Escola Politécnica da Universidade de São Paulo

Exercício Programa de Cálculo Numérico (MAP3121)

Gustavo Santos da Silva	10432387
Eduardo Tadashi Asato	10823810

São Paulo
2021

Gustavo Santos da Silva
Eduardo Tadashi Asato

Exercício Programa de Cálculo Numérico

Relatório apresentado como requisito para avaliação na disciplina de MAP3121 - Métodos Numéricos e Aplicações, no curso de Engenharia Elétrica oferecido pela Escola Politécnica da Universidade de São Paulo.

São Paulo
2021

SUMÁRIO

1. Introdução

O exercício programa busca implementar o algoritmo QR com deslocamento espectral a fim de calcular os autovalores e autovetores de uma matriz simétrica n x n. Para tal, diversos conceitos envolvendo cálculo numérico e estratégias de programação, são utilizados. Dentre eles as rotações de Givens.

Desse modo, a partir do projeto desenvolvido, busca-se também verificar a resolução da evolução de um sistema massa-mola, esboçando um gráfico que mostre os deslocamentos de cada mola em relação a sua posição de equilíbrio.

Para a resolução, buscou-se compreender o algoritmo QR, através de múltiplos testes, verificando de forma conveniente a efetividade do código. Nesse sentido, comparações entre a solução analítica e pelo código foram efetuadas.

Para o presente exercício programa, a dupla optou pela elaboração em Python 3.7, por contar com recursos de domínio do grupo.

2. Desenvolvimento

2.1 Primeira Tarefa: Cálculo de autovalores e autovetores para matriz tridiagonal

2.1.1 O método QR

Antes de definir o método QR, deve-se definir a matriz tridiagonal simétrica, que é da forma como está na figura 2.1.1.1.

$$\begin{bmatrix} \alpha_0 & \beta_0 & 0 & \dots \\ \beta_0 & \alpha_1 & \beta_1 & \dots \\ 0 & \beta_1 & \alpha_2 & \dots \\ \vdots & \vdots & \vdots & \ddots \end{bmatrix}$$

Figura 2.1.1.1

O método QR é baseado na decomposição QR, que recebe uma matriz tridiagonal simétrica, A, e retorna Q, uma matriz ortogonal, e R, uma matriz triangular superior, como na figura 2.1.1.2.

$$A = QR$$

Figura 2.1.1.2

Se multiplicarmos a equação anterior por Q, pela direita, e por Q invertida, pela esquerda, obtém-se um novo A'. Esse novo A' obtido também é uma matriz tridiagonal simétrica, mas com os elementos fora da diagonal principal menores.

Se isso for feito iterativamente, só sobram elementos na diagonal principal, que serão exatamente os autovalores da matriz A, e Q será um vetor cujas colunas são os autovetores.

Esta matriz cujos elementos na diagonal principal são os autovalores de A será chamada de Λ, e a matriz com os autovetores, será Q, então obtém-se o que está na figura 2.1.1.4.

$$Q^{-1}QRQ = Q^TQRQ = Q^TAQ = A'$$

Figura 2.1.1.3

$$A = Q^T\Lambda Q$$

Figura 2.1.1.4

Neste método, como o último β converge mais rápido para zero, então uma condição de convergência pode ser criada: toda vez que o último β ficar menor do que uma certa tolerância, em módulo, pode-se dizer que o último α convergiu para

um dos autovalores, então pode-se passar a trabalhar sobre uma submatriz, para encontrar o próximo autovalor, e assim por diante.

Para melhorar o algoritmo, podem ser feitos os chamados deslocamentos espectrais, subtraindo da diagonal principal um valor próximo de um dos autovalores da matriz antes de fazer a decomposição QR, e, depois de fazer a remontagem da matriz $A^{(k+1)}$, somar de volta. Assim diminuem-se significativamente as iterações necessárias para atingir convergência.

O algoritmo está na forma de pseudocódigo na figura 2.1.1.5, extraído do documento do exercício programa. A linha 6 do pseudocódigo corresponde à decomposição QR, a linha 7, à remontagem, a linha 8, à atualização dos autovetores.

A implementação será detalhada a seguir no documento de forma abstraída, mas, depois, na forma de código.

Algoritmo QR tridiagonal com deslocamento

1: Sejam $A^{(0)} = A \in \mathbb{R}^{n \times n}$ uma matriz tridiagonal simétrica, $V^{(0)} = I$ e $\mu_0 = 0$. O algoritmo calcula 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: $V = V^{(k)}$

Figura 2.1.1.5 - Método QR na forma de pseudocódigo (com a linha 3 modificada)

2.1.1.1 A decomposição QR

Para implementar a decomposição QR foram utilizadas multiplicações sucessivas por matrizes de rotações de Givens, que resultam nas matrizes $Q^{(k)}$ e $R^{(k)}$, como mostra a figura 2.1.1.1.3. Uma rotação de Givens é uma matriz da forma como na figura 2.1.1.1.1, com os seus senos e cossenos calculados usando a forma estável, na figura 2.1.1.1.2.

$$Q_i^{(k)} = [q_{IJ}]$$

$$q_{IJ} = \begin{cases} c_i, \text{ se } I = J = i \vee I = J = i+1 \\ s_i, \text{ se } I = i+1 \wedge J = i \\ -s_i, \text{ se } I = i \wedge J = i+1 \\ 0, \text{ caso contrário} \end{cases}$$

Figura 2.1.1.1.1

$$\text{Se } |\alpha_i| > |\beta_i|, \text{ então } \tau = -\beta_i/\alpha_i, \quad c_i = \frac{1}{\sqrt{1+\tau^2}}, \quad s_i = c_i\tau$$

$$\text{Caso contrário, então } \tau = -\alpha_i/\beta_i, \quad s_i = \frac{1}{\sqrt{1+\tau^2}}, \quad c_i = s_i\tau$$

Figura 2.1.1.1.2

$$R^{(k)} = Q_{n-1}^{(k)} \dots Q_0^{(k)} A^{(k)} = Q^{(k)} A^{(k)}$$

Figura 2.1.1.1.3

Como o método QR é iterativo, e ele realiza a decomposição QR, que também é iterativa, as iterações da decomposição QR, por convenção, serão chamadas de subiterações *i*, e as do método como um todo, de iterações *k*.

Para a aplicar os deslocamentos espectrais, é só subtrair μ_k da diagonal principal de $A^{(k)}$ antes de fazer a decomposição QR.

2.1.1.1.1 A decomposição QR, otimizações na implementação

A primeira otimização tem a ver com os senos e cossenos das rotações de Givens: como será necessário remontar a matriz tridiagonal depois de fazer a decomposição QR, então será necessário armazenar o valor dos senos e cossenos das rotações de Givens das subiterações durante cada iteração, mas não precisam ser armazenados entre iterações, então dois vetores de tamanho *n* podem ser usados para armazenar estes valores. Não é necessário armazenar as matrizes $Q_i^{(k)}$ inteiras de cada subiteração.

A segunda otimização tem a ver com o próximo passo depois da decomposição QR, que é a remontagem da matriz tridiagonal $A^{(k+1)}$: como $A^{(k+1)}$ só tem valores nas três diagonais centrais, não é necessário calcular valores fora delas, então a otimização é calcular só as três diagonais centrais de $R^{(k)}$. Portanto, foi possível armazenar a matriz de trabalho usando somente três vetores, um de tamanho máximo n, e dois de tamanho máximo n-1.

As últimas duas otimizações têm a ver com as multiplicações sucessivas por matrizes de rotações de Givens: para ilustrar estas otimizações, considere a

situação inicial, em que é feita a primeira multiplicação por matriz de rotações de Givens sobre a matriz da zero-ésima iteração, sem deslocamentos espectrais.

Durante a fase de planejamento, uma multiplicação foi feita usando variáveis simbólicas em *matlab* e o resultado está na figura 2.1.1.1.1.1. Como mostra a figura, somente as linhas 0 e 1 foram alteradas, mas como só interessa calcular os elementos nas três diagonais centrais, então só é necessário alterar cinco elementos, por subiteração, um dos quais, o *beta0*c+alfa0*s*, é zerado, então só é necessário calcular quatro elementos durante cada subiteração da decomposição QR otimizada.

Agora generalizando para uma subiteração i: uma subiteração otimizada da decomposição QR está na figura 2.1.1.1.1.2, claro o último cálculo desta figura não é realizado na última subiteração de uma iteração k.

Depois de realizar o loop que faz decomposição QR, no programa, as três diagonais centrais de $R^{(k)}$ estarão nos vetores *Diagonal principal*, *Subdiagonal Inferior* e *Subdiagonal Superior*, e os senos e cossenos das subiterações da decomposição QR estão nos vetores *c* e *s*.

```
A =

[ alfa0, beta0,     0,     0]
[ beta0, alfa1, beta1,     0]
[     0, beta1, alfa2, beta2]
[     0,     0, beta2, alfa3]

>> Q0 = [c -s 0 0; s c 0 0; 0 0 1 0; 0 0 0 1]

Q0 =

[ c, -s, 0, 0]
[ s,  c, 0, 0]
[ 0,  0, 1, 0]
[ 0,  0, 0, 1]

>> Q0*A

ans =

[ alfa0*c - beta0*s, beta0*c - alfa1*s, -beta1*s,     0]
[ beta0*c + alfa0*s, alfa1*c + beta0*s,  beta1*c,     0]
[                 0,             beta1,    alfa2, beta2]
[                 0,                 0,    beta2, alfa3]
```

Figura 2.1.1.1.1.1

(Diagonal Principal da Próxima Subiteração)(i) = c(i) (Diagonal Principal da Subiteração Atual)(i) - s(i)(Subdiagonal Inferior da Subiteração Atual)(i)

(Diagonal Principal da Próxima Subiteração)(i+1) = s(i) (Subdiagonal Superior da Subiteração Atual)(i) + c(i)(Diagonal Superior da Subiteração Atual)(i+1)

(Subdiagonal Inferior da Próxima Subiteração)(i) = 0

(Subdiagonal Superior da Próxima Subiteração)(i) = c(i) (Subdiagonal Superior da Subiteração Atual)(i) - s(i)(Diagonal Principal da Subiteração Atual)(i+1)

(Subdiagonal Superior da Próxima Subiteração)(i+1) = c(i) (Subdiagonal Superior da Subiteração Atual)(i+1)

Figura 2.1.1.1.1.2

2.1.1.2 A remontagem de A(k+1)

Agora que a decomposição QR foi realizada, a remontagem de $A^{(k+1)}$ pode ser começada. Também será feita de forma iterativa, e, também, suas iterações são chamadas de subiterações, por convenção. Mas agora as matrizes de rotações de Givens são multiplicadas pela direita, invertidas, que, como elas são ortogonais, é a mesma coisa que multiplicar pelas matrizes transpostas.

