1) Système masse-ressort-amortisseur (20 points)

Vous devez écrire un programme permettant de simuler un système de deuxième ordre : le système masse-ressort-amortisseur. La réponse temporelle d'un système de deuxième ordre peut être simulée facilement à l'aide d'une somme de deux exponentielles, d'où le nom système de deuxième ordre.

Le guide de développement se déroule de la façon suivante : tout d'abord vous définirez les constantes nécessaires à la bonne exécution du programme; ensuite, vous réaliserez une fonction permettant de calculer les deux constantes de temps du système; puis une autre permettant de calculer les deux gains du système; avant de terminer, vous écrirez une fonction permettant de calculer la valeur du système pour un point dans le temps; et vous conclurez le tout avec l'écriture de la routine principale, qui simule pour une série de points dans le temps.

a) constantes.m (2.5 points)

Définissez une série de constantes pour les paramètres suivants : la vitesse initiale, est égale à 0; la position initiale est égale à -10; la résolution de la simulation est de 0.1 seconde et le temps de simulation maximal est de 10 secondes. Les points vous sont accordés conditionnellement à une bonne déclaration et à une bonne utilisation des constantes au cours de ce numéro.

```
VITESSE_INITIALE = 0; % condition initiale de la dérivé (vitesse)
POSITION_INITIALE = -10; % condition initiale du système

RESOLUTION = 0.1;
TEMPS_MAX = 10;
```

b) calculer_constantes_temps.m (2.5 points)

Un système de deuxième ordre se définit généralement à l'aide de la pulsation propre, ω_0, et le taux d'amortissement, ς. Malheureusement, cette définition ne se prête pas bien à la simulation de la réponse temporelle, nous devons donc travailler un peu pour obtenir une formulation qui est plus facile à travailler.

Notre fonction reçoit donc la pulsation propre et le taux d'amortissement, mais doit s'en servir pour calculer les deux racines du système à l'aide de l'équation suivante, prenez note que la racine 1 est calculée en utilisant le « + » et la racine 2 avec le « - », en place du « ± »:

$$\text{Racines:} \qquad \omega_n = -\omega_0 \varsigma \pm \omega_0 \sqrt{\varsigma^2 - 1}$$

Les racines ne sont toutefois pas encore la meilleure façon de définir le système, mais permettent de calculer facilement les constantes de temps à l'aide de l'équation suivante :

$$\text{Constante de temps:} \qquad \tau_n = -\frac{1}{\omega_n}$$

et ce sont les constantes de temps τ_1 et τ_2 qui sont retournées par la fonction.

Réalisez la fonction permettant de calculer les constantes de temps à partir de la pulsation propre et du facteur d'amortissement.

```
%
% Fichier : calculer_constantes_temps.m
% Description : Cette fonction permet de calculer les constantes de temps
%               d'un système de deuxième ordre, à partir de la pulsation
%               propre et du facteur d'amortissement.
%
% Entrées :
%     - pulsation_propre, la pulsation propre du système
%     - facteur_amorti, le facteur d'amortissement du système

%
%
% Sortie :
%     - tau_1, constante de temps de la première exponentielle
%     - tau_2, constante de temps de la seconde exponentielle
%

function [tau_1, tau_2] = calculer_constantes_temps(pulsation_propre, facteur_amorti)


    sous_calcul = pulsation_propre*sqrt(facteur_amorti^2-1);

    racine_1 = -pulsation_propre*facteur_amorti + sous_calcul;

    racine_2 = -pulsation_propre*facteur_amorti - sous_calcul;

    tau_1 = -1/racine_1;
    tau_2 = -1/racine_2;




end
```

c) calculer_gain.m (2.5 points)

