MAP3121 - Métodos Numéricos e Aplicações

Exercício Programa 1

Autovalores e Autovetores de Matrizes Tridiagonais Simétricas

Data: 01/07/2021	
Membros:	
11261110 - Antonio Lago Araújo Seixas	**Turma** : 1
11259715 - Vanderson da Silva dos Santos	**Turma:** 2

São Paulo
2021

Índice

Introdução

Descrição do problema

O presente trabalho traz a importância da obtenção de autovetores e autovalores de matrizes tridiagonais simétricas junto com sua aplicabilidade em problemas reais. Com isso, o Exercício Programa desenvolvido faz o uso de um método numérico para a obtenção de tais valores, sendo ele: o Algoritmo QR.

O Algoritmo QR, faz uso de dois outros métodos, sendo eles: a Rotação de Givens e a Fatoração QR. Ao longo deste relatório, será explicado como a dupla raciocinou e implementou tais métodos de forma a ter a execução plena do algoritmo.

Ao fim, o algoritmo é aplicado a um problema físico: a obtenção das frequências fundamentais de dois sistemas massa-mola com 5 e 10 massas interligadas em série. Com isso é demonstrado como o método implementado é utilizado para soluções de equações diferenciais.

Abordagem de resolução do problema

Para a resolução do problema, por uma preferência de praticidade, a dupla optou pelo uso da linguagem de programação *python 3.7*, fazendo uso das bibliotecas *numpy*, a qual é utilizada principalmente para as operações entre vetores e matrizes, e *matplotlib*, usada para a construção dos gráficos exigidos.

Com isso, foram criado 5 arquivos principais sendo eles:

- “QR_algorithm.py”: O qual é implementado o algoritmo QR
- “assignment_a.py”: O qual é implementado a tarefa A
- “assignment_b.py”: O qual é implementado a tarefa B
- “assignment_c.py”: O qual é implementado a tarefa C
- “_main_.py”: Arquivo main no qual o usuário escolhe qual das tarefas que pretende executar.

Implementação

Algoritmo QR

Rotações de givens e fatoração QR

As rotações de givens tratam-se de uma transformação algébrica ortogonal. Ela visa rotacionar uma certa matriz A no sentido anti-horário em um ângulo θ em relação à origem. Para isso, a matriz A é multiplicada pela matriz Q, a qual representa a transformação.

No problema proposto, a rotação de givens é utilizada especificamente para a aplicação da fatoração QR, esta a qual aplica sucessivas transformações de givens específicas a fim de transformar a matriz A em uma matriz R triangular superior. Portanto, a dupla implementou somente o caso alvo da transformação de givens a qual é utilizada no problema.

Neste caso específico, para uma matriz A_{nxn} tridiagonal simétrica, deve-se aplicar "n" vezes as transformações de givens, sendo que cada transformação Q_k; $1 < k < n$ admite o seguinte formato:

$$Qk = \begin{bmatrix} c_k & -s_k & 0 & \dots & 0 \\ s_k & c_k & 0 & \dots & 0 \\ 0 & 0 & 0 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix} \Rightarrow Q_{k+1} = \begin{bmatrix} 0 & \dots & \dots & \dots & 0 \\ \vdots & c_{k+1} & -s_{k+1} & \dots & 0 \\ \vdots & s_{k+1} & c_{k+1} & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Sendo c_k e s_k definidos pelas equações

$$c_k = \frac{\alpha_k}{\sqrt{\alpha_k^2 + \beta_k^2}} \quad \text{e} \quad s_k = -\frac{\beta_k}{\sqrt{\alpha_k^2 + \beta_k^2}}$$

Sendo α_k os valores na diagonal principal de A e β_k, os valores da subdiagonal de A.Nesta situação, a dupla implementou duas funções, sendo elas:

- *givens_rotations_Qk(k,A)*: Esta função recebe os parâmetros *k* e *A*, os quais correspondem ao número da interação e a matriz a ser transformada, respectivamente. A partir de tais parâmetros, ela calcula os valores de c_k e s_k e retorna a transformação de givens correspondente a interação. Sua implementação está mostrada na figura a seguir:

```
## Function to get the givens_rotation_Qk matrix for givens rotation
def givens_rotation_Qk(k,A):
    Q = np.eye(len(A))
    alfa_k = A[k,k]
    beta_k = A[k+1,k]
    ck = alfa_k/(np.sqrt((alfa_k**2)+(beta_k**2)))
    sk = -(beta_k)/(np.sqrt((alfa_k**2)+(beta_k**2)))
    Q[k,k] = ck
    Q[k,k+1] = -sk
    Q[k+1,k] = sk
    Q[k+1,k+1] = ck
    return Q
```

- *QR_fatoration(A)*: Esta recebe como parâmetro a matriz A, a qual é o alvo da fatoração e retorna a matriz R, triangular superior, e matriz Q a qual se refere a multiplicação de todas transformações de givens realizadas. Para isso a função, declara as matrizes R = A e Q sendo a identidade, e partir de um *for loop* tais valores são atualizados sendo multiplicados pelo retorno da chamada da função *givens_rotations(k,A)*. Segue abaixo a sua implementação:

```python
## Function which returns a matrix after a givens rotation
def QR_fatoration(A):
    R = A
    matrix_i_size = len(A)
    Q =np.eye(matrix_i_size)
    Q_iteration_size = matrix_i_size - 1
    for i in range (0, Q_iteration_size,1):
        Q = Q@np.transpose(givens_rotation_Qk(i,R))
        R = givens_rotation_Qk(i,R)@R
    return [R,Q]
```

Algoritmo QR

O algoritmo QR visa determinar os autovalores e autovetores de uma matriz simétrica A. Sua aplicação pode ser feita de duas formas: com e sem deslocamentos espectrais.

Os deslocamentos espectrais atuam para minimizar as interações necessárias para a convergência do algoritmo. O deslocamento espectral são definidos a partir da heurística de Wilkinson, μ_k, sendo 'k' referente a interação correspondente ela é definida como:

$$d_k = (\alpha_{n-1}^{(k)} - \alpha_n^{(k)})/2 \text{ defina } \mu_k = \alpha_n^{(k)} + d_k - sgn(d_k)\sqrt{d_k^2 + (\beta_{n-1}^{(k)})^2}$$

$$\text{onde } sgn(d) = 1 \text{ se } d \geq 0 \text{ e } sgn(d) = -1 \text{ caso contrário.}$$

Sendo fornecido pelo enunciado, a dupla se pautou no pseudocódigo do algoritmo QR, o qual segue na figura abaixo:

1: Sejam $A^{(0)} = A \in \mathbb{R}^{n\times n}$ uma matriz tridiagonal simétrica, $V^{(0)} = I$ e $\mu_0 = 0$. O algoritmo calc
a sua forma diagonal semelhante $A = V\Lambda V^T$.