2.1.1.2.2 A remontagem de A(k+1), otimizações na implementação

A primeira otimização, que já foi citada no bloco anterior, é a de calcular somente os valores das três diagonais centrais, mas aqui é possível aplicar mais uma otimização: como a matriz $A^{(k+1)}$ termina simétrica, então, em cada subiteração, é possível calcular somente os valores para a diagonal principal e subdiagonal inferior, e, ao fim de uma subiteração, copiar os valores para a subdiagonal superior, economizando tempo de execução.

Aqui também é possível aplicar a otimização de só calcular os elementos afetados pela matriz de rotação de Givens, então só é necessário calcular três valores, como mostra a figura 2.1.1.2.2.1.

Depois de obter $A^{(k+1)}$, se for necessário aplicar deslocamentos espectrais, só somar à diagonal principal o μ_k.

(Diagonal Principal da Próxima Subiteração)(i) = c(i)(Diagonal Principal da Subiteração Atual)(i) - s(i)(Subdiagonal Superior da Subiteração Atual)(i)

(Diagonal Principal da Próxima Subiteração)(i+1) = c(i)(Diagonal Principal da Subiteração Atual)(i+1)

(Subdiagonal Inferior da Próxima Subiteração)(i) = -s(i)(Diagonal Principal da Subiteração Atual)(i+1)

Figura 2.1.1.2.2.1

2.1.1.3 Atualização da matriz de Autovetores

A atualização dos autovetores consiste em multiplicar o Q da iteração atual com o da iteração anterior, iniciando com a matriz identidade.

Esta parte também foi implementada de forma iterativa, e também foram aplicadas otimizações para somente calcular os valores necessários, os detalhes da otimização estão mais à frente no documento.

2.1.2 Implementação: O Código

No programa criado pelo grupo, a aplicação do método QR é realizada pela classe singleton 'CalculaAutovalsAutovecs', através de seu método público *AutovalAutovec(alfa, beta, epsilon, deslocamentos, V)*.

2.1.2.1 Implementação: a classe *CalculaAutovalsAutovec*:

Esta classe possui os seguintes métodos:

- *AutovalAutovec(alfa, beta, epsilon, deslocamentos, V)*: é o método que faz o cálculo de fato dos autovalores e autovetores da matriz A.

  Seus argumentos são: alfa, vetor numpy que armazena a diagonal principal da matriz cujos autovalores e autovetores se quer calcular; beta, vetor numpy que armazena a subdiagonal imediatamente acima ou abaixo de alfa; epsilon, float que indica a condição de convergência; deslocamentos, booleano que indica para o método se devem ser usados deslocamentos espectrais; V, matriz numpy que serve como condição inicial para o cálculo dos autovetores.

  Retorna uma tupla *(Diagonal_principal, temp_V, k)*, em que *Diagonal_principal* é um vetor numpy com os autovalores, *temp_V* é a matriz cujas colunas são os autovetores, e 'k', que é o número de iterações que foram necessárias para convergir;
- __calcula_c, __calcula_s, __calcula_mu e __normaliza são métodos privados auxiliares que retornam, respectivamente, um dos cossenos das rotações de Givens, um dos senos das rotações de Givens, o μk para fazer deslocamentos espectrais, o vetor normalizado.

2.1.2.2 Implementação: Detalhes da classe *CalculaAutovalsAutovec*:

O método QR com deslocamentos espectrais foi implementado com um algoritmo que é composto por vários loops aninhados. Como convencionado anteriormente, as iterações do loop externo, são chamadas de iterações, mas as iterações dos três loops internos que terminam com as três diagonais centrais de $R^{(k)}$ e de $A^{(k+1)}$ e com a matriz $V^{(k+1)}$ são chamadas de subiterações.

Fora do loop externo, existe um preâmbulo (destacado no trecho de código da figura 2.1.2.2.1) com a instanciação de alguns vetores *numpy*: *c* e *s*, que armazenam os valores de *c[i]* e de *s[i]* para cada subiteração *i* da decomposição QR, para uma dada iteração *k* do loop externo. Como a matriz da rotação de Givens tem só dois valores, então foi possível armazenar esses valores em apenas dois vetores.

Neste programa só são armazenados os valores de *c[i]* e *s[i]* para a iteração atual, como citado no item 2.1.1.1.1.

```
c = np.zeros(alfa.shape[0] - 1)  # Vetor que armazena os valores de c em cada subiteração i da decomposição QR da k-ésima iteração
s = np.zeros(alfa.shape[0] - 1)  # Vetor que armazena os valores de s em cada subiteração i da decomposição QR da k-ésima iteração
mu = 0
```

Figura 2.1.2.2.1

Para os vetores das três linhas de código destacadas na figura 2.1.2.2.2, a função destes vetores muda de acordo com o loop que o programa está: podem armazenar a diagonal principal, e as duas subdiagonais imediatamente acima e abaixo da primeira de $R^{(k)}$ ou de $A^{(k+1)}$, ao final dos seus respectivos loops.

Apesar de a matriz $R^{(k)}$ não ser tridiagonal, quando ela é multiplicada à direita por $Q^{(k)T}$, para virar $A^{(k+1)}$, ela se torna tridiagonal, então não é necessário calcular os valores para fora dessas 3 diagonais, mais uma otimização, como citado no item .2.1.1.1.1.

```
# De acordo com o for, esses 3 vetores mudam de função
Diagonal_principal = np.copy(alfa)
Subdiagonal_superior = np.copy(beta)
Subdiagonal_inferior = np.copy(beta)
```

Figura 2.1.2.2.2

Uma próxima linha de código importante é a destacada na figura 2.1.2.2.3, que declara a variável ‘m’.

À medida que iterações do método QR são realizadas, o último valor das subdiagonais vai diminuindo em módulo, e, quando o módulo fica menor do que o parâmetro *epsilon*, dizemos que o último valor da diagonal principal convergiu para um dos autovalores da matriz.

Então neste programa, a variável ‘m’ indica até que índice a decomposição QR deve rodar na matriz. Assim o loop da decomposição QR roda somente numa submatriz m+1 por m+1, e ignora o resto da matriz, onde os autovalores já convergiram.

```
m = alfa.shape[0] - 1
```

Figura 2.1.2.2.3

As linhas de código abaixo são o loop externo, cuja condição de parada é m>=0, para impedir ele de ficar executando para sempre. E o if que só vai executar os três loops internos se a submatriz não puder ser reduzida, ou seja, se *m* não puder diminuir.

```
while (m >= 0):

    # Estes quatro vetores temp são usados como temporários para armazenar as células da
    # matriz que são atualizadas entre subiterações dos três fors
    temp_diagonal_principal = np.zeros(2)
    temp_subdiagonal = np.zeros(2)

    # Quando multiplicamos o subresultado de V(k+1) por Qi(k), só são modificadas
    # duas colunas do subresultado, aqui estão os temporários das duas colunas modificadas
    temp_V_coluna_esquerda = np.zeros(alfa.shape[0])
    temp_V_coluna_direita = np.zeros(alfa.shape[0])

    # O if só é executado se a submatriz é pelo menos 2x2 e o último beta não puder ser zerado
    if (m > 0 and abs(Subdiagonal_inferior[m - 1]) >= epsilon):
        #[...]
```

Figura 2.1.2.2.4

Para k>0, ou seja, a partir da 2ª iteração, o valor de μ_k é calculado, caso esteja na 1ª iteração o valor é 0.

```
if (k > 0):
    mu = self.__calcula_mu(Diagonal_principal, Subdiagonal_inferior, m, deslocamentos)
    Diagonal_principal = Diagonal_principal - mu
```

Figura 2.1.2.2.5

O loop da figura 2.1.2.2.6 faz a decomposição QR, então implementa as otimizações citadas no item 2.1.1.1.1.

```
for i in range(0, m):
    # Calcula c e s da subiteração i da decomposição QR #Otimização 1
    c[i] = self.__calcula_c(Diagonal_principal, Subdiagonal_inferior, i)
    s[i] = self.__calcula_s(Diagonal_principal, Subdiagonal_inferior, i)

    # Calcula os novos valores das duas células atualizadas da diagonal principal do subresultado de R(k)
    temp_diagonal_principal[0] = c[i] * Diagonal_principal[i] - s[i] * Subdiagonal_inferior[i]
    temp_diagonal_principal[1] = s[i] * Subdiagonal_superior[i] + c[i] * Diagonal_principal[i + 1]

    # Calcula os novos valores das duas células atualizadas da 1° subdiagonal acima da diagonal principal do subresultado de R(k)
    temp_subdiagonal[0] = c[i] * Subdiagonal_superior[i] - s[i] * Diagonal_principal[i + 1]
    if (i != m - 1):
        # Não existe Subdiagonal_superior[i+1] na última subiteração, por isso este if
        temp_subdiagonal[1] = c[i] * Subdiagonal_superior[i + 1]

    # Atualiza o subresultado com os valores temporários
    Subdiagonal_inferior[i] = 0.0  # Otimização 2
    Diagonal_principal[i] = np.copy(temp_diagonal_principal[0])
    Diagonal_principal[i + 1] = np.copy(temp_diagonal_principal[1])
    Subdiagonal_superior[i] = np.copy(temp_subdiagonal[0])

    if (i != m - 1):
        Subdiagonal_superior[i + 1] = np.copy(temp_subdiagonal[1])
```

