para o compartimento plasmático.

Finalmente com o valor de Ka, podemos partir para a resolução do sistema de equações diferenciais acima mencionado, usando os métodos de resolução de Sistemas de Equações Diferenciais Ordinárias.

Métodos de resolução de Sistemas de Equações Diferenciais Ordinárias empregados

A sua precisão pode ser vagamente julgada a partir de o indicador *QC*, o coeficiente de convergência, calculado por:

$$\frac{S' - S}{S'' - S}$$

Em que *S* é uma solução obtida com *n* passos, *S'* é uma solução obtida com *2n* passos e *S''* vem de *4n* passos.

O *QC*, quando cumprido para o respetivo método, permite-nos estimar o erro absoluto também a partir da fórmula:

$$S'' - S' \approx (2^{ordem\ do\ método} - 1) * Erro$$

Em que a ordem destes do método representa a forma como o erro vai variar dependendo do valor do passo a tomar (dependendo do *n*)). A ordem de um método mede o quão rapidamente este converge para a solução analítica quando se diminuem os passos na integração numérica.

Método de Euler e Euler melhorado

Este é o mais simples e mais antigo dos métodos numéricos utilizados na solução particular de equações diferenciais, sendo baseado na expansão da função em séries de Taylor, assim, podendo-se portanto expandir a função na vizinhança do ponto x como:

$$y(x_{n+1}) = y(x_n) + h.f\left(x_n, y(x_n)\right)$$

Como este é um método de ordem 1, o seu quociente de convergência desejado é $2^1 = 2$.

```python
def euler(func_mi, func_mp, inital_t, initial_mi, initial_mp, final_t, steps)
    list_t = [inital_t]
    list_mi = [initial_mi]
    list_mp = [initial_mp]
    mi = initial_mi
    mp = initial_mp
    t = inital_t
    h = (final_t - inital_t) / steps
    while t < final_t:
        t = t + h
        mi = mi + func_mi(t, mi, mp) * h
        mp = mp + func_mp(t, mi, mp) * h
        list_t.append(t)
        list_mi.append(mi)
        list_mp.append(mp)
    print("mi: ", mi)
    print("Mp: ", mp)
    return mi, mp, list_t, list_mi, list_mp
```

Figura 10 - Implementação do método de Euler em Python.

```python
def euler_improved(func_mi, func_mp, inital_t, initial_mi, initial_mp, final_t, steps):
    list_t = [inital_t]
    list_mi = [initial_mi]
    list_mp = [initial_mp]
    mi = initial_mi
    mp = initial_mp
    t = inital_t
    h = (final_t - inital_t) / steps
    mi_prev = mi - func_mi(mi, mp) * h
    mp_prev = mp - func_mp(mi, mp) * h
    while t < final_t:
        mi_ = func_mi(t, mi, mp)
        mp_ = func_mp(t, mi, mp)
        p_ = mi_prev + 2 * mi_ * h
        p1_ = mp_prev + 2 * mp_ * h
        mi_prev = mi
        mp_prev = mp

        p_1 = func_mi(t + h, p_, p1_)
        p1_1 = func_mp(t + h, p_, p1_)

        delta_mi = (p_1 + mi_)/2 * h
        delta_mp = (p1_1 + mp_)/2 * h
        mi = mi + delta_mi
        mp = mp + delta_mp
        t = t + h
        list_t.append(t)
        list_mi.append(mi)
        list_mp.append(mp)
    return mi, mp, list_t, list_mi, list_mp
```

Figura 11 - Implementação do método de Euler melhorado em Python.

As vantagens dos métodos de Euler:

É um método simples de implementar e cada iteração é computacionalmente leve, requerendo apenas calcular valores da função f(x) e efetuar operações aritméticas básicas.

As desvantagens dos métodos de Euler:

Devido a limitações computacionais, erros de arredondamento crescem quando se diminui o tamanho dos passos, devido ao aumento do número de iterações, logo aumento do número de operações aritméticas a realizar. Por este motivo, por vezes ocorre até mesmo divergência ou mesmo valores errados.
Como o método de Euler é apenas de primeira ordem, ele vai convergir para a solução relativamente lentamente, daí que requer tipicamente um número elevado de iterações, exacerbando o problema acima.

Uma forma de resolver este problema é aumentar a ordem do método numérico, e é aqui que entram os métodos de Runge-Kutta.