2: $k = 0$

3: **para** $m = n, n-1, \ldots, 2$ **faça**

4: **repita**

5: se $k > 0$ calcule μ_k pela heurística de Wilkinson

6: $A^{(k)} - \mu_k I \rightarrow Q^{(k)}R^{(k)}$

7: $A^{(k+1)} = R^{(k)}Q^{(k)} + \mu_k I$

8: $V^{(k+1)} = V^{(k)}Q^{(k)}$

9: $k = k + 1$

10: **até que** $|\beta_{m-1}^{(k)}| < \epsilon$

11: **fim do para**

12: $\Lambda = A^{(k)}$

13: $V = V^{(k)}$

Sendo Λ a matriz de autovalores de A, V sua matriz de autovetores e μ_k o deslocamento espectral, sendo igual a 0 quando não se utiliza o deslocamento, foram implantadas mais duas funções a fim de se criar um código para tal algoritmo sendo elas:

- *wilkinson_heuristics(A,n):* Esta função recebe como parâmetros a matriz [A] e o inteiro n. A partir da matriz [A] é calculado o deslocamento espectral sendo que o inteiro "n" indica para qual posição da matriz deve ser calculado o deslocamento. A função retorna os valores de μ_k e de $\beta_{n-1}^{(k)}$. Segue abaixo sua implementação:

```
def wilkinson_heuristics(A,n):
    get_sgnd = lambda d: 1 if d >= 0 else -1
    alfa_ant = A[n-2,n-2]
    alfa = A[n-1,n-1]
    beta = A[(n-2)+1,n-2]
    dk = (alfa_ant - alfa)/2
    uk = alfa + dk - get_sgnd(dk)*(np.sqrt((dk**2)+(beta**2)))
    return [uk,beta]
```

- *QR_algorithm(A, erro = 1/1000000, spectral_shift = true):* Esta função recebe como parâmetros a matriz A, a qual será aplicada o algoritmo, e o valor do erro, para fins de convergência, e o spcetral_shift, o qual especifica de seve utilizar ou não o deslocamento espectral, caso não seja especificado na chamada da função, o erro admite valor 0,000001 e admite-se o uso do deslocamento espectral. Dentro da implementação da função, é definida uma função auxiliar *get_uk* a qual chama a função *wilkinson_heuristics(A,n)* caso não seja a primeira interação ou spectral_shift = true, caso contrário é retornado o valor 0. A implementação da função seguiu fielmente o pseudocódigo fornecido, havendo dois loops um *for loop*, para iterar sobre subdiagonal da matriz A, e um *while loop*, para que se realize iterações até que o valor de β_k seja menor do que o erro estabelecido. A função, além de retornar as matrizes Λ e V, também é retornado a variável k, a qual indica o número de iterações que o algoritmo necessitou efetuar. A implementação da função está mostrada a seguir:

```
def QR_algorithm(A, erro = 1/1000000, spectral_shift = True):
    #round_parameter = len(str(int(1/erro)))
    get_uk = lambda A,n,k,spectral_shift: wilkinson_heuristics(A,n)[0] if ((k > 0) and (spectral_shift == True)) else 0
    V = I = np.eye(len(A))
    k = 0
    for m in range(len(A),1,-1):
        beta = wilkinson_heuristics(A,m)[1]
        while(abs(beta)>erro):
            uk = get_uk(A,m,k,spectral_shift)
            R,Q = QR_fatoration(A - (uk*I))
            A = R@Q + uk*I
            V = V@Q
            k = k + 1
            beta = wilkinson_heuristics(A,m)[1]
        A = np.matrix.round(A,7) # A = np.matrix.round(A,round_parameter - 1)
    lamb = A
    return [V,lamb,k]
```

Tarefas

Tarefa A

A primeira tarefa consiste na aplicação direta do Algoritmo QR em uma matriz tridiagonal simétrica, analisando, além dos resultados dos autovalores e autovetores, o número de iterações necessárias com e sem a utilização de deslocamento espectrais.

A matriz demandada pela tarefa possui os valores de sua diagonal principal, α_k, iguais a 2 e os valores de sua subdiagonal, β_k, iguais a -1, sendo ela com dimensões nxn com os seguintes valores de n: 4,8, 16 e 32.

Para a implementação desta primeira tarefa, foram feitas as seguintes funções:

- *make_tridiagonal_matrix(n,alfa, beta)*: Esta função visa retornar uma matriz tridiagonal simétrica de tamanho "n", com valores "alfa" em sua diagonal principal e valores "beta" em suas sub-diagonais. Segue abaixo sua implementação:

```
def make_tridiagonal_matrix(n,alfa,beta):
    A = alfa*np.eye(n)
    for i in range (0,n-1):
        A[i,i+1] = beta
        A[i+1,i] = beta
    return A
```

- *test_matrix_values(A,V,lamb):* Esta função visa apenas printar os valores de A, V e Λ de forma mais organizada, além de printar o valor de $V * \Lambda * V^T$, averiguando se o algoritmo QR foi executado corretamente. Segue abaixo sua implementação:

```
def test_matrix_values(A,V,lamb):
    print("V = \n", V)
    print("lamb \n= ",lamb)
    new_A = V@lamb@np.transpose(V)
    new_A = np.matrix.round(new_A,3)
    print("A = \n", A)
    print("V Λ V_transposto = \n", new_A)
```