Figura 2.1.2.2.6

O loop da figura 2.1.2.2.7 faz a remontagem da matriz tridiagonal, então implementa as otimizações citadas no item 2.1.1.1.2.2.

```
# Este for faz o cálculo de A(k+1)
#
# Este for realiza m subiterações i
# A cada subiteração o subresultado de A(k+1) é multiplicado por Qi(k)T
# Até no final ter a própria A(k+1)
#
# Neste for
# Diagonal_principal: Vetor que armazena a diagonal principal do subresultado de A(k+1), e. no final do for, da própria A(k+1)
# Subdiagonal_superior: Vetor que armazena a 1° subdiagonal acima da diagonal principal do subresultado de A(k+1), e. no final do for, da própria A(k+1)
# Subdiagonal_inferior: Vetor que armazena a 1° subdiagonal abaixo da diagonal principal do subresultado de A(k+1), e. no final do for, da própria A(k+1)
#
# Otimizações implementadas neste for:
# 1. Como a matriz A(k+1) termina simétrica, então só calculamos o valor para a diagonal imediatamente abaixo da principal
# 2. Como as multiplicações por Qi(k)T só modificam duas colunas do subresultado de A(k+1), então só é necessário calcular 3 células do subresultado
for i in range(0, m):
    # Calcula os novos valores das duas células atualizadas da diagonal principal do subresultado de A(k+1)
    temp_diagonal_principal[0] = c[i] * Diagonal_principal[i] - s[i] * Subdiagonal_superior[i]
    temp_diagonal_principal[1] = c[i] * Diagonal_principal[i + 1]

    # Calcula o novo valos da célula atualizada da 1° subdiagonal abaixo da diagonal principal do subresultado de A(k+1)
    temp_subdiagonal[0] = -s[i] * Diagonal_principal[i + 1]  # Otimização 1

    # Atualiza o subresultado com os valores temporários
    Diagonal_principal[i] = np.copy(temp_diagonal_principal[0])
    Diagonal_principal[i + 1] = np.copy(temp_diagonal_principal[1])
    Subdiagonal_inferior[i] = np.copy(temp_subdiagonal[0])
    Subdiagonal_superior[i] = np.copy(temp_subdiagonal[0])  # Como A(k+1) é simétrica, não precisamos calcular a subdiagonal superior #Otimização 1

# Faz A(k+1) = R(k)*Q(k)+muk*I
if (k > 0):
    Diagonal_principal = Diagonal_principal + mu
```

Figura 2.1.2.2.7

No último loop, $V^{(k+1)}$ é calculada, também aplicando a otimização de apenas calcular os valores necessários, como feito nos outros 2 loops.

```
# Este for faz a multiplicação V(k+1)=V(k)*Q(k)T, e armazena o resultado em temp_V
#
# Este for realiza m subiterações subiteracao
# A cada subiteração o subresultado de V(k+1) é multiplicado por Qi(k)T
# Até no final ter a própria V(k+1)
#
# Otimizações implementadas neste for:
# 1. Como as multiplicações por Qi(k)T só modificam duas colunas do subresultado de V(k+1), então só é necessário calcular 2 colunas do subresultado
for subiteracao in range(0, m):
    for linha in range(0, alfa.shape[0]):
        temp_V_coluna_esquerda[linha] = temp_V[linha, subiteracao] * c[subiteracao] - temp_V[linha, subiteracao + 1] * s[subiteracao]
        temp_V_coluna_direita[linha] = temp_V[linha, subiteracao + 1] * c[subiteracao] + temp_V[linha, subiteracao] * s[subiteracao]

        temp_V[linha, subiteracao] = temp_V_coluna_esquerda[linha]
        temp_V[linha, subiteracao + 1] = temp_V_coluna_direita[linha]
```

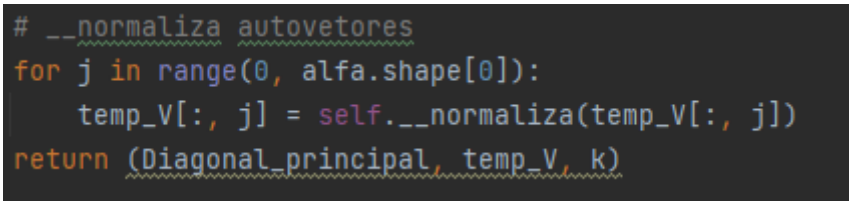
Figura 2.1.2.2.8

Saindo do escopo do if, mas ainda dentro do escopo do while, o código abaixo reduz a submatriz caso seja possível

```
# Este if faz a eliminação de beta se este for menor que epsilon e diminui o escopo (m=m-1)
# do algoritmo, para que ele passe a trabalhar com a submatriz
if (abs(Subdiagonal_inferior[m - 1]) < epsilon):
    Subdiagonal_inferior[m - 1] = 0.0
    Subdiagonal_superior[m - 1] = 0.0
    m = m - 1
```

Figura 2.1.2.2.9

Fora do escopo do while, o código abaixo normaliza os autovetores e retorna a tupla.



```
# __normaliza autovetores
for j in range(0, alfa.shape[0]):
    temp_V[:, j] = self.__normaliza(temp_V[:, j])
return (Diagonal_principal, temp_V, k)
```

Figura 2.1.2.2.10

2.1.2.2 Implementação: a classe EstimadorDeErros**:**

É a classe encarregada de imprimir na tela estimativas de erro. Ela possui dois métodos públicos:

- *EstimativasErroGerais(n, alfa, beta, Lambda, V)*: que calcula estimativas de erro para o caso geral (aquele que não possui solução analítica).
  Ele recebe os argumentos n, alfa, beta, Lambda, V, que são, respectivamente, o tamanho da matriz cujos autovalores foram determinados; o vetor alfa, contendo os valores da diagonal principal da matriz original; o vetor beta, contendo os valores de uma das subdiagonais da matriz original; o vetor Lambda, contendo os autovalores; e V, matriz com os autovetores nas colunas.
  Não retorna nada, mas imprime na tela as estimativas de erro.
- *EstimativasErroAnalitico(n, Lambda, V)*: que calcula estimativas de erro para o caso com solução analítica.
  Recebe os argumentos n, Lambda e V, com os mesmos significados que no método anterior.
  Também não retorna nada, só imprime na tela.

O código que implementa as estimativas de erro para o caso geral está na figura 2.1.2.2.1. Este método imprime na tela 3 estimativas de erro.

A primeira estimativa verifica o quão próxima de ser ortogonal está a matriz com os autovetores obtida pelo método QR.

A segunda estimativa se baseia na equação da figura 2.1.1.4. Se fizermos A*Q-Q*L, isso deveria dar um resultado nulo, então esta estimativa de erro verifica o quão próxima de zero está esta subtração.

A terceira estimativa de erro faz o cálculo dos autovalores fazendo A*v*inv(v), e compara isso com os autovalores obtidos. O resultado é o máximo dos erros.

```
def EstimativasErroGerais(self, n, alfa, beta, Lambda, V):
    # Faz uma estimativa de erro = max(abs(matmul(Q,QT)-I))
    erro = np.max(np.abs(np.matmul(V, np.transpose(V)) - np.identity(n)))
    print('max(abs(matmul(Q,QT)-I)) = {0}'.format(erro))

    # Faz uma estimativa de erro = max(abs(matmul(A,Q)-matmul(Q,L)))
    A = np.zeros((n, n))

    for i in range(0, n):
        A[i, i] = alfa[i]
    for i in range(0, n - 1):
        A[i, i + 1] = beta[i]
        A[i + 1, i] = beta[i]

    L = np.zeros((n, n))

    for i in range(0, n):
        L[i, i] = Lambda[i]

    erro = np.max(np.abs(np.matmul(A, V) - np.matmul(V, L)))
    print('max(abs(matmul(A,Q)-matmul(Q,L))) = {0}\n'.format(erro))

    # Faz uma estimativa de erro comparando (A*v)/v, com os autovalores obtidos
    erros = np.zeros(n)
    for i in range(0, n):
        erros[i] = math.sqrt(math.pow(np.mean(np.divide(np.matmul(A, V[:, i]), V[:, i])) - Lambda[i], 2))
        print('Autovalor obtido fazendo matdiv(matmul(A,v),v): {0:12.10f},'
              '  Autovalor obtido pelo método QR: {1:12.10f}, erro: {2}'
              .format(np.mean(np.divide(np.matmul(A, V[:, i]), V[:, i])), Lambda[i], erros[i]))
    print('Erro máximo = {0}\n'.format(np.max(erros)))
```

Figura 2.1.2.2.1

O código que implementa as estimativas de erro para o caso analítico está na figura 2.1.2.2.2. Este método calcula estimativas de erro para o caso com solução analítica, aquele em que todos os valores de alfa são 2, e todos os de beta, -1.

Primeiro o código calcula os autovalores reais usando a fórmula da figura 2.1.2.2.3, os armazena em autovalores_reais, e ordena de forma decrescente. Este último passo é feito pois o método QR é um método de potência, ou seja, os autovalores são encontrados em ordem decrescente.

Depois a comparação entre autovalores reais e autovalores obtidos pelo método é feita fazendo o módulo da diferença. O resultado final desta comparação é o máximo destes erros.

