

École de Technologie Supérieure

Devoir 3 – 02 avril 2015

Techniques de simulation numériques

SYS810

Milan Assuied

<https://www.sharelatex.com/project/>

1 Représentation d'état et simulation continue

1. Calcul de la représentation d'état commandable

Note: On réalise les simulations pour les formes canoniques commandables et observables. Les calculs sont détaillés dans ce document, en revanche, le paramétrage de la simulation est réalisé via les algorithmes de la classe Discretizer.

On souhaite simuler le système suivant:

$$G(s) = \frac{100s}{s^3 + 11s^2 + 30s + 200} \quad (1.1)$$

2. Calcul de la représentation d'état commandable:

Cette représentation est immédiate et découle de la fonction de transfert:

$$\boxed{\begin{cases} \dot{X} = \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ -200 & -30 & -11 \end{pmatrix} .X + \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} .U \\ Y = (0 \ 100 \ 0) .X + (0) .U \end{cases}}$$

3. Calcul de la représentation d'état observable:

La matrice A est identique. Il est en revanche nécessaire de calculer les paramètres β :

Il vient:

$$\begin{cases} \beta_3 = b_3 & = 0 \\ \beta_2 = b_2 - a_2 * \beta_3 & = 0 \\ \beta_1 = b_1 - a_2 * \beta_2 - a_1 \beta_3 & = 100 \\ \beta_0 = b_0 - a_2 * \beta_1 - a_1 \beta_2 - a_0 \beta_3 & = -1100 \end{cases}$$

On en déduit la représentation d'état observable:

$$\boxed{\begin{cases} \dot{X} = \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ -200 & -30 & -11 \end{pmatrix} .X + \begin{pmatrix} \beta_2 \\ \beta_1 \\ \beta_0 \end{pmatrix} .U \\ Y = (1 \ 0 \ 0) .X + (\beta_3) .U \end{cases}}$$

4. Simulation.

Le modèle suivant est réalisé pour la simulation:

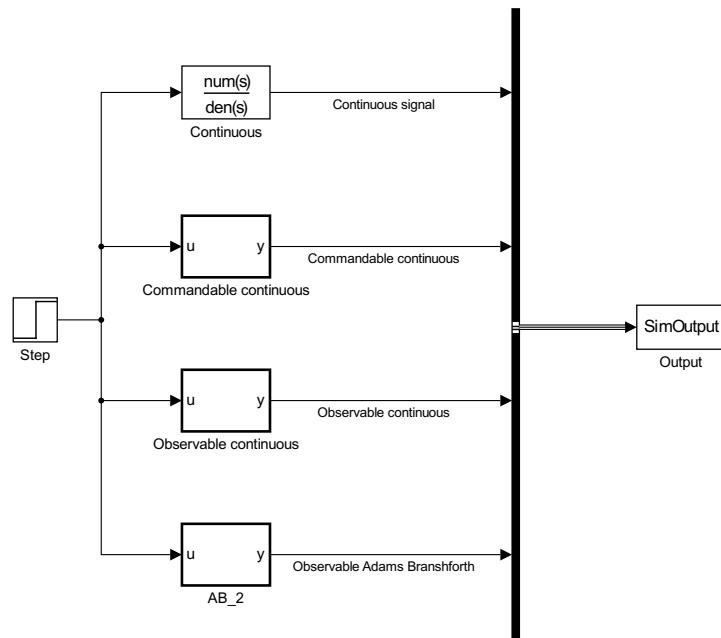


Figure 1: Modèle général

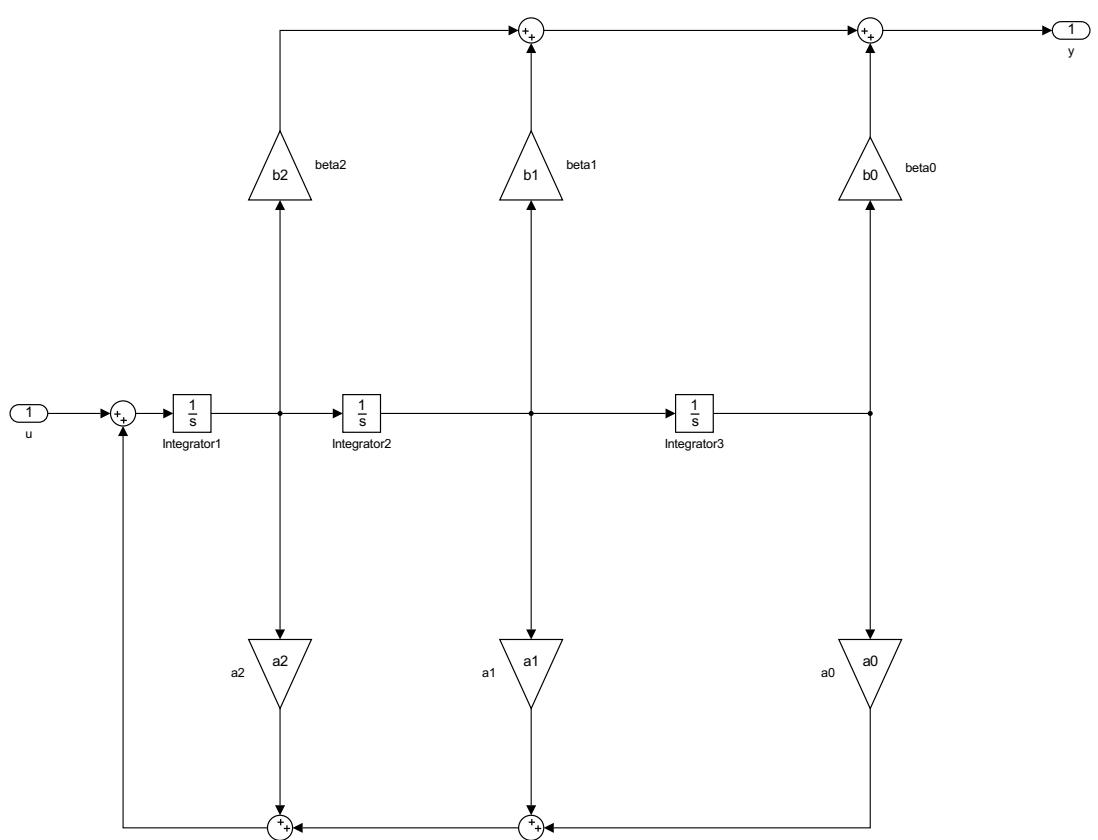


Figure 2: Représentation d'état commandable