De façon similaire, le système a deux gains, mais chacun d'eux est calculé de la même façon, il suffit de changer les indices. Voici les équations à utiliser :

$$gain_1 = \frac{\tau_1 * \tau_2}{\tau_1 - \tau_2} * (VITESSE\_INITIALE + \frac{POSITION\_INITIALE}{\tau_2})$$

$$gain_2 = \frac{\tau_1 * \tau_2}{\tau_2 - \tau_1} * (VITESSE\_INITIALE + \frac{POSITION\_INITIALE}{\tau_1})$$

```
%
% Fichier : calculer_gains.m
% Description : Cette fonction permet de calculer les gains du système
%               de deuxième ordre.
%
% Entrées :
%     - tau_1, constante de temps de la première exponentielle
%     - tau_2, constante de temps de la seconde exponentielle
%
%
% Sortie :
%     - gain_1, gain de la première exponentielle
%     - gain_2, gain de la seconde exponentielle
%

function [gain_1, gain_2] = calculer_gains(tau_1, tau_2)


    constantes;

    gain_1 = tau_1*tau_2/(tau_1-tau_2) * (VITESSE_INITIALE + POSITION_INITIALE /tau_2);
    gain_2 = tau_1*tau_2/(tau_2-tau_1) * (VITESSE_INITIALE + POSITION_INITIALE /tau_1);







end
```

d) (2.5 points)

Vous devez maintenant écrire une fonction qui permet de calculer l'équation suivante, pour un temps, *t*, donné.

$$x(t) = gain_1 * e^{-\frac{t}{t_1}} + gain_2 * e^{-\frac{t}{t_2}}$$

```
%
% Fichier : simuler_systeme.m
% Description : Cette fonction permet de simuler le système à un moment
%               donné.
%
% Entrées :
%     - gain_1, gain de la première exponentielle
%     - gain_2, gain de la seconde exponentielle
%     - tau_1, constante de temps de la première exponentielle
%     - tau_2, constante de temps de la seconde exponentielle
%     - temps, temps auquel on veut déterminer l'état du système

%
%
% Sortie :
%     - etat, état du système au moment demandé
%

function [etat] = simuler_systeme(gain_1, gain_2, tau_1, tau_2, temps)

    etat = gain_1*exp(-temps/tau_1)+gain_2*exp(-temps/tau_2);



end
```

e) Programme principal (10 points)

Le programme_principal demande à l'usager d'entrer la pulsation propre et le facteur amorti du système. Il suit ensuite les étapes suivantes :

- calcule les constantes de temps.
- calcule les gains.
- Initialise le tableau **temps** contenant les temps à simuler allant de 0 au temps maximal de la simulation, en utilisant la résolution définie.
- Initialise le tableau **etats** de la même dimension pour contenir les états du système.
- Le programme remplit ensuite le tableau d'état du système en calculant l'état pour tous les temps de la simulation.

```
%
% Fichier : simuler_systeme.m
% Description : Cette fonction permet de simuler un système de deuxième ordre
%               pour une durée de 10 secondes avec des intervalles de temps
%                de 0.1 secondes. L'usager entre la pulsation propre et le
%               taux d'amortissement du système et le problème est résolue en
%                calculant la somme des exponentielles correspondantes.
%
% Entrées :
%     - AUCUN
%
% Sortie :
%     - temps, échelle de temps de la simulation
%     - etats, états du système au cours de la simulation
%

function [temps etats] = programme_principal()

    constantes;

    pulsation_propre = input('Entrez la pulsation propre; ');
    facteur_amorti = input('Entrez le facteur amorti; ');

    [tau_1 tau_2] = calculer_constantes_temps(pulsation_propre, facteur_amorti);
    [gain_1 gain_2] = calculer_gains(tau_1, tau_2);

    temps = 0:RESOLUTION:TEMPS_MAX;

    for ii=1:numel(temps)
        etats(ii) = simuler_systeme(gain_1, gain_2, tau_1, tau_2, temps(ii));
    end









end
```

3) Tracer un programme de tri (10 points)