Para a comparação entre autovetores, primeiro os autovetores são calculados pela fórmula analítica, pela fórmula da figura 2.1.2.2.4, e são normalizados. Montando estes autovetores em uma matriz parecida com a matriz de autovetores gerada pelo método QR, eles podem ser comparados. A estimativa de erro final é o máximo do módulo da diferença entre as duas matrizes elemento a elemento.

```
def EstimativasErroAnalitico(self, n, Lambda, V):
    # Calcula os autovalores reais, pela fórmula analitica, e ordena eles do maior para o menor
    autovalores_reais = 2 * (1 - np.cos(np.arange(n, 0, -1) * math.pi / (n + 1)))
    autovalores_reais = np.flip(np.sort(autovalores_reais))

    autovalores_obtidos = np.copy(np.flip(np.sort(Lambda)))

    erros = np.zeros(n)

    # Faz a comparação entre os autovalores reais e os obtidos pelo método QR
    print('Autovalor obtido | Autovalor real | erro')
    for i in range(0, n):
        erros[i] = math.sqrt(math.pow(autovalores_obtidos[i] - autovalores_reais[i], 2))
        print('{0:12.10f}       {1:12.10f}     {2}'.format(autovalores_obtidos[i], autovalores_reais[i],erros[i]))
    print('-----------------------------')
    print('Erro máx: ', np.max(erros))
    print('\n')

    # Calcula os autovetores reais, pela fórmula analítica, montando uma matriz semelhante à V
    autovetores_reais = np.zeros((n, n))
    for j in range(0, n):
        autovetores_reais[:, j] = np.sin(np.arange(1, n + 1, 1) * (n - j) * math.pi / (n + 1))
        autovetores_reais[:, j] = self.__normaliza(autovetores_reais[:, j])

    # Faz a comparação entre os autovetores reais e os obtidos pelo método QR
    print('Erro obtido fazendo max(abs(Autovetores_reais-Autovetores_obtidos)) = {0}\n\n'.format(np.max(abs(autovetores_reais - V))))
```

Figura 2.1.2.2.2

$$\lambda_j = 2(1 - cos(\frac{j\pi}{n+1}))$$

Figura 2.1.2.2.3

$$v_j = (sin(\frac{j\pi}{n+1})), sin(\frac{2j\pi}{n+1})), \ldots, sin(\frac{nj\pi}{n+1}))$$

Figura 2.1.2.2.4

2.2 Segunda Tarefa: Resposta temporal de sistema massa-mola com n massas e n+1 molas

2.2.1 Sistema massa-mola

Um sistema massa-mola com uma massa e uma mola possui equação diferencial como da figura 2.2.1.1, cuja solução está na figura 2.2.1.2. Como no problema proposto pelo documento do exercício-programa a velocidade inicial é nula, então a solução não tem seno.

$$\ddot{x}(t) + \lambda x(t) = 0$$

Figura 2.2.1.1

$$x(t) = x(0)\cos(\sqrt{\lambda}t) + \dot{x}(0)\sin(\sqrt{\lambda}t)$$

Figura 2.2.1.2

2.2.2 Sistema massa-mola com n massas e n+1 molas

Suponha que este problema seja estendido para *n* massas, ligadas por *n+1* molas, entre dois anteparos, então a equação diferencial, vetorial, está na figura 2.2.2.1.

$$\ddot{X}(t) + AX(t) = 0$$

Figura 2.2.2.1

$$\ddot{X}(t) + Q\Lambda Q^T X(t) = 0$$

Figura 2.2.2.2

Como uma massa só está ligada a no máximo duas massas adjacentes, então a matriz A é simétrica tridiagonal, portanto, diagonalizável pelo método QR. Desta forma obtemos a equação da figura 2.2.2.2, que pode ser reescrita na forma da figura 2.2.2.3, em que $Y(t) = Q^T X(t)$

$$\ddot{Y}(t) + \Lambda Y(t) = 0$$

Figura 2.2.2.3

$$y_i(t) = y_i(0)\cos(\sqrt{\lambda_i}t)$$

Figura 2.2.2.4

Como Λ é diagonal, as soluções y(t) são da forma da figura 2.2.2.4 (velocidade inicial nula). Para obter as condições iniciais, pode-se fazer $Y(0) = QX(0)$. Convertendo de volta para x(t), o que está na figura 2.2.2.6 é obtido, e foi isso que foi implementado no programa documentado aqui.

$$x_i(t) = \sum_{j=1}^{n} Q_{i,j} y_j(t)$$

Figura 2.2.2.6

2.3 Interface de usuário

Na especificação do exercício-programa, dois itens foram pedidos, cálculo de autovalores e autovetores de matrizes tridiagonais simétricas, e resposta temporal para sistema de *n* massas e *n+1* molas. Estes dois modos foram implementados usando a classe singleton Interface.

Ela possui um método público, e os outros são todos privados. Possui também vários atributos, dentre os quais, os mais importantes são uma instância de *CalculaAutovalsAutovecs* e outra de *EstimadorDeErros*. Diagramas de sequência de telas estão nas figura 2.3.1 e 2.3.2

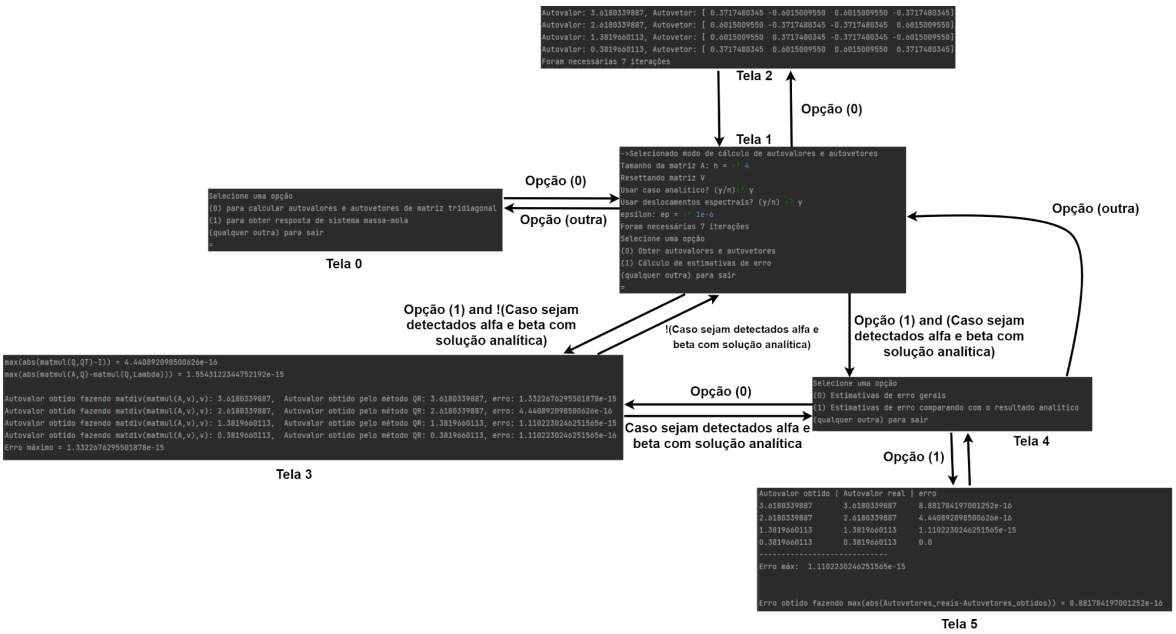


Figura 2.3.1 - Diagrama de sequência de telas para modo de cálculo de autovalores e autovetores de matriz tridiagonal simétrica

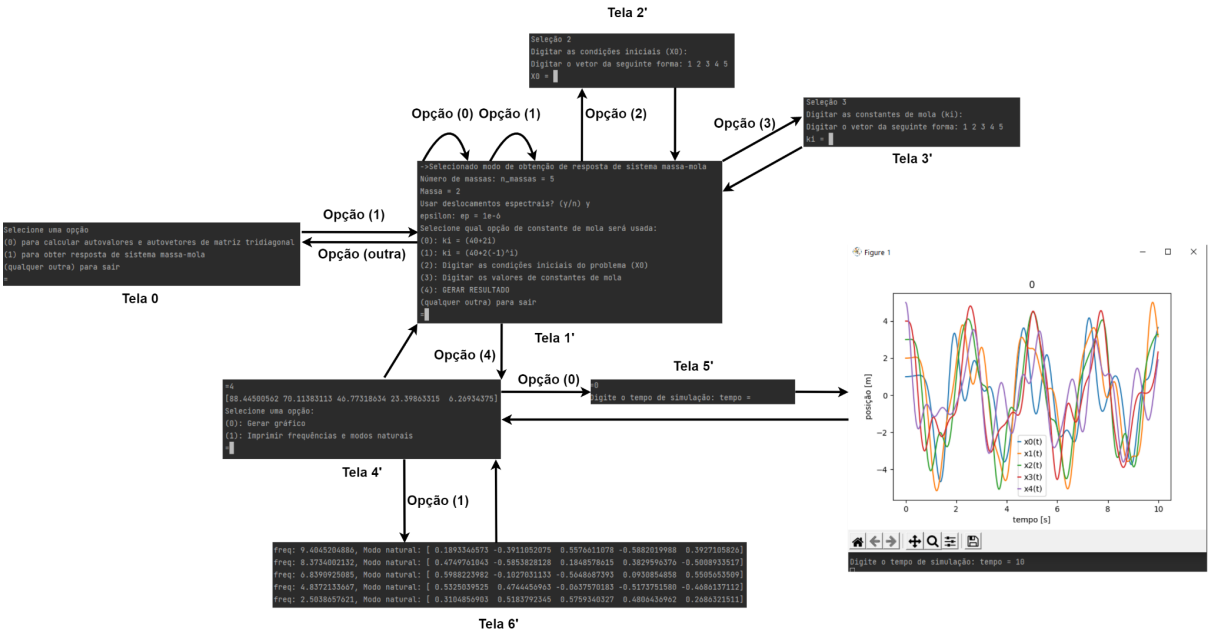


Figura 2.3.2 - Diagrama de sequência de telas para modo de resposta temporal de sistema massa-mola

2.3.1 Implementação da Interface de usuário

A classe singleton que implementa a interface de usuário tem os atributos no código da figura 2.3.1.1, com os significados na listagem abaixo:

- epsilon: entrada de usuário de tolerância para definição da condição de convergência;

- n: entrada de usuário de tamanho de matriz tridiagonal cujos autovalores e autovetores são calculados;
- alfa, beta: vetores de tamanho n com os elementos da matriz tridiagonal
- V: matriz onde os autovetores são armazenados nas colunas;
- Lambda: vetor onde os autovalores são armazenados;
- k: inteiro com o número de iterações necessárias para convergência
- calculadora: instância da classe *CalculaAutovalsAutovecs()*, que utiliza o método QR;
- estimadorDeErros: instância da classe *EstimadorDeErros()*, que imprime na tela as estimativas de erros;
- resultados: tupla onde os resultados de *CalculaAutovalsAutovecs()* são armazenados;
- n_massas: entrada de usuário para o número de massas no sistema massa-mola;
- massa: entrada de usuário de massa das massas do sistema massa-mola;
- X0: entrada de usuário para as condições iniciais do sistema massa-mola;
- Y0: X0 convertido para condições iniciais das equações yi(t);
- ki: entrada de usuário com as constantes de mola das molas do sistema massa-mola;
- usar_caso_analítico: entrada de usuário que define para sistema se o caso de matriz tridiagonal com solução analítica deve ser usado;
- usar_deslocamentos_espectrais: entrada de usuário que define se a calculadora deve usar deslocamentos espectrais.

```
alfa = np.zeros(1)
beta = np.zeros(1)

n = 0

usar_caso_analitico = False
usar_deslocamentos_espectrais = False
epsilon = 0.000001
V = np.identity(alfa.shape[0])
Lambda = 0
k = 0

calculadora = CalculaAutovalsAutovecs()
estimadorDeErros = EstimadorDeErros()

resultados = ()

n_massas = 0
massa = 0.0
X0 = np.zeros(n_massas)
Y0 = np.zeros(n_massas)
ki = np.zeros(n_massas+1)
```