Método de Runge-Kutta

Os métodos de Runge-Kutta procuram avaliar a derivada média no intervalo de integração, usando para isso uma média ponderada de derivadas calculadas em diversos pontos do mesmo intervalo. Cada método de Runge-Kutta consiste em comparar um polinómio de Taylor apropriado para eliminar o cálculo das derivadas, fazendo-se várias avaliações da função a cada passo. O método de Runge-Kutta pode ser entendido como um aperfeiçoamento do método de Euler, com uma melhor estimativa da derivada da função.

As vantagens dos métodos de Runge-Kutta:

➔São métodos de ordem superior, o que significa que necessitam de menos iterações para convergir na solução, diminuindo os erros resultantes de executar múltiplas operações num computador de seguida.

As desvantagens dos métodos de Runge-Kutta:

➔Mais complexo de implementar.
➔Cada iteração individual realiza mais operações, sendo portanto, mais computacionalmente taxante.

Método de Runge-Kutta de Segunda Ordem

Para o método Runge-Kutta de Segunda Ordem, o algoritmo corresponde ao seguinte:

```python
def rk2_systems(func_mi, func_mp, initial_t, initial_mi, initial_mp, final_t, steps):
    list_t = [initial_t]
    list_mi = [initial_mi]
    list_mp = [initial_mp]
    h = (final_t - initial_t) / steps
    t = initial_t
    mi = initial_mi
    mp = initial_mp
    while t < final_t:
        t = t + h
        mi_linha = func_mi(t, mi, mp)
        mp_linha = func_mp(t, mi, mp)
        mi_mid = func_mi(t + h/2, mi + (h/2) * mi_linha, mp + (h/2) * mp_linha)
        mp_mid = func_mp(t + h/2, mi + (h/2) * mi_linha, mp + (h/2) * mp_linha)
        delta_mi = mi_mid * h
        delta_mp = mp_mid * h
        mi = mi + delta_mi
        mp = mp + delta_mp
        list_t.append(t)
        list_mi.append(mi)
        list_mp.append(mp)
    print("mi: ", mi)
    print("Mp: ", mp)
    return mi, mp, list_t, list_mi, list_mp
```

Figura 12 - Implementação do método RK2 em Python.

Método de Runge-Kutta Quarta Ordem

```python
def rk4_systems(func_mi, func_mp, intial_t, initial_mi, initial_mp, final_t, steps):
    list_t = [intial_t]
    list_mi = [initial_mi]
    list_mp = [initial_mp]
    h = (final_t - intial_t) / steps
    t = intial_t
    mi = initial_mi
    mp = initial_mp
    while t < final_t:
        t = t + h
        d1_mi = h * func_mi(t, mi, mp)
        d1_mp = h * func_mp(t, mi, mp)

        d2_mi = h * func_mi(t + h/2, mi + d1_mi/2, mp + d1_mp/2)
        d2_mp = h * func_mp(t + h/2, mi + d1_mi/2, mp + d1_mp/2)

        d3_mi = h * func_mi(t + h/2, mi + d2_mi/2, mp + d2_mp/2)
        d3_mp = h * func_mp(t + h/2, mi + d2_mi/2, mp + d2_mp/2)

        d4_mi = h * func_mi(t + h, mi + d3_mi, mp + d3_mp)
        d4_mp = h * func_mp(t + h, mi + d3_mi, mp + d3_mp)

        mi = d1_mi/6 + d2_mi/3 + d3_mi/3 + d4_mi/6 + mi
        mp = d1_mp/6 + d1_mp/3 + d3_mp/3 + d4_mp/6 + mp

        list_t.append(t)
        list_mi.append(mi)
        list_mp.append(mp)
    print("mi: ", mi)
    print("Mp: ", mp)
    return mi, mp, list_t, list_mi, list_mp
```

Figura 13 - Implementação do método RK4 em Python.