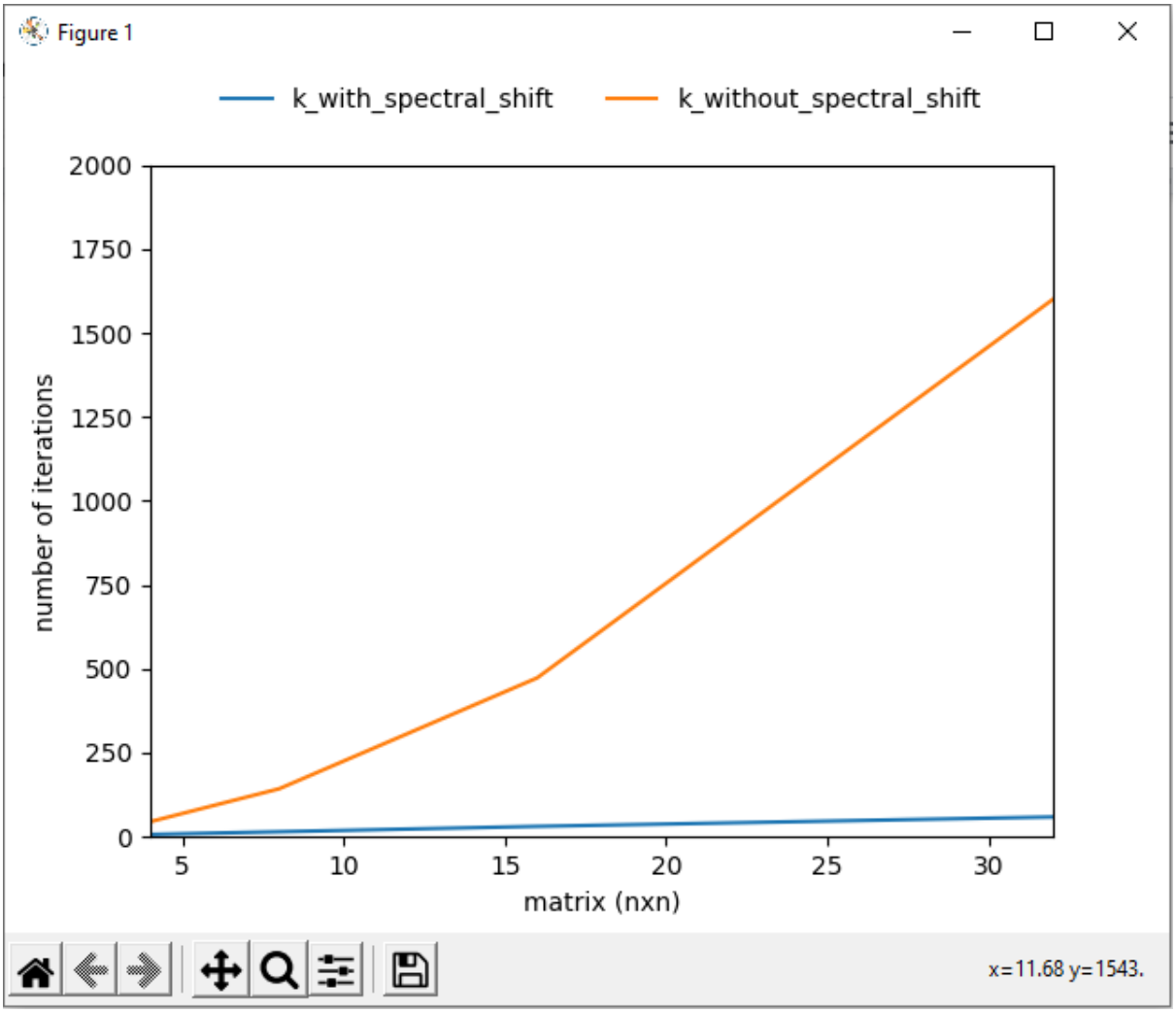
- *assignment_a()*: Esta função trata-se da tarefa em si. Ela inicialmente declara os valores do erro e de alfa e beta, e em seguida 3: um para os valores de n, e outros dois para os valores de k, com ou sem deslocamento espectral, para cada valor de n; tais vetores são usados posteriormente. O elemento principal da função é um *for loop*, no qual para cada valor de n é chamada a função *make_tridiagonal_matrix (n,alfa,beta), e a partir dela é aplicada a função QR_algorithm(A,erro,spectral_shift)* com e sem deslocamento espectral, sendo exposto o número de interações em cada caso. Além disso, a cada *loop* são adicionados valores aos 3 vetores ditos anteriormente, os quais, fora do *for loop*, são utilizados para construir um gráfico de k, com e sem deslocamento espectral, em função de n. A implementação da função segue abaixo:

```python
def assignment_a():
    erro = 1e-6
    alfa = 2
    beta = -1

    k_with_spectral_shift_list = np.array([])
    k_without_spectral_shift_list = np.array([])
    print("Assignment A: Example of using the QR algorithm in a tridiagonal matrix")
    print("For this assignment we used: \n 'n' as size of the matirx \n 'k' as number of iterations")
    n_list = np.array([])
    for i in range (2,6):
        n = 2**i
        print("For n = ", n)
        n_list = np.append(n_list,n)
        A_matrix = make_tridiagonal_matrix(n,alfa,beta)

        V,lamb,k = QR_algorithm(A_matrix,erro = erro,spectral_shift = True)
        k_with_spectral_shift_list = np.append(k_with_spectral_shift_list,k)
        print("k with spectral shift = ", k)
        test_matrix_values(A_matrix,V,lamb)

        V,lamb,k = QR_algorithm(A_matrix,erro = erro,spectral_shift = False)
        k_without_spectral_shift_list = np.append(k_without_spectral_shift_list,k)
        print("k without spectral shift = ", k)
        #test_matrix_values(A_matrix,V,lamb)

    plt.xlim(min(n_list), max(n_list))
    plt.ylim(0, 2000)

    plt.plot(n_list,k_with_spectral_shift_list,label='k_with_spectral_shift')
    plt.plot(n_list,k_without_spectral_shift_list,label='k_without_spectral_shift')
    plt.ylabel('number of iterations')
    plt.xlabel('matrix (nxn)')

    plt.legend(loc='upper center', bbox_to_anchor=(0.5, 1.15), ncol=2, frameon=False)
    plt.show()
```

Ao executar o código, é obtido a seguinte saída no terminal:

```
================================================================================================
============================== assignment A ====================================================
================================================================================================

Assignment A: Example of using the QR algorithm in a tridiagonal matrix
For this assignment we used:
 'n' as size of the matirx
 'k' as number of iterations
For n =  4
k with spectral shift =  7
k without spectral shift =  45
For n =  8
k with spectral shift =  15
k without spectral shift =  143
For n =  16
k with spectral shift =  31
k without spectral shift =  473
For n =  32
k with spectral shift =  59
k without spectral shift =  1600
```

Além da saída acima, é gerado o seguinte gráfico do número de iterações em função tamanho da matriz A:



Tarefa B

A tarefa B consiste em solucionar um sistema massa-mola de 5 massas interligadas em série. Para isso, como indicado no enunciado foi a realizado a seguinte transformação:

$$X''(t) + AX(t) = 0 \Rightarrow Y''(t) + \Lambda Y(t) = 0$$
$$Y(t) = Q^T X(t)$$

Resolvendo a equação diferencial em Y, sendo “a” um autovalor,têm-se a seguinte solução:

$$Y_k(t) = Y_k(0)\cos\left(t\sqrt{a_k}\right)$$

Com isso,passando as condições iniciais de X para o domínio de Y,e em seguida multiplicando as soluções obtidas em Y por Q, pode-se obter as soluções em X.