Figura 2.3.1.1 - Atributos da classe *CalculaAutovalsAutovecs()*

No escopo principal do programa uma instância de *Interface()* é criada, e seu único método público é chamado, *promptTelaInicial()*, cujo código está na figura 2.3.1.2. Este método implementa a tela 0, das figuras de diagramas de sequência. É daqui que os dois modos do programa podem ser selecionados: o modo de cálculo de autovalores e autovetores, selecionando 0, que chama o método privado *__promptCalcularAutovalsAutovecs()*; e o modo de resposta temporal de sistema massa-mola, selecionando 1, que chama o método privado *__promptMassaMola()*.

```python
# Esta é a primeira tela vista pelo usuário
def promptTelaInicial(self):
    # Menu de seleção inicial
    print('Selecione uma opção')
    print('(0) para calcular autovalores e autovetores de matriz tridiagonal')
    print('(1) para obter resposta de sistema massa-mola')
    print('(qualquer outra) para sair')
    selecao = input('=')

    if(selecao == '0'):
        self.__promptCalcularAutovalsAutovecs()
        print('->Saindo...')
        self.promptTelaInicial()
    elif(selecao == '1'):
        self.__promptMassaMola()
        print('->Saindo...')
        self.promptTelaInicial()
    else:
        print('->Saindo...')
        return
```

Figura 2.3.1.2

2.3.1.1 Modo de cálculo de autovalores e autovetores de matriz tridiagonal

Este modo é iniciado quando *__promptCalcularAutovalsAutovecs()* é chamado, seu código está na figura 2.3.1.1.1.

Dentro de um bloco try-catch, faz a entrada de usuário dos atributos n, alfa, beta, usar_deslocamentos_espectrais, usar_caso_analítico e epsilon. Além disto, também 'Resetta' a matriz V para a matriz identidade, como demanda o algoritmo, como mostra a figura 2.1.1.5. Se o usuário selecionar usar_caso_analítico, o programa calcula automaticamente os vetores alfa e beta, a partir da entrada n. Se algum erro de digitação for detectado, o sistema volta para a tela 0.

Após fazer a entrada de usuário, chama *calculadora.AutovalAutovec(alfa, beta, epsilon, deslocamentos, V)*, coloca os resultados na tupla *resultados*, e extrai da mesma *Lambda*, *V* e *k*.

Por fim, chama o método privado *__promptTelaResultadosCalculoAutovalsAutovecs()*, cujo código pode ser encontrado na figura 2.3.1.1.2.

```
def __promptCalcularAutovalsAutovecs(self):
    print('->Selecionado modo de cálculo de autovalores e autovetores') #Confirmação de seleção

    try:
        self.n = int(input('Tamanho da matriz A: n = '))
        print('Resettando matriz V')
        self.V = np.identity(self.n)
        if(input('Usar caso analítico? (y/n)') == 'y'):
            self.usar_caso_analitico = True
            self.alfa = 2*np.ones(self.n)
            self.beta = -1*np.ones(self.n-1)
        else:
            print('Digitar os vetores da seguinte forma (n=3): 1 2 3')
            self.alfa = np.array(list(map(float, input("alfa = ").strip().split()))[:self.n])
            self.beta = np.array(list(map(float, input("beta = ").strip().split()))[:self.n - 1])

        # Solicita ao usuário se deseja usar deslocamentos espectrais
        if(input('Usar deslocamentos espectrais? (y/n) ') == 'y'):
            self.usar_deslocamentos_espectrais = True
        else:
            self.usar_deslocamentos_espectrais = False

        self.epsilon = float(input('epsilon: ep = '))

        if(self.alfa.shape[0] != self.n and self.beta.shape[0] != self.n-1):
            raise Exception()
    except:
        print('Erro de digitação')
        return

    # Aplica o método de obtenção de autovalores e autovetores usando decomposição QR
    self.resultados = self.calculadora.AutovalAutovec(self.alfa, self.beta, self.epsilon,
                                                      self.usar_deslocamentos_espectrais, self.V)
    self.Lambda = self.resultados[0]
    self.V = self.resultados[1]
    self.k = self.resultados[2]
    self.__promptTelaResultadosCalculoAutovalsAutovecs()
```

Figura 2.3.1.1.1

```
def __promptTelaResultadosCalculoAutovalsAutovecs(self):

    # Número de iterações necessárias
    print('Foram necessárias {0:d} iterações'.format(self.k))

    # Menu de seleção
    print('Selecione uma opção')
    print('(0) Obter autovalores e autovetores')
    print('(1) Cálculo de estimativas de erro')
    print('(qualquer outra) para sair')
    selecao = input('=')

    if(selecao == '0'):
        self.__mostrarAutovalsAutovecs()
        self.__promptTelaResultadosCalculoAutovalsAutovecs()
    elif(selecao == '1'):
        self.__promptEstimativasDeErro()
        self.__promptTelaResultadosCalculoAutovalsAutovecs()
    else:
        return
```

Figura 2.3.1.1.2

```
def __mostrarAutovalsAutovecs(self):
    for i in range(0, self.alfa.shape[0]):
        np.set_printoptions(formatter={'float': '{: 12.10f}'.format})
        print('Autovalor: {0:12.10f}, Autovetor:'.format(self.Lambda[i]), self.V[:, i])
```

Figura 2.3.1.1.3

```
def __promptEstimativasDeErro(self):
    # Porém se for detectada matriz com solução analítica, outro menu surge
    if (np.array_equal(self.alfa, 2 * np.ones(self.n)) and np.array_equal(self.beta, -1 * np.ones(self.n - 1))):
        print('Selecione uma opção')
        print('(0) Estimativas de erro gerais')
        print('(1) Estimativas de erro comparando com o resultado analítico')
        print('(qualquer outra) para sair')
        selecao = input('=')

        if(selecao == '0'):
            self.estimadorDeErros.estimarErro(self.n, self.alfa, self.beta, self.Lambda, self.V, False)
            self.__promptEstimativasDeErro()
        elif(selecao == '1'):
            self.estimadorDeErros.estimarErro(self.n, self.alfa, self.beta, self.Lambda, self.V, True)
            self.__promptEstimativasDeErro()
        else:
            return
    else:
        self.estimadorDeErros.estimarErro(self.n, self.alfa, self.beta, self.Lambda, self.V, False)
        return
```

Figura 2.3.1.1.4

Este método imprime a tela 1. Se a opção 0 for selecionada, o método privado *__mostrarAutovalsAutovecs()*, cujo código está na figura 2.3.1.1.3, que apenas faz a impressão na tela dos autovalores encontrados, e os respectivos autovetores, e, depois, volta para a mesma tela. Caso a opção 1 seja selecionada, o método privado *__promptEstimativasDeErro()* é chamado, cujo código está na figura 2.3.1.1.4.

Para as estimativas de erro, se matriz tridiagonal com solução analítica for detectada, aparecem duas opções: imprimir estimativas de erro gerais, que servem para qualquer matriz tridiagonal simétrica; e estimativas de erro para caso analítico, que faz comparações com a solução analítica.

2.3.1.2 Modo de cálculo de resposta temporal de sistema massa-mola

Este modo é iniciado quando método privado *__promptMassaMola()* é chamado. Seu código está na figura 2.3.1.2.1.

Assim como *__promptCalcularAutovalsAutovecs()*, *__promptMassaMola()* também faz a entrada de usuário, mas só dos atributos n_massas, massa, usar_deslocamentos_espectrais e epsilon, os outros são 'resettados' para serem entrados em outra tela.

```
def __promptMassaMola(self):
    print('->Selecionado modo de obtenção de resposta de sistema massa-mola')

    try:
        self.n_massas = int(input('Número de massas: n_massas = '))

        print('Resettando matriz V')
        self.V = np.identity(self.n_massas)

        self.X0 = np.zeros(self.n_massas)
        self.Y0 = np.zeros(self.n_massas)
        self.ki = np.zeros(self.n_massas+1)

        self.massa = float(input('Massa = '))

        # Solicita ao usuário se deseja usar deslocamentos espectrais
        if (input('Usar deslocamentos espectrais? (y/n) ') == 'y'):
            self.usar_deslocamentos_espectrais = True
        else:
            self.usar_deslocamentos_espectrais = False

        self.epsilon = float(input('epsilon: ep = '))
    except:
        print('Digitação errada')
        return
    self.__promptMenuMassaMola()
```