Cálculo de QC e Erro Absoluto

Método	QC (mi)	Erro absoluto (mi)	QC (mp)	Steps	Erro absoluto (mp)
Euler	1.9946866485872223	0.001043172602258381	1.992018282687645	1250	0.00435383861294802
RK2	6.939294242403852	3.678549338112968	0.008163747846071531	1781	0.0002188798503790083
RK4	5.993247523190297	0.001762710610153452 3	5.8649453540805295e-05	1403	0.000429977459181248

Tabela 3 - Resultados obtidos da aplicação dos métodos de resolução de sistemas de equações diferenciais ordinárias

Evolução de Massa no compartimento central e no compartimento plasmático ao longo do tempo

```
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

from getting_mi_and_mp_rk4 import *


mi3, mp3, list_t, list_mi, list_mp = rk4_systems(
    func_MI, func_MP, 0, 0, 0, 100, 10000)
plt.xlabel("time(h)")
plt.ylabel("mass(g)")
plt.plot(list_t, list_mp)
plt.plot(list_t, list_mi)
plt.legend(["Compartimento plasmático", "Compartimento central"])
plt.show()
```

Figura 14 - Evolução dos valores de massa do fármaco no compartimento plasmático e central para um Ka de 3.84

Para Ka = 3.84

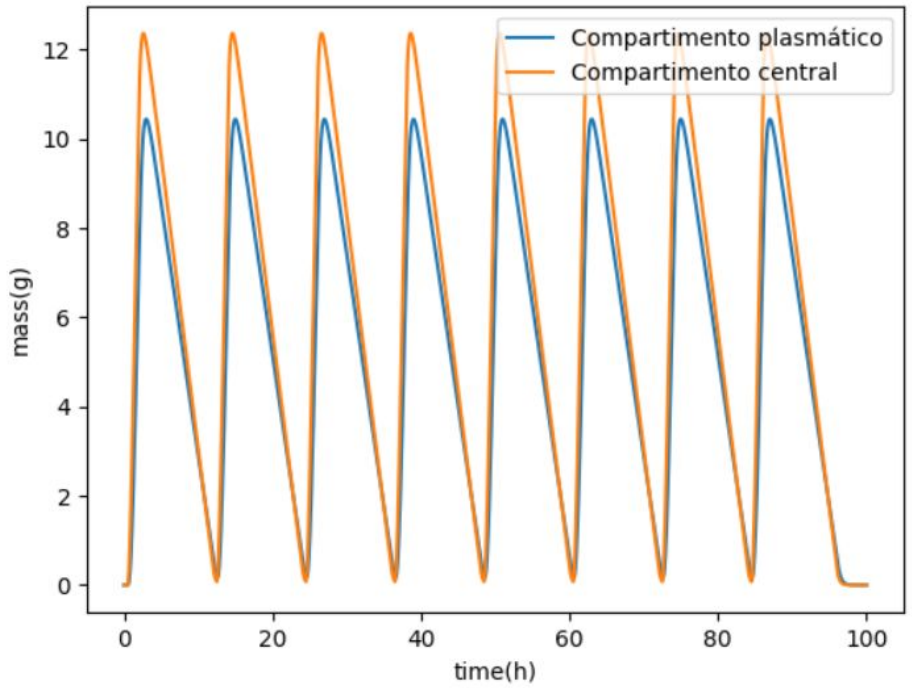

Figura 14 - Evolução dos valores de massa do fármaco no compartimento plasmático e central para um Ka de 3.84

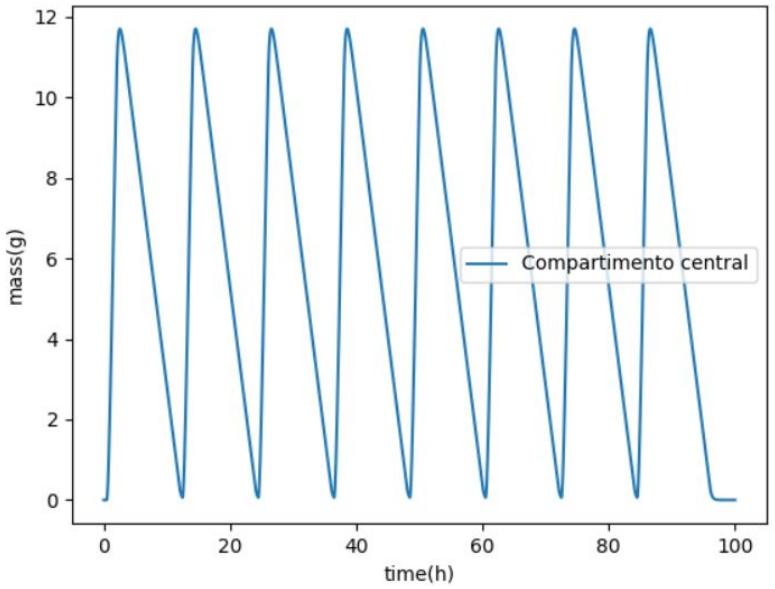

Figura 15 - Evolução dos valores de massa do fármaco no compartimento central para um Ka de 3.84