Com isso, utilizando as condições iniciais de X(0) =(−2, −3, −1, −3, −1) e X(0) = (1, 10, −4, 3, −2), a pôde-se encontrar as soluções e X para cada massa. Para isso, a dupla implementou as seguintes funções:

- *make_A_matriz(n,k,m)*: Esta função recebe os parâmetros n - referente ao tamanho da matriz -, k - referente a constante de cada mola - e m - referente a massa das molas e ela retorna a matriz A. Segue abaixo sua implementação:

```
def make_A_matrix(n,k,m):
    A = np.zeros((n,n))
    for i in range(1,n):
        ki = k(i)
        ki_plus = k(i+1)
        A[i-1,i-1] = ki + ki_plus
        A[(i-1),(i-1)+1] = A[(i-1)+1,(i-1)] = -ki_plus
    A[n-1,n-1] = k(n) + k(n+1)
    A = A/m
    return A
```

- *solve_edo(t,y0, lamb)*:
  Essa função tem como objetivo obter o valor da Y na posição t do tempo. A EDO é resolvida usando a fórmula de Y logo acima. O seu retorno é o valor das 5 massas, em forma de vetor, na posição t do tempo.

```
def solve_edo(t,y0,lamb):
    y_t = lambda t,i,y0,lamb,: y0[i]*np.cos(np.sqrt(lamb[i,i])*t)
    y = np.zeros(shape = [len(y0),1], dtype = float)
    for i in range(0,len(y0)):
        y[i] = y_t(t,i,y0,lamb)
    return y
```

- *get_points(dt, x, y):*
  Essa função recebe o intervalo de cada função e retorna três vetores. Um deles (t_array) contém todos os pontos no tempo e X e Y, o segundo (x_array) retorna o valor de X nos respectivos pontos no tempo e o último (y_array) retorna um vetor valor de Y nos respectivos pontos no tempo.

```
def get_points(dt,x,y):
    t_array = np.array([])
    time = 10 #segundos

    color =  np.array([])

    n = len(y(0))
    y_array = np.zeros(shape = [n,1], dtype = float)
    x_array = np.zeros(shape = [n,1], dtype = float)

    for t in np.arange(0,time,dt):
        t_array = np.append(t_array,t)
        y_array = np.append(y_array,y(t),axis = 1)
        x_array = np.append(x_array,x(t),axis = 1)

    y_array = np.delete(y_array,0,1)
    x_array = np.delete(x_array,0,1)

    return [t_array,x_array,y_array]
```

- *plot_graphic(t_array, y_array, y_label = “frequency”, x_label = “time”, label = “x”):*
  Essa função representa os valores de X e Y em um determinado intervalo no tempo e representa eles em um gráfico. Isso fará que para cada massa haja um gráfico

```
def plot_graphics(t_array,y_array,y_label = 'frequency',x_label = 'time',label = "x"):
    color_array = np.array(["blue","green","red","cyan","magenta","yellow","black","blue","green","red","cyan","magenta","yellow","black"])
    quantity_graphics = len(y_array)
    fig, axs = plt.subplots(quantity_graphics)
    for i in range(0,quantity_graphics,1):
        axs[i].plot(t_array,y_array[i],label='{la}{it}'.format(it = i,la = label),color=color_array[i])
        axs[i].set_title('{la}{it}'.format(it = i,la = label))
        axs[i].set_ylabel(y_label)
        axs[i].set_xlabel(x_label)
    plt.show()
```

- *print_separado(t_array, x_array):*
  Essa função representa os valores de X e Y de cada massa em um determinado intervalo no tempo e representa eles em um gráfico. Isso fará que somente um gráfico seja representado.

```
def print_separado(t_array,x_array):
    color_array = np.array(["blue","green","red","cyan","magenta","yellow","black","blue","green","red","cyan","magenta","yellow","black"])
    quantity_graphics = len(x_array)
    for i in range(0,quantity_graphics,1):
        plt.xlim(min(t_array), max(t_array))
        plt.ylim(min(x_array[i]), max(x_array[i]))

        plt.plot(t_array,x_array[i],label='massa {a}'.format(a=i+1),color=color_array[i])
        plt.ylabel('x(t)')
        plt.xlabel('time(s)')

        plt.legend(loc='upper center', bbox_to_anchor=(0.5, 1.15), ncol=2, frameon=False)
        plt.show()
```

- *print_frequency(lamb, n)*: Esta função recebe como parâmetros a matriz lamb de autovalores e o valor n, referente ao tamanho da matriz. Ela visa printar a frequência e o período de cada massa. Sua implementação segue abaixo:

```
def print_frequency(lamb,n):
    frequencia = (np.sqrt(np.diagonal(lamb)))/(2*np.pi)
    periodo = (1/frequencia)
    for i in range(0,n):
        print('massa {a}: frequencia: {b} periodo: {c}'.format(a=i+1,b=frequencia[i],c=periodo[i]))
```

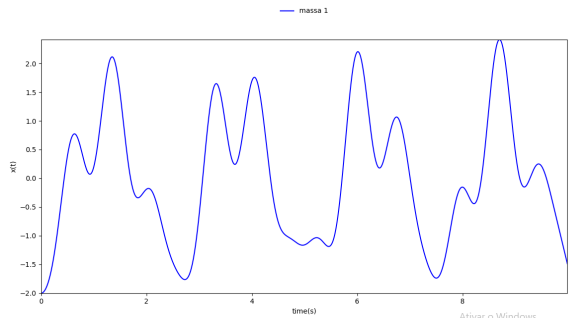
- *assignment_b()*: Essa função recebe os valores iniciais de x, que compõem a matriz X, tanto para a amostra 1, quanto para a amostra 2. A partir disso, ele gera a matriz A. A partir da matriz A, se faz a decomposição dela em seus autovalores e autovetores. A partir disso, calcula-se os valores iniciais de y que compõem a matriz Y. Sabendo os valores de Y e X, pode-se resolver a EDO de Y, que caracteriza o problema. Sabendo os valores de Y, pode-se obter os valores de X. Assim feito, há uma função para pegar todos os valores de Y e X em um determinado intervalo no tempo e colocá-los em um gráfico logo em seguida.

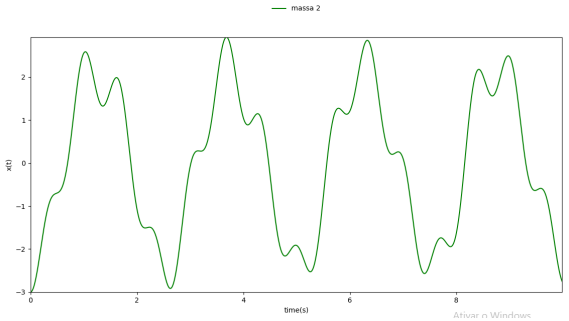
```python
def assignment_b():
    erro = 1e-6
    # dates
    m = 2 # kg
    n = 5 # numbers of mass
    k = lambda i: 40 + 2*i # k mola
    x_0_1 = np.array([[-2],[-3],[-1],[-3],[-1]])
    x_0_2 = np.array([[1],[10],[-4],[3],[-2]])

    A = make_A_matrix(n,k,m)
    Q,lamb = QR_algorithm(A,erro = erro,spectral_shift = True)[0:2]
    y_0_1 = np.transpose(Q)@x_0_1
    y_0_2 = np.transpose(Q)@x_0_2

    y_1 = lambda t: solve_edo(t,y_0_1,lamb)
    x_1 = lambda t: Q@solve_edo(t,y_0_1,lamb)

    y_2 = lambda t: solve_edo(t,y_0_2,lamb)
    x_2 = lambda t: Q@solve_edo(t,y_0_2,lamb)

    dt = 0.01

    print("amostra 1")
    t_array,x_array,y_array = get_points(dt,x_1,y_1)
    print_frequency(lamb,n)
    print("A massa com maior frequência é massa 1")
    escolha = input("amostra 1: Deseja ver todos graficos juntos(1) ou separado(2): ")
    if(escolha == "1"):
        plot_graphics(t_array,x_array,y_label = 'x(t)', x_label = 'time',label = "x")
        print("\n")
    else:
        print_separado(t_array,x_array)

    print("amostra 2")
    t_array,x_array,y_array = get_points(dt,x_2,y_2)
    print_frequency(lamb,n)
    print("A massa com maior frequência é massa 1")
    escolha = input("amostra 2: Deseja ver todos graficos juntos(1) ou separado(2): ")
    if(escolha == "1"):
        plot_graphics(t_array,x_array,y_label = 'x(t)', x_label = 'time',label = "x")
        print("\n")
    else:
        print_separado(t_array,x_array)
```