Figura 2.3.1.2.1

```
def __promptMenuMassaMola(self):
    print('Selecione qual opção de constante de mola será usada:')
    print('(0): ki = (40+2i)')
    print('(1): ki = (40+2(-1)^i)')
    print('(2): Digitar as condições iniciais do problema (X0)')
    print('(3): Digitar os valores de constantes de mola')
    print('(4): GERAR RESULTADO')
    print('(qualquer outra) para sair')
    selecao = (input('='))

    if(selecao == '0'):
        print('Seleção 0')
        self.ki = 40 + 2 * np.arange(1, self.n_massas + 2)
        self.__promptMenuMassaMola()
    elif(selecao == '1'):
        print('Seleção 1')
        self.ki = 40 + 2 * (-1) ** np.arange(1, self.n_massas + 2)
        self.__promptMenuMassaMola()
    elif(selecao == '2'):
        print('Seleção 2')
        print('Digitar as condições iniciais (X0):')
        print('Digitar o vetor da seguinte forma: 1 2 3 4 5')

        try:
            self.X0 = np.array(list(map(float, input("X0 = ").strip().split())))[:self.n_massas])
            if (self.X0.shape[0] != self.n_massas):
                raise Exception()
        except:
            print('Digitação errada')
        self.__promptMenuMassaMola()
```

```
    elif(selecao == '3'):
        print('Seleção 3')
        print('Digitar as constantes de mola (ki):')
        print('Digitar o vetor da seguinte forma: 1 2 3 4 5')

        try:
            self.ki = np.array(list(map(float, input("Ki = ").strip().split())))[:self.n_massas+1])
            print('ki=',self.ki)
            if (self.ki.shape[0] != self.n_massas+1):
                raise Exception()
        except:
            print('Digitação errada')
        self.__promptMenuMassaMola()
    elif(selecao == '4'):
        self.alfa = np.ones(self.n_massas)
        self.beta = np.ones(self.n_massas - 1)
        self.V = np.identity(self.n_massas)

        for i in range(0, self.n_massas):
            self.alfa[i] = (self.ki[i] + self.ki[i + 1]) / self.massa
        for i in range(0, self.n_massas - 1):
            self.beta[i] = -self.ki[i + 1] / self.massa

        self.resultados = self.calculadora.AutovalAutovec(self.alfa, self.beta, self.epsilon,
                                                          self.usar_deslocamentos_espectrais, self.V)
        self.Lambda = self.resultados[0]
        self.V = self.resultados[1]

        print(self.resultados[0])

        self.__promptTelaResultadosMassaMola()
        self.__promptMenuMassaMola()
    else:
        return
```

Figura 2.3.1.2.2

Depois disto, a tela 1' é impressa, através do método *__promptMenuMassaMola()*, cujo código se encontra na figura 2.3.1.2.2.

Se trata de um menu: se a opção 0 for selecionada, as constantes de mola são calculadas a partir da fórmula $k_i = (40 + 2i)$; se a opção 1 for selecionada, as constantes de mola são calculadas a partir da fórmula $k_i = (40 + 2(-1)^i)$; se a opção 2 for selecionada, o usuário deve entrar as condições iniciais do sistema massa-mola; se a opção 3 for selecionada, o usuário pode entrar valores customizados de constante de mola; por fim, se a opção 4 for selecionada, o sistema gera um relatório de resultados.

Dando um foco maior na opção 4, ela faz algo a mais: monta a matriz A do sistema massa-mola usando os fors destacados na figura 2.3.1.2.3; e calcula os autovalores e autovetores da mesma, como mostra o código destacado na figura 2.3.1.2.4.

```
for i in range(0, self.n_massas):
    self.alfa[i] = (self.ki[i] + self.ki[i + 1]) / self.massa
for i in range(0, self.n_massas - 1):
    self.beta[i] = -self.ki[i + 1] / self.massa
```

Figura 2.3.1.2.3

```
self.resultados = self.calculadora.AutovalAutovec(self.alfa, self.beta, self.epsilon,
                                                   self.usar_deslocamentos_espectrais, self.V)
self.Lambda = self.resultados[0]
self.V = self.resultados[1]

print(self.resultados[0])

self.__promptTelaResultadosMassaMola()
self.__promptMenuMassaMola()
```

Figura 2.3.1.2.4

```
def __promptTelaResultadosMassaMola(self):
    print('Selecione uma opção:')
    print('(0): Gerar gráfico')
    print('(1): Imprimir frequências e modos naturais')
    selecao = (input('='))

    if(selecao == '0'):
        self.__gerarGrafico()
        self.__promptTelaResultadosMassaMola()
    elif(selecao == '1'):
        for i in range(0, self.n_massas):
            np.set_printoptions(formatter={'float': '{: 12.10f}'.format})
            print('freq: {0:12.10f}, Modo natural:'.format(math.sqrt(self.Lambda[i])), self.V[:,i])
        self.__promptTelaResultadosMassaMola()
    else:
        print('->Saindo...')
        return
```

Figura 2.3.1.2.5

Depois disto o método *__promptTelaResultadosMassaMola()* é chamado, cujo código pode ser encontrado na figura 2.3.1.2.5. Se a opção 1 for selecionada, uma impressão parecida com a tela 2 é feita, mas neste caso, ao invés de imprimir os autovalores, as suas raízes quadradas são impressas, que são as frequências naturais do sistema massa-mola.

```
def __gerarGrafico(self):
    # Este for gera Y(0) a partir de X(0)
    # Faz isto usando a relação Y(0)=Q*X(0)
    for i in range(0, self.n_massas):
        soma = 0
        for j in range(0, self.n_massas):
            soma = soma + np.transpose(self.V)[i, j] * self.X0[j]
        self.Y0[i] = soma

    # Esta função é usada por plt.plot para gerar o gráfico
    # Aqui a função xi(t) = sum_{j=1}^n Q_{ij}y_j(t) é implementada
    def xi(i, t, n_massas, V, Y0, Lambda):
        soma = 0
        for j in range(0, n_massas):
            soma = soma + Y0[j] * V[i, j] * np.cos(np.sqrt(Lambda[j]) * t)
        return soma

    # Solicitar ao usuário o tempo de simulação desejado
    tempo_de_simulacao=0
    try:
        tempo_de_simulacao = float(input('Digite o tempo de simulação: tempo = '))
    except:
        print('Digitação errada')
        self.__gerarGrafico()

    # Gerar gráfico
    fig, axs = plt.subplots(int(math.ceil(self.n_massas/2)), 2)
    t = np.arange(0, tempo_de_simulacao, tempo_de_simulacao/10000)
    for i in range(0, self.n_massas):
        ax = axs[int(math.floor(i/2)),i % 2]
        ax.plot(t, xi(i, t, self.n_massas, self.V, self.Y0, self.Lambda), label='x{0:d}(t)'.format(i))
        ax.yaxis.set_major_locator(ticker.MaxNLocator(integer=True, symmetric=False, min_n_ticks=5))
        ax.grid()
        ax.legend(handlelength=-0.4)
        ax.set_xlabel('tempo [s]')
        ax.set_ylabel('posição [m]')

    plt.ioff()
```

Figura 2.3.1.2.6

Caso a opção de gerar gráfico seja selecionada, o método da figura 2.3.1.2.6 é chamado.

A primeira operação deste método é calcular as condições iniciais em termos de Y(t), isso é feito fazendo $Y(0) = QX(0)$, implementado com o primeiro for.

A função xi é usada para plotar o gráfico, ela faz a implementação da função da figura 2.2.2.6.

O método também faz a entrada de usuário de tempo de simulação, em segundos.

Por fim, as respostas temporais das n_massas são plotadas usando matplotlib.

3. Resultados

3.1. Comparação entre autovalores e autovetores obtidos pelo método QR com e sem deslocamentos espectrais com a solução analítica

Com uma tolerância de epsilon=1e-6, os resultados obtidos estão na tabela abaixo.

É importante notar que 'erro geral máximo' é a maior das estimativas de erros geradas pela impressão, idem para 'erro analítico máximo'.

	sem deslocamentos espectrais			com deslocamentos espectrais		
n	k	erro geral máximo	erro analítico máximo	k	erro geral máximo	erro analítico máximo
4	45	6,5060818E-07	6,4544993E-07	7	1,5543122E-15	1,1102230E-15
8	143	1,3102974E-02	1,9872943E-06	15	6,7170217E-01	9,0264018E-11
16	473	6,1971393E-07	5,2911683E-06	31	9,2528689E-09	6,8306805E-01
32	1600	1,5260682E-03	1,4480001E-05	59	9,0818706E-02	4,3317809E-01

Como é possível notar pela tabela, o número de iterações usando deslocamentos espectrais cai significativamente, em ordens de grandeza.

Analisando só o caso com deslocamentos espectrais, existem algumas configurações nas quais um dos erros fica muito maior que o esperado.

Por exemplo, para n=8, o erro geral fica muito grande, como mostra a tabela e a figura 3.1.1, mas os autovalores e autovetores estão próximos do valor real por causa do erro analítico pequeno.

Já para n=16 em diante, o erro analítico começa a ficar grande, pois os autovetores obtidos começam a diferir dos autovetores reais, como mostra a figura 3.1.2.

```
max(abs(matmul(Q,QT)-I)) = 6.661338147750939e-16
max(abs(matmul(A,Q)-matmul(Q,L))) = 3.267598691625295e-11

Autovalor obtido fazendo matdiv(matmul(A,v),v): 3.8793852416,  Autovalor obtido pelo método QR: 3.8793852416, erro: 4.440892098500626e-16
Autovalor obtido fazendo matdiv(matmul(A,v),v): 3.5320888862,  Autovalor obtido pelo método QR: 3.5320888862, erro: 3.4638958368304884e-14
Autovalor obtido fazendo matdiv(matmul(A,v),v): 2.3282978316,  Autovalor obtido pelo método QR: 3.0000000000, erro: 0.6717021683735531
Autovalor obtido fazendo matdiv(matmul(A,v),v): 2.3472963553,  Autovalor obtido pelo método QR: 2.3472963553, erro: 9.459100169806334e-14
Autovalor obtido fazendo matdiv(matmul(A,v),v): 1.6527036447,  Autovalor obtido pelo método QR: 1.6527036447, erro: 6.661338147750939e-16
Autovalor obtido fazendo matdiv(matmul(A,v),v): 0.7800016824,  Autovalor obtido pelo método QR: 1.0000000000, erro: 0.2199983176089854
Autovalor obtido fazendo matdiv(matmul(A,v),v): 0.4679111138,  Autovalor obtido pelo método QR: 0.4679111138, erro: 4.440892098500626e-16
Autovalor obtido fazendo matdiv(matmul(A,v),v): 0.1206147584,  Autovalor obtido pelo método QR: 0.1206147584, erro: 4.235944928154822e-12
Erro máximo = 0.6717021683735531
```