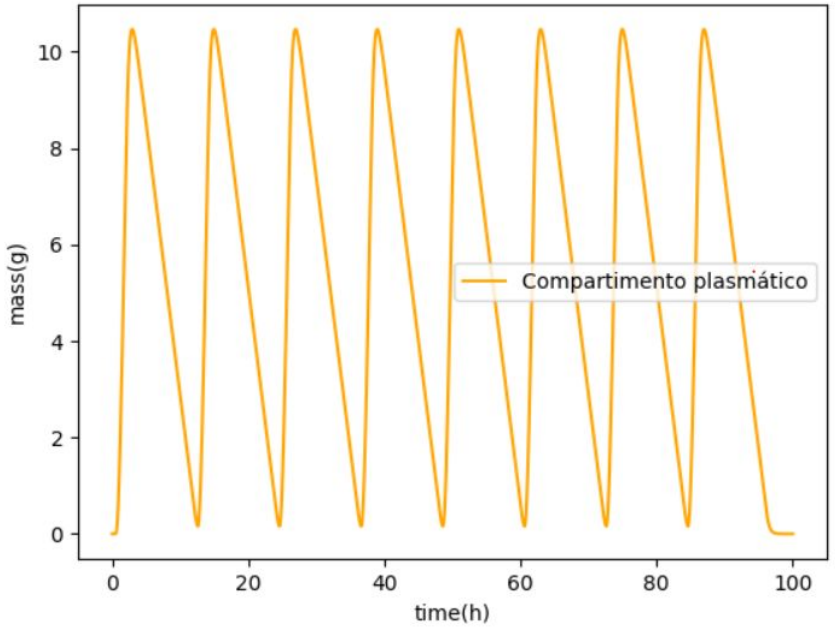

Figura 16 - Evolução dos valores de massa do fármaco no compartimento plasmático para um Ka de 3.84

Para Ka = 0.0017803

Como obtivemos dois valores para o Ka, decidimos mapear a oscilação dos valores de mi e mp tanto para um como para o outro, verificando esta enorme disparidade: de um lado, o valor mais elevado para a constante de absorção produz um gráfico em que a linha com mais variação corresponde ao compartimento central e a com menor variação ao compartimento plasmático, como é esperado. Como este Ka é relativamente grande, verificamos rápida absorção do fármaco, pelo que o traço de ambas as funções se encontra bastante próximo.

Por outro lado, para o Ka menor de 0.0017803, verifica-se que o aumento do fármaco no compartimento central segue uma função semelhante a degraus, já que este se vai acumulando devido à significativamente mais lenta saída para o compartimento plasmático, dada pela constante de absorção cinética.

A função relativa ao compartimento plasmático segue uma forma aproximadamente igual à do compartimento central mas com valores inferiores.