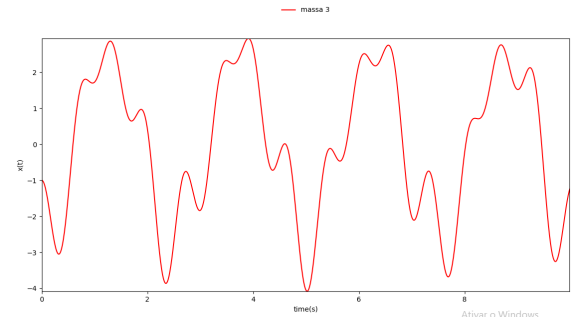
Frequências de cada uma das massas da amostra 1:

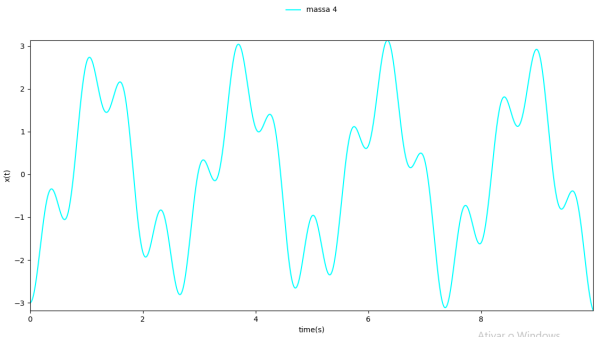
```
amostra 1
massa 1: frequencia: 1.4967759229962574 periodo: 0.6681026763165674
massa 2: frequencia: 1.332668035078792 periodo: 0.7503744417092458
massa 3: frequencia: 1.088475378527183 periodo: 0.9187162334834812
massa 4: frequencia: 0.7698664188411034 periodo: 1.2989266391243843
massa 5: frequencia: 0.3985026143191384 periodo: 2.5093938259565745
A massa com maior frequência é massa 1
```

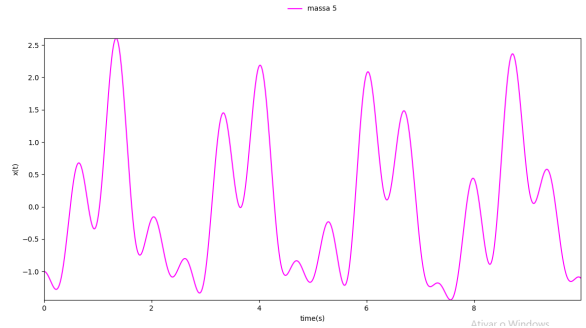
Ordenando da massa 1 a massa 5, seguem os gráficos da amostra 1*:*







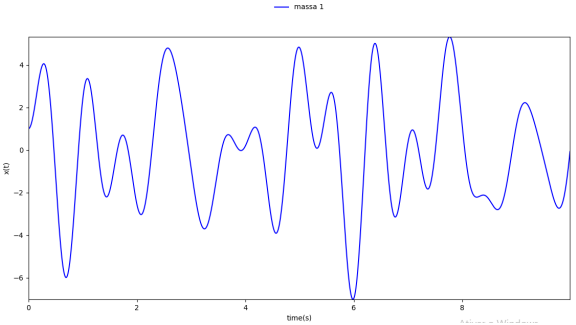


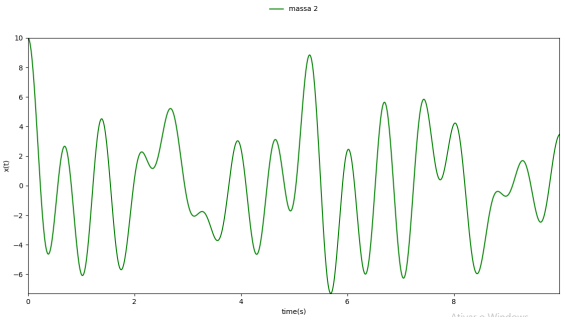


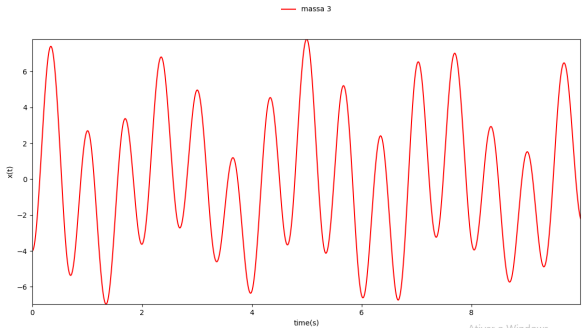
Frequências de cada uma das massas da amostra 2:

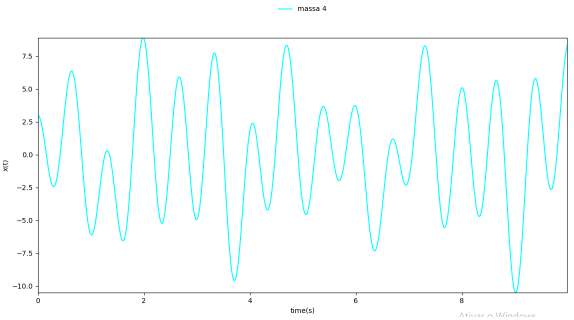
```
amostra 2
massa 1: frequencia: 1.4967759229962574 periodo: 0.6681026763165674
massa 2: frequencia: 1.332668035078792 periodo: 0.750374417092458
massa 3: frequencia: 1.088475378527183 periodo: 0.9187162334834812
massa 4: frequencia: 0.7698664188411034 periodo: 1.2989266391243843
massa 5: frequencia: 0.3985026143191384 periodo: 2.5093938259565745
A massa com maior frequência é massa 1
```

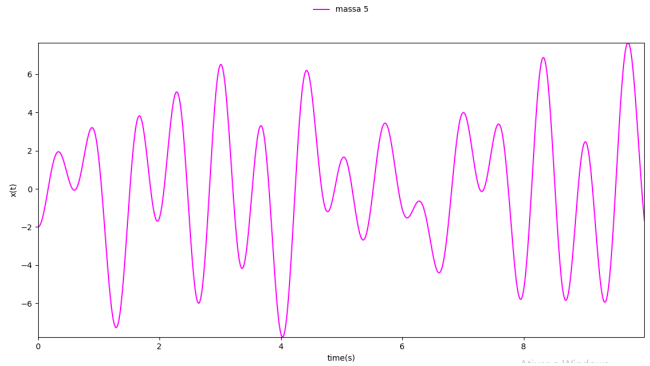
Ordenando da massa 1 a massa 5, seguem os gráficos da amostra 2:











Tarefa C

A tarefa C trata-se da mesma situação da tarefa B, porém com 10 massas interligadas ao invés de 5.