Figura 3.1.1 - Erro geral para n=8

```
Autovalor obtido | Autovalor real | erro
3.9659461994       3.9659461994     4.440892098500626e-15
3.8649444588       3.8649444588     3.1086244689504383e-15
3.7004342715       3.7004342715     2.6645352591003757e-15
3.4780178344       3.4780178344     3.1086244689504383e-15
3.2052692728       3.2052692728     1.3322676295501878e-15
2.8914767116       2.8914767116     4.440892098500626e-16
2.5473259801       2.5473259801     4.440892098500626e-16
2.1845367189       2.1845367189     0.0
1.8154632811       1.8154632811     8.881784197001252e-16
1.4526740199       1.4526740199     0.0
1.1085232884       1.1085232884     2.220446049250313e-16
0.7947307272       0.7947307272     2.220446049250313e-16
0.5219821656       0.5219821656     0.0
0.2995657285       0.2995657285     3.552713678800501e-15
0.1350555412       0.1350555412     3.774758283725532e-15
0.0340538006       0.0340538006     2.220446049250313e-16
----------------------------
Erro máx:  4.440892098500626e-15


Erro obtido fazendo max(abs(Autovetores_reais-Autovetores_obtidos)) = 0.683068050802037
```

Figura 3.1.2 - Erro analítico para n=16

3.2. Respostas temporais para sistema massa-mola

Para um sistema com 5 massas de 2 kg, e molas seguindo $k_i = (40+2i)\ \mathrm{N/m}$. As frequências e modos naturais estão na figura 3.2.1. E as respostas temporais estão nas figuras 3.2.2 à 3.2.4.

```
freq: 9.4045204886, Modo natural: [ 0.1893346573 -0.3911052075  0.5576611078 -0.5882019988  0.3927105826]
freq: 8.3734002132, Modo natural: [ 0.4749761043 -0.5853828128  0.1848578615  0.3829596376 -0.5008933517]
freq: 6.8390925085, Modo natural: [ 0.5988223982 -0.1027031133 -0.5648687393  0.0930854858  0.5505653509]
freq: 4.8372133667, Modo natural: [ 0.5325039525  0.4744456963 -0.0637570183 -0.5173751580 -0.4686137112]
freq: 2.5038657621, Modo natural: [ 0.3104856903  0.5183792345  0.5759340327  0.4806436962  0.2686321511]
```

Figura 3.2.1 - Frequências e modos naturais para sistema massa-mola com n_massas=5, m=2 Kg e $k_i = (40+2i)\ \mathrm{N/m}$

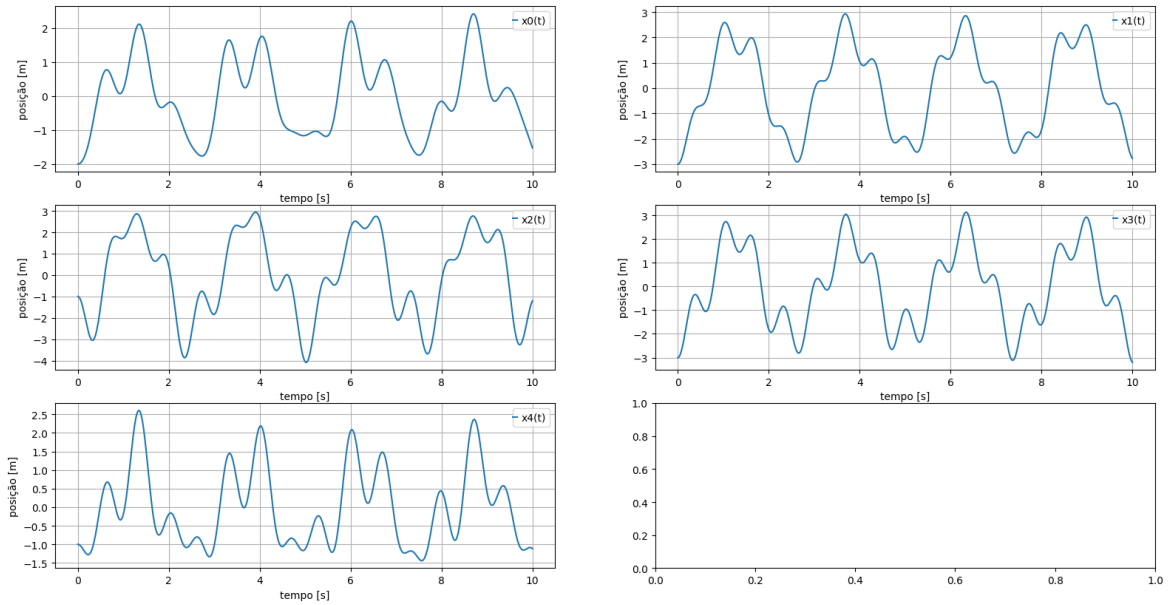


Figura 3.2.2 - Resposta temporal para sistema massa-mola com n_massas=5, m=2 Kg, $k_i = (40+2i)\ \mathrm{N/m}$ e X(0) = (-2,-3,-1,-3,-1)

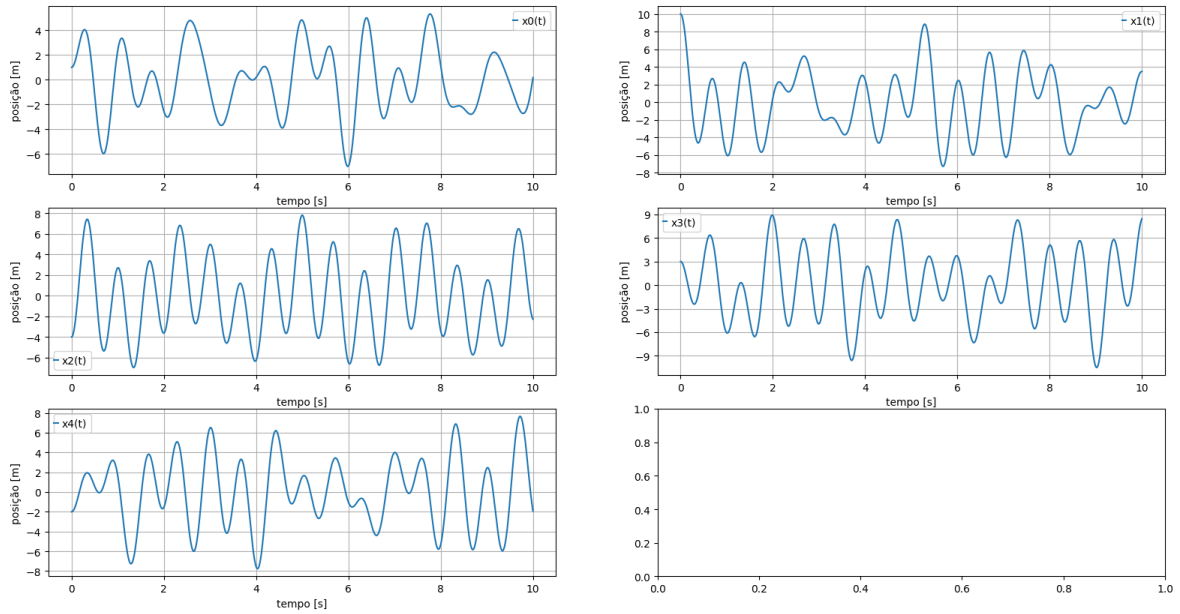


Figura 3.2.3 - Resposta temporal para sistema massa-mola com n_massas=5, m=2 Kg, $k_i = (40+2i)\ \mathrm{N/m}$ e X(0) = (1,10,-4,3,-2)

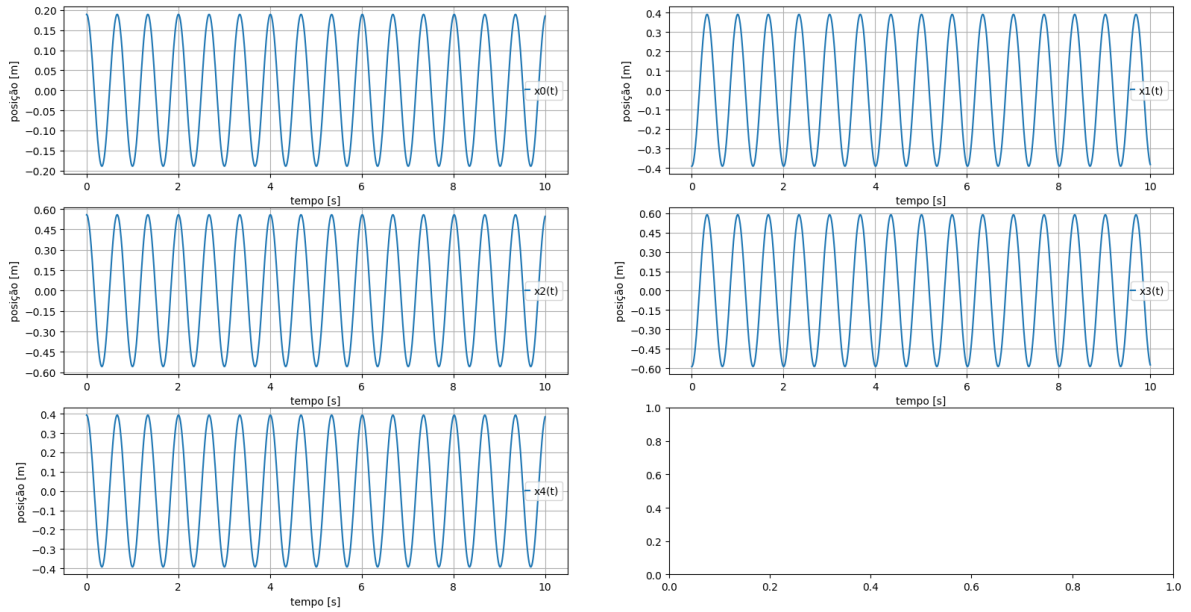


Figura 3.2.4 - Resposta temporal para sistema massa-mola com n_massas=5, m=2 Kg, $k_i = (40+2i)$ N/m e X(0) = (0.1893346573,-0.3911052075,0.5576611078,-0.5882019988, 0.3927105826)