Voici le programme d'un tri par insertion.

```
function tableau = tri_insertion(tableau)
% TRI_INSERTION trie le tableau recu en parametre a l'aide d'un tri par insertion.
%
% PARAMETRES : tableau : Le tableau de donnees a trier.
%
% ATTENTION : - La fonction doit recevoir un parametre, sinon elle leve
%               l'erreur suivante :
%               'La fonction doit recevoir un parametre'
%             - tableau doit etre un tableau de nombres reels, sinon la
%               fonction leve l'erreur suivante :
%               'Le parametre tableau doit etre une matrice de nombres reels'
%
% APPEL :
%           tableau = tri_insertion([-1 2 -5 9 12 3 17]);
%-------------------------------------------------------------------------

    % STRATEGIE : Pour chaque element i, on l'echange avec son precedent
    %             jusqu'a ce que les i premiers elements soit en ordre.

    % Validation du parametre recu.
    if (nargin == 0)
        error('La fonction doit recevoir un parametre');
    elseif (~isnumeric(tableau) | ~isreal(tableau))
        error('Le parametre tableau doit etre une matrice de nombres reels.');
    end;

    % Tri du vecteur par insertion.
    for indice = 2:numel(tableau)
        valeur_a_reculer = tableau(indice);

        % Pour chaque valeur tab(i) superieure a la valeur_a_reculer,
        % on copie cette valeur dans tab(i + 1).
        position = indice - 1;
        while (position >= 1 & tableau(position) > valeur_a_reculer)
            tableau(position + 1) = tableau(position);
            position = position - 1;
        end;
        % Ensuite, on copie la valeur_a_reculer dans la derniere valeur
        % tab(i) superieure trouvee.
        tableau(position + 1) = valeur_a_reculer;


        fprintf('%d : ', indice);
        fprintf('%d ', tableau);
        fprintf('\n');
    end;
return;
```

Vous devez tracer l'algorithme en indiquant ce qui apparaitra dans la fenêtre de commande, si l'appel suivant est lancé :

```
>> tab = tri_insertion([10 12 -5 0 4]);

2 : 10 12 -5 0 4
3 : -5 10 12 0 4
4 : -5 0 10 12 4
5 : -5 0 4 10 12
```

4) Programme de somme de dé (15 points)

Vous devez écrire une procédure appelée : simulation_somme_de, qui compte le nombre d'occurrences de chacune des valeurs pouvant être obtenues par la somme de deux dés à 100 faces (utilisé randi pour simuler un de et les valeurs possibles sont de 2 à 200). Au total, le programme simulera 10 000 lancés et affichera ensuite le nombre de fois que chacun des nombres aura été obtenu selon le standard qui suit.

Dans le cas où le nombre d'occurrences des sommes résultantes {2,3,4,..., 200} sont {15,23,42,... ,15}

```
>>simulation_somme_de()
Les résultats :
2 : 15
3 : 23
4 : 42
...
200 : 15
```

Notes :

i) Évidemment les résultats doivent être affichés pour toutes les valeurs de 2 à 200, des valeurs ont été sautées ici juste pour que l'exemple soit concis.

ii) Vous devez faire l'utilisation de constantes là où nécessaire.

```
%
% Fichier : simulation_somme_de.m
% Description : Cette procédure simule et affiche le résultat de la
%               simulation de la somme de deux dés à 100 faces, réalisé
%               10 000 fois.
%
% Entrées :
%     - AUCUN
%
% Sortie :
%     - AUCUN
%

function simulation_somme_de()


NB_TIR = 10000;
NB_FACES = 100;

de_1 = 0;
de_2 = 0;
somme_de = 0;

compte_de = zeros(NB_FACES*2-1,1);

for ii=1:10000

    de_1 = randi(NB_FACES);
    de_2 = randi(NB_FACES);
    somme_de = de_1+de_2

    compte_de(somme_de-1) = compte_de(somme_de-1)+1;

end


for ii=1:(NB_FACES*2-1)
    fprintf('%i : %i\n',ii, compte_de(ii+1));
end









end
```