Com isso, para solucionar as equações foi adotada a mesma resolução da tarefa B e foi utilizado as posições iniciais X(0) =(−2, −3, −1, −3, −1, 2, −3, −1, −3, −1) e X(0) = (1, 10, −4, 3, −2, 1, 10, −4, 3, −2). Para isso, foi implementada a seguinte função:

- *assignment_c()*: Esta função segue os mesmos comandos da função *assignment_b(),* apenas alterando o valor de ‘n’, número de molas, assim como a adição de valores iniciais. Sua implementação segue abaixo:

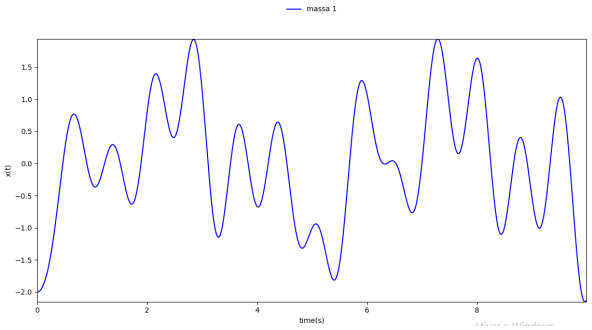
```python
def assignment_c():
    erro = 1e-6
    # dates
    m = 2 # kg
    n = 10 # numbers of mass
    k = lambda i: 40 + 2*((-1)**i) # k mola
    x_0_1 = np.array([[-2],[-3],[-1],[-3],[-1],[-2],[-3],[-1],[-3],[-1]])
    x_0_2 = np.array([[1],[10],[-4],[3],[-2],[1],[10],[-4],[3],[-2]])

    A = make_A_matrix(n,k,m)
    Q,lamb = QR_algorithm(A,erro = erro,spectral_shift = True)[0:2]
    y_0_1 = np.transpose(Q)@x_0_1
    y_0_2 = np.transpose(Q)@x_0_2

    y_1 = lambda t: solve_edo(t,y_0_1,lamb)
    x_1 = lambda t: Q@solve_edo(t,y_0_1,lamb)

    y_2 = lambda t: solve_edo(t,y_0_2,lamb)
    x_2 = lambda t: Q@solve_edo(t,y_0_2,lamb)

    dt = 0.01

    print("amostra 1")
    t_array,x_array,y_array = get_points(dt,x_1,y_1)
    print_frequency(lamb,n)
    escolha = input("amostra 1: Deseja ver todos graficos juntos(1) ou sepado(2): ")
    if(escolha == "1"):
        plot_graphics(t_array,x_array,y_label = 'x(t)', x_label = 'time',label = "x")
        print("\n")
    else:
        print_separado(t_array,x_array)
```

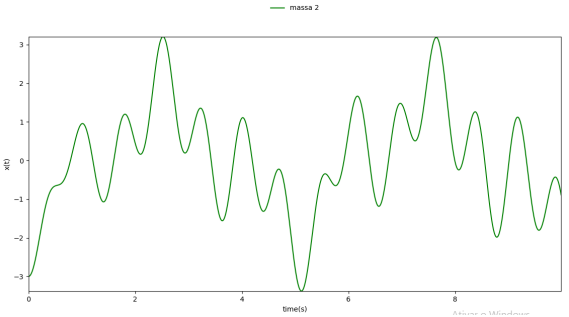
```python
        print_separado(t_array,x_array)

    print("amostra 2")
    t_array,x_array,y_array = get_points(dt,x_2,y_2)
    print_frequency(lamb,n)
    escolha = input("amostra 2: Deseja ver todos graficos juntos(1) ou sepado(2): ")
    if(escolha == "1"):
        plot_graphics(t_array,x_array,y_label = 'x(t)', x_label = 'time',label = "x")
        print("\n")
    else:
        print_separado(t_array,x_array)

if __name__ == "__main__":
    assignment_c()
```

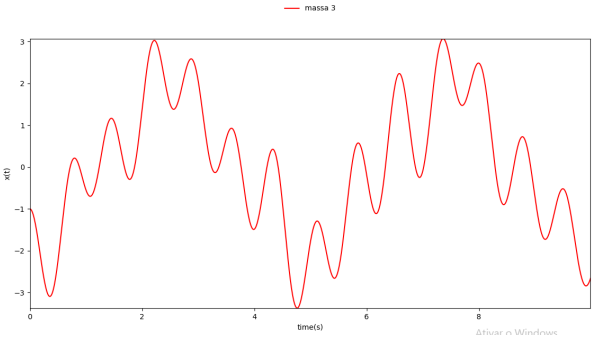
Frequência de cada uma das massas na amostra 1:

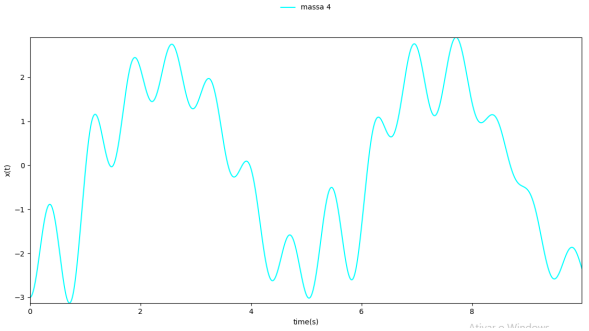
```
amostra 1
massa 1: frequencia: 1.4092066849093787 periodo: 0.7096191145760188
massa 2: frequencia: 1.3666019505229223 periodo: 0.7317419674524508
massa 3: frequencia: 1.2968321197779809 periodo: 0.7711098335312678
massa 4: frequencia: 1.202267342201418 periodo: 0.8317617595508705
massa 5: frequencia: 1.0911272467327295 periodo: 0.9164833918265712
massa 6: frequencia: 0.9142565322803213 periodo: 1.0937849112281637
massa 7: frequencia: 0.7622184157154192 periodo: 1.3119599046441286
massa 8: frequencia: 0.587069093798034 periodo: 1.7033770139908342
massa 9: frequencia: 0.20139559823247172 periodo: 4.965351818889786
massa 10: frequencia: 0.39852576036399406 periodo: 2.5092480824492966
A massa com maior frequência é massa 1
```