Para um sistema com 10 massas de 2 Kg com molas $k_i = (40 + 2(-1)^i)$, as frequências e modos naturais obtidos estão na figura 3.2.5, e as respostas temporais, nas figuras 3.2.6 a 3.2.8.

```
freq: 8.8543067475, Modo natural: [ 0.1252452151 -0.2290123547  0.3244020093 -0.3859712504  0.4214931883 -0.4214931883  0.3859712504 -0.3244020093  0.2290123547 -0.1252452151]
freq: 8.5866132967, Modo natural: [ 0.2400737896 -0.3856034102  0.4192013498 -0.3244365006  0.1126311279  0.1126311279 -0.3244365006  0.4192013498 -0.3856034102  0.2400737896]
freq: 8.1482365262, Modo natural: [ 0.3345508219 -0.4204787426  0.2143393222  0.1110417051 -0.3911544059  0.3911544059 -0.1110417051 -0.2143393222  0.4204787426 -0.3345508219]
freq: 7.5540684929, Modo natural: [ 0.3982121962 -0.3235749201 -0.1495257687  0.4142582781 -0.2067811438 -0.2067811438  0.4142582781 -0.1495257687 -0.3235749201  0.3982121962]
freq: 6.8557546803, Modo natural: [ 0.3951974267 -0.1317582996 -0.3882451084  0.2486502959  0.3374870525 -0.3374870525 -0.2486502959  0.3882451084  0.1317582996 -0.3951974267]
freq: 5.7444432074, Modo natural: [ 0.3951974267  0.1317582996 -0.3882451084 -0.2486502959  0.3374870525  0.3374870525 -0.2486502959 -0.3882451084  0.1317582996  0.3951974267]
freq: 4.7891595509, Modo natural: [ 0.3982121962  0.3235749201 -0.1495257687 -0.4142582781 -0.2067811439  0.2067811438  0.4142582781  0.1495257686 -0.3235749201 -0.3982121962]
freq: 3.6886639198, Modo natural: [ 0.3345508220  0.4204787428  0.2143393224 -0.1110417049 -0.3911544056 -0.3911544056 -0.1110417049  0.2143393224  0.4204787428  0.3345508220]
freq: 2.5040112007, Modo natural: [ 0.2400737873  0.3856034061  0.4192013440  0.3244364937  0.1126311204 -0.1126311355 -0.3244365075 -0.4192013556 -0.3856034143 -0.2400737918]
freq: 1.2654058719, Modo natural: [ 0.1252452192  0.2290123614  0.3244020167  0.3859712562  0.4214931905  0.4214931865  0.3859712447  0.3244020017  0.2290123476  0.1252452106]
```

Figura 3.2.5 - Frequências e modos naturais para sistema massa-mola com n_massas=10, m=2 Kg e $k_i = (40 + 2(-1)^i)$

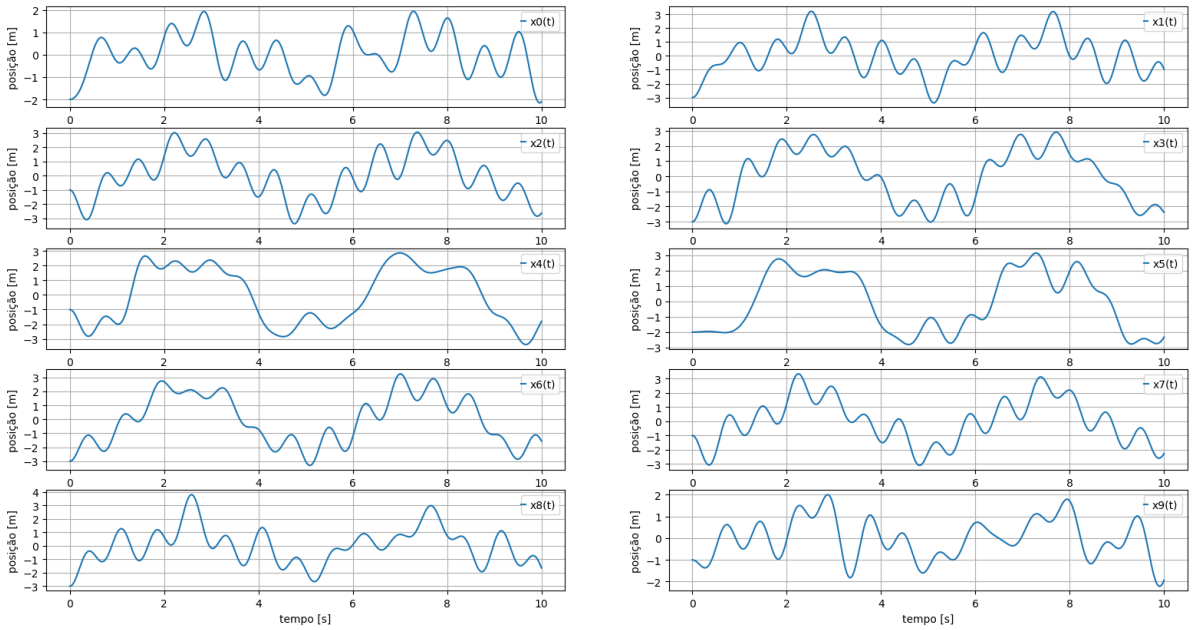


Figura 3.2.6 - Resposta temporal para sistema massa-mola com n_massas=10, m=2 Kg, $k_i = (40 + 2(-1)^i)$ e X(0) = (-2,-3,-1,-3,-1,-2,-3,-1,-3,-1)

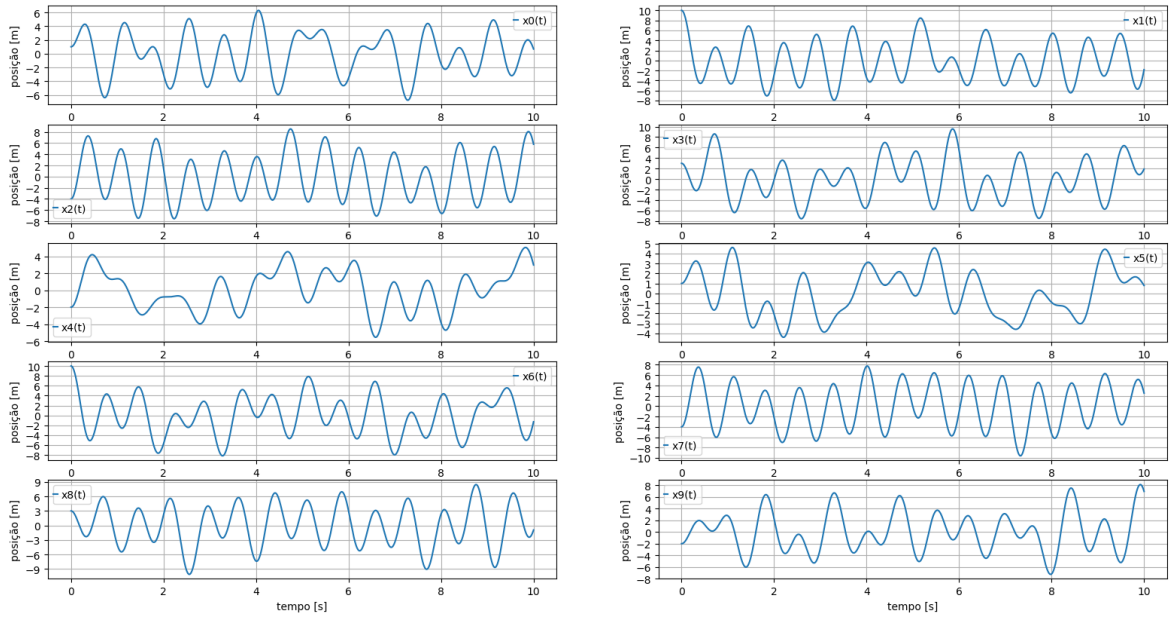


Figura 3.2.6 - Resposta temporal para sistema massa-mola com n_massas=10, m=2 Kg, $k_i = (40 + 2(-1)^i)$ e X(0) = (1,10,-4,3,-2,1,10,-4,3,-2)

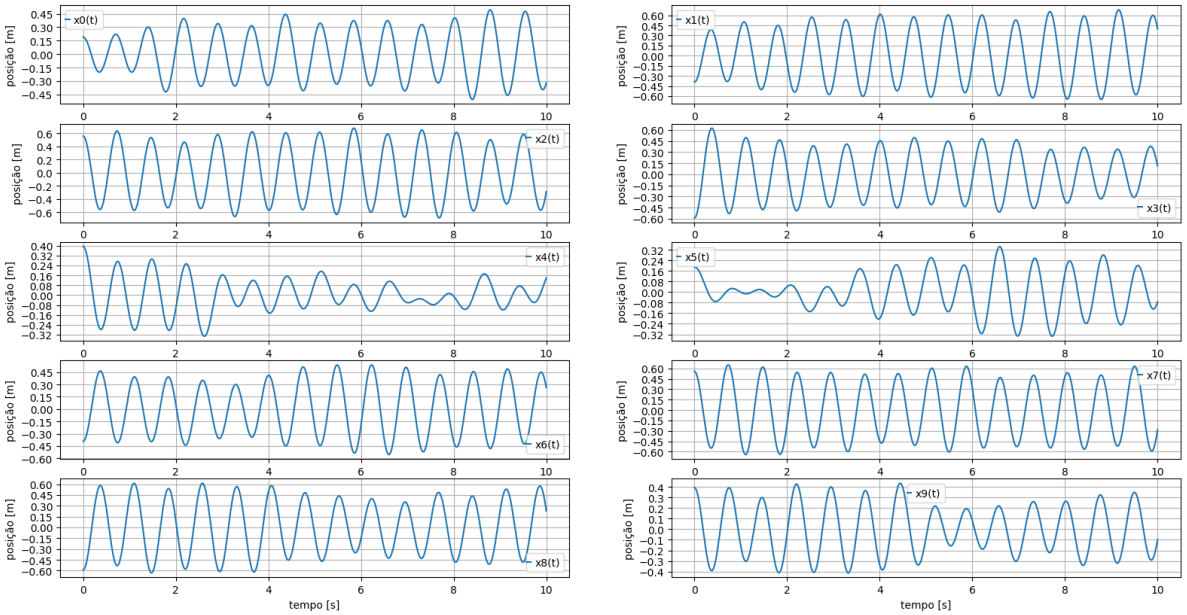


Figura 3.2.4 - Resposta temporal para sistema massa-mola com n_massas=10, m=2 Kg, $k_i = (40 + 2(-1)^i)$ e X(0) = (0.1893346573,-0.3911052075,0.5576611078,-0.5882019988, 0.3927105826,0.1893346573,-0.3911052075,0.5576611078,-0.5882019988, 0.3927105826)