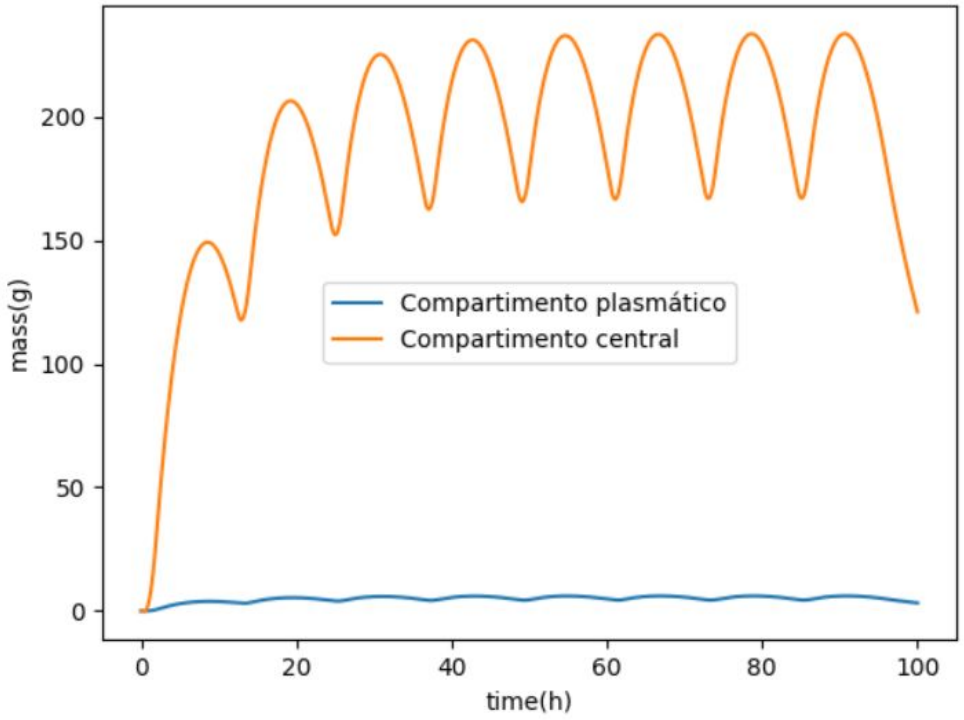

Figura 17 - Evolução dos valores de massa do fármaco no compartimento plasmático e central para um Ka de 0.0017803.

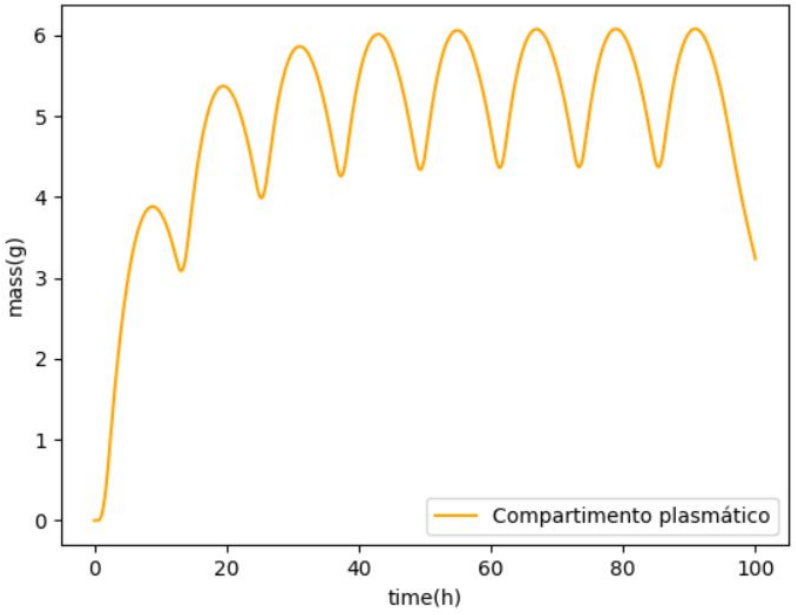

Figura 18 - Evolução dos valores de massa do fármaco no compartimento plasmático para um Ka de 0.0017803.

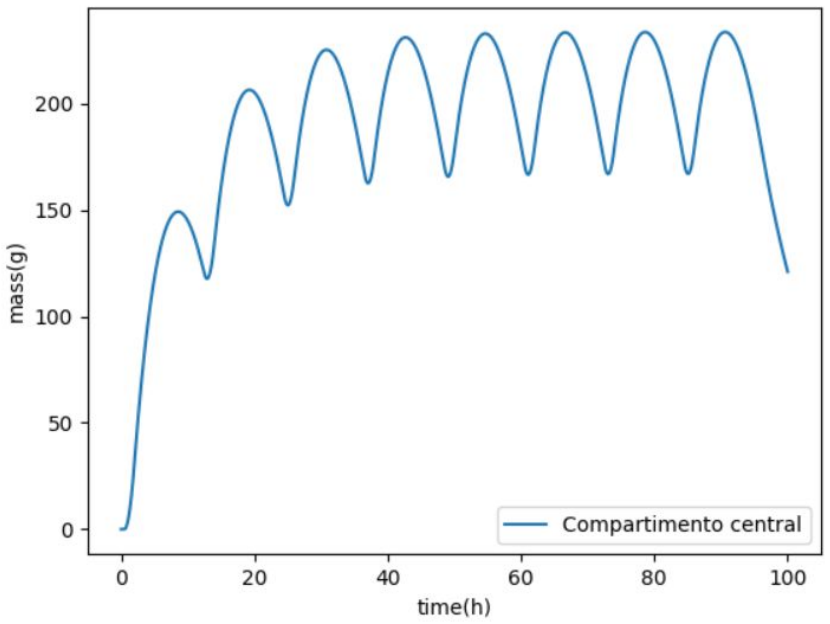

Figura 19 - Evolução dos valores de massa do fármaco no compartimento central para um Ka de 0.0017803.