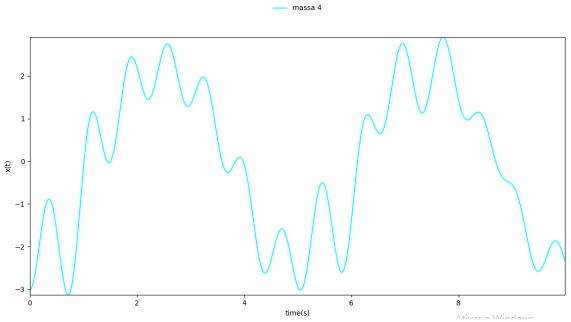
Ordenando da massa 1 a massa 10, seguem os gráficos da amostra 1:

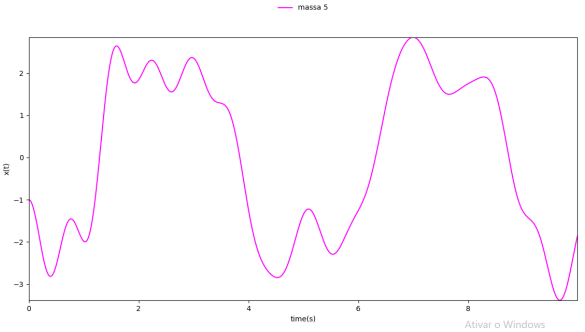

massa 1
x(t)
time(s)

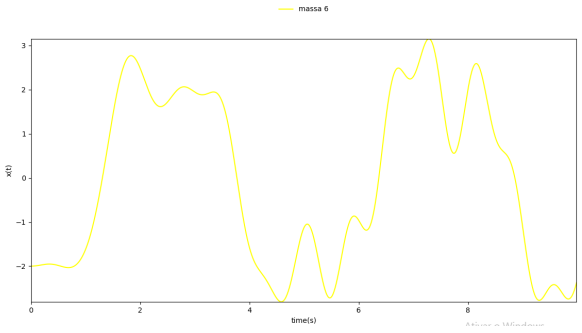

massa 2
x(t)
time(s)

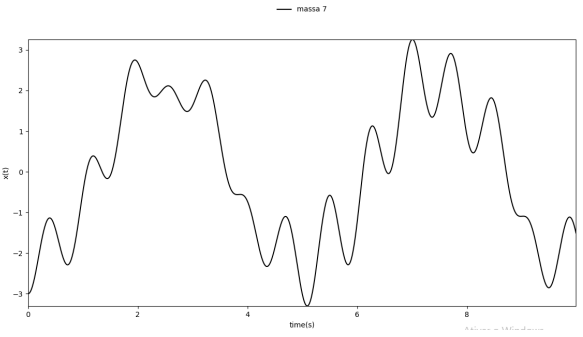

massa 3
x(t)
time(s)

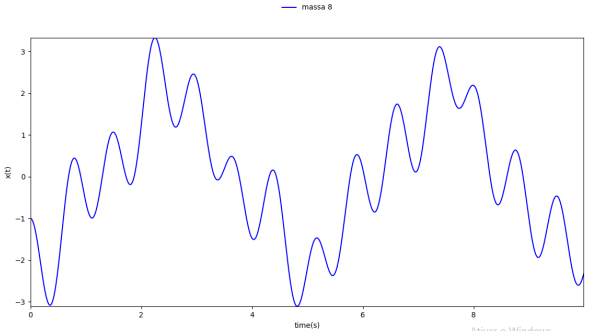

massa 4
x(t)
time(s)

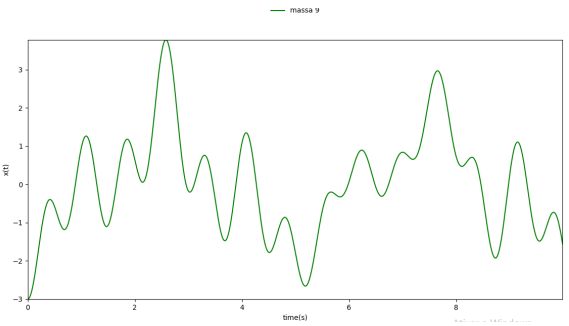

massa 4
x(t)
time(s)

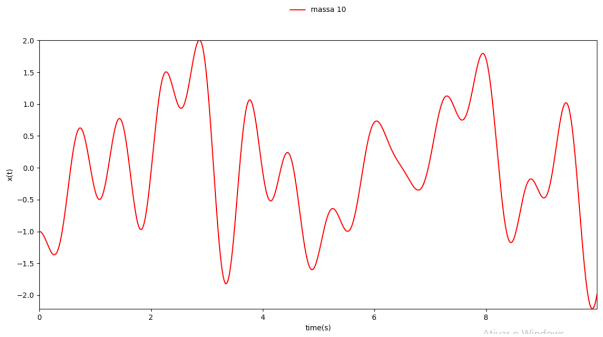

massa 5
x(t)
time(s)


massa 6
x(t)
time(s)


massa 7
x(t)
time(s)


massa 8
x(t)
time(s)

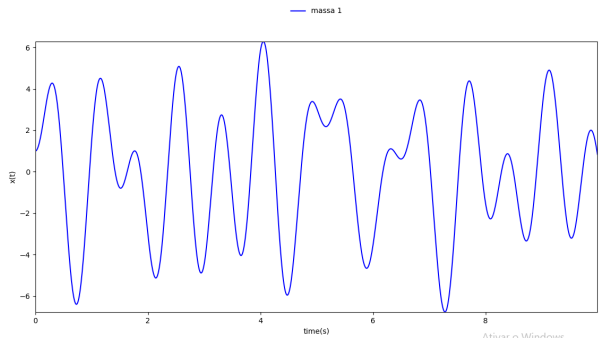

massa 9
x(t)
time(s)

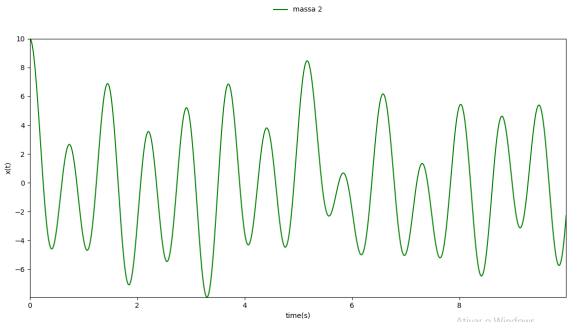


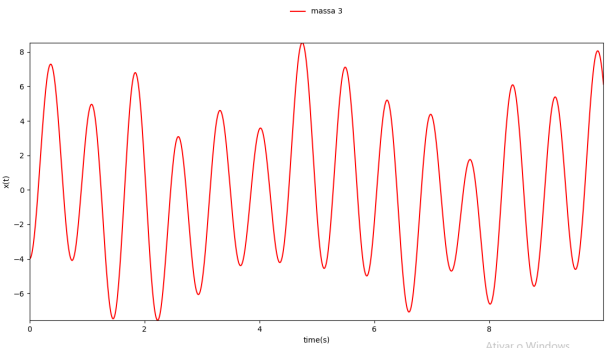
Frequência de cada uma das massas na amostra 2:

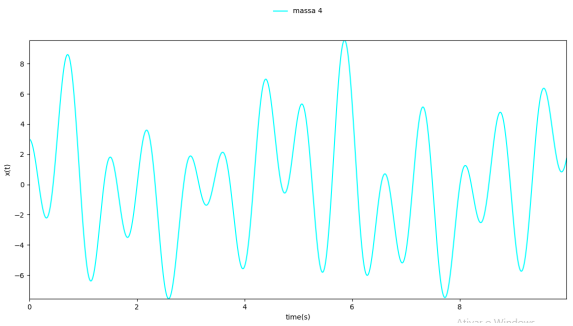
```
amostra 2
massa 1: frequencia: 1.4092066849093787 periodo: 0.7096191145760188
massa 2: frequencia: 1.3666019505229223 periodo: 0.7317419674524508
massa 3: frequencia: 1.2968321197779809 periodo: 0.7711098335312678
massa 4: frequencia: 1.202267342201418 periodo: 0.8317617595508705
massa 5: frequencia: 1.0911272467327295 periodo: 0.9164833918265712
massa 6: frequencia: 0.9142565322803213 periodo: 1.0937849112281637
massa 7: frequencia: 0.7622184157154192 periodo: 1.3119599046441286
massa 8: frequencia: 0.587069093798034 periodo: 1.7033770139908342
massa 9: frequencia: 0.20139559823247172 periodo: 4.965351818889786
massa 10: frequencia: 0.39852576036399406 periodo: 2.5092480824492966
A massa com maior frequência é massa 1
```

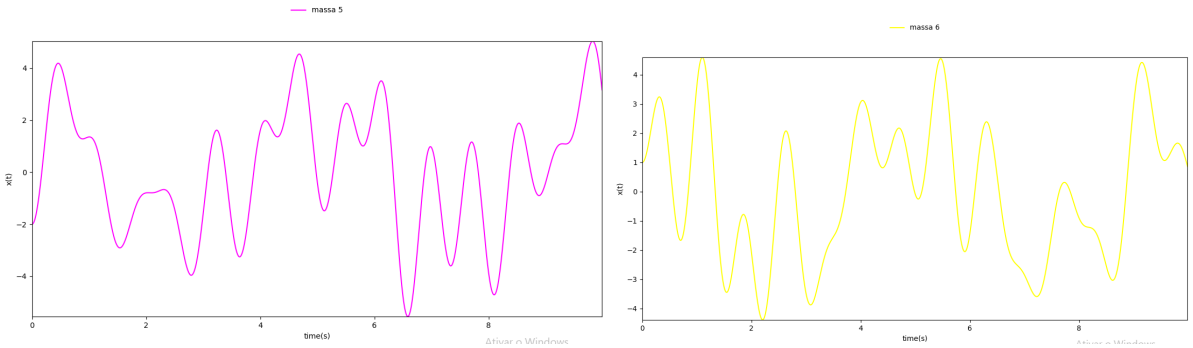
Ordenando da massa 1 a massa 10, seguem os gráficos da amostra 2:

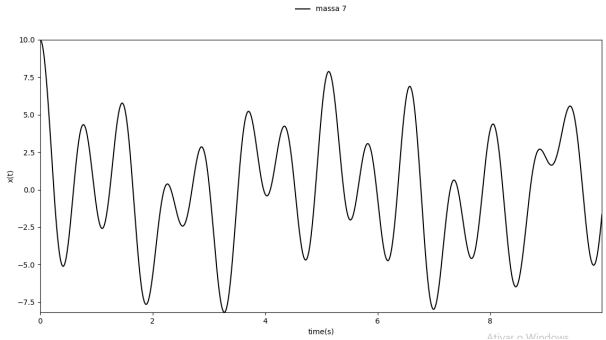


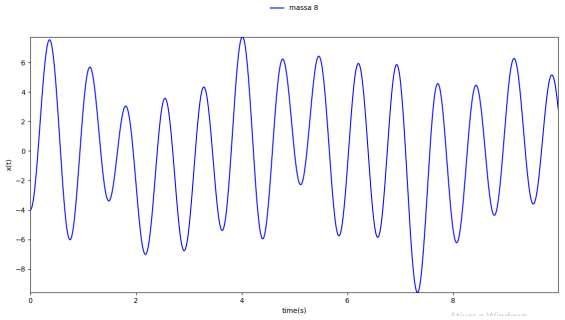


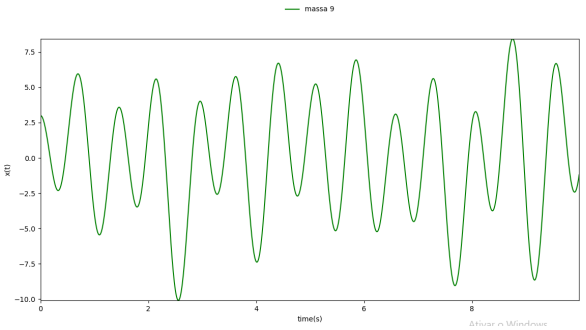


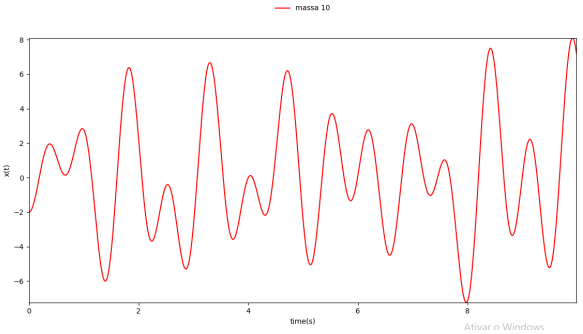












Conclusão

Ao realizar o presente experimento, a dupla pôde se debruçar sobre a utilização de métodos numéricos para a solução de problemas. Ao utilizar o Algoritmo QR ao sistema massa mola em série, foi possível solucionar uma equação diferencial de forma mais simples, o que traz a motivação de usar tal método para outras possíveis soluções de engenharia.

Ao desenvolver, executar e aplicar o algoritmo, a dupla realizou testes e comparou com os valores esperados, podendo concluir que se obteve na realização do trabalho. Entretanto, a dupla não obteve a relação do erro entre o valor analítico e do valor numérico, o qual poderia ser útil para analisar a precisão do método e, com isso, averiguar em quais situações sua utilização é viável.

Ao fim, ao trabalhar intensamente com uma linguagem de programação, foi percebido ainda mais sua importância para o desenvolvimento de softwares que visam facilitar a vida acadêmica e da sociedade em geral.