Qualidade dos resultados

No que toca à qualidade dos resultados, tentamos sempre obter os resultados com o menor erro possível.

No que toca à obtenção do *Ka*, utilizamos sempre aquele valor obtido pelos nossos métodos que mais se aproximasse do valor dado pelo *máxima*, pois consideramos que este programa nos dá os valores mais exatos.

Quanto à obtenção dos valores de *mi* e *mp*, para podermos gerar os gráficos, procuramos sempre obter um quociente de convergência (*QC*) que mais se adequasse aos métodos utilizados (mas nem sempre foi possível).

Assim, partindo dos resultados obtidos pela análise dos gráficos, podemos concluir que as nossas estimativas estão corretas.

Conclusão

A partir da observação dos gráficos obtidos podemos concluir que descrevem o comportamento da função de forma praticamente idêntica mesmo utilizando diferentes métodos, sendo que as únicas discrepâncias encontradas são quando se utiliza um passo maior e portanto menor número de iterações. Em ambos, a massa do fármaco no compartimento plasmático começa por aumentar até atingir um máximo local, voltando depois a diminuir até ser novamente administrado o fármaco ao fim de doze horas, posteriormente atinge um novo máximo local, superior ao obtido anteriormente e a partir deste ponto, a função descreve um comportamento aproximadamente periódico repetindo-se a cada administração do fármaco de 12 em 12 horas.

Quanto à massa do fármaco no compartimento central, o comportamento desta função é periódico, começando por aumentar a massa de fármaco até atingir um pico que é mais ou menos igual sempre que é administrada nova dosagem descendo depois até à administração seguinte.

Quanto aos métodos utilizados, no método de Newton tivemos como primeiro inconveniente o facto deste método exigir o conhecimento da derivada da função e o cálculo desta em cada iteração. Seria possível para evitar isto, usar sempre o valor da derivada no ponto original, mas, esta estratégia, apesar de diminuir o esforço de cálculo aumenta significativamente o número de iterações necessário para obter a precisão pretendida, dependendo bastante da estrutura local da função, o que o tornaria um pouco menos de confiança.

O método de Euler provou-se o mais impreciso, apesar da sua implementação ser das mais simples. A sua margem de erro pode ser atribuída à substituição não apropriada e ingénua de um integral por uma soma de partes, que apesar de pequenas, constituem uma diferença significativa. Há ainda a apontar o facto de que estamos a usar como aproximação um valor de derivada apenas válido para o ponto inicial do intervalo.
Foi aplicado o critério de convergência que diz que se o quociente de convergência for aproximadamente 2, a função está a convergir, e é possível calcular o erro absoluto.
No entanto, o quociente de convergência obtido por nós, não se aproxima sempre de dois ao longo da função, para certos passos(h) utilizados. Concluímos que este se deve ao facto de, efetivamente, a função utilizada não convergir para nenhum ponto, isto é, a nossa função de administração, por ser periódica e não contínua, não permite que obtenhamos um valor de QC de 2 ao longo da mesma. Assim , como o QC obtido nem sempre foi o valor pretendido de 2, podemos descartar o erro absoluto calculado, visto que não tem qualquer significado nestas circunstâncias.

O método de Runge-Kutta de quarta ordem (rk4), trata-se de uma técnica mais fiável que a do método de Euler visto que, em vez de utilizar o valor inicial da derivada para o resto do intervalo, utiliza a derivada média, ponderada.
Tal como o método de Euler, para avaliar o erro cometido também é comum utilizar o critério de convergência, no entanto, pelas mesmas razões discutidas anteriormente este tornou-se inviável sendo que o QC muitas vezes é diferente de 16.

Bibliografia

(1)
https://www.fc.unesp.br/Home/Departamentos/Matematica/revistacqd2228/v07a02-comparacao-entre-metodos-numericos.pdf

(2) https://www.indice.eu/pt/medicamentos/DCI/valaciclovir/informacao-geral

(3) Madureira, C., Vila, C., Dinis, M. de L., Sousa, R. J., Carvalho, J. S. 17 de novembro de 2020. “Métodos Numéricos”. “Um curso para o Mestrado Integrado em Engenharia Informática e de Computadores da FEUP”.

(4) https://www2.unitins.br/bibliotecamidia/Files/Documento/BM_634024318710992500leitura_complementar_3_aula_1_calculo_numerico.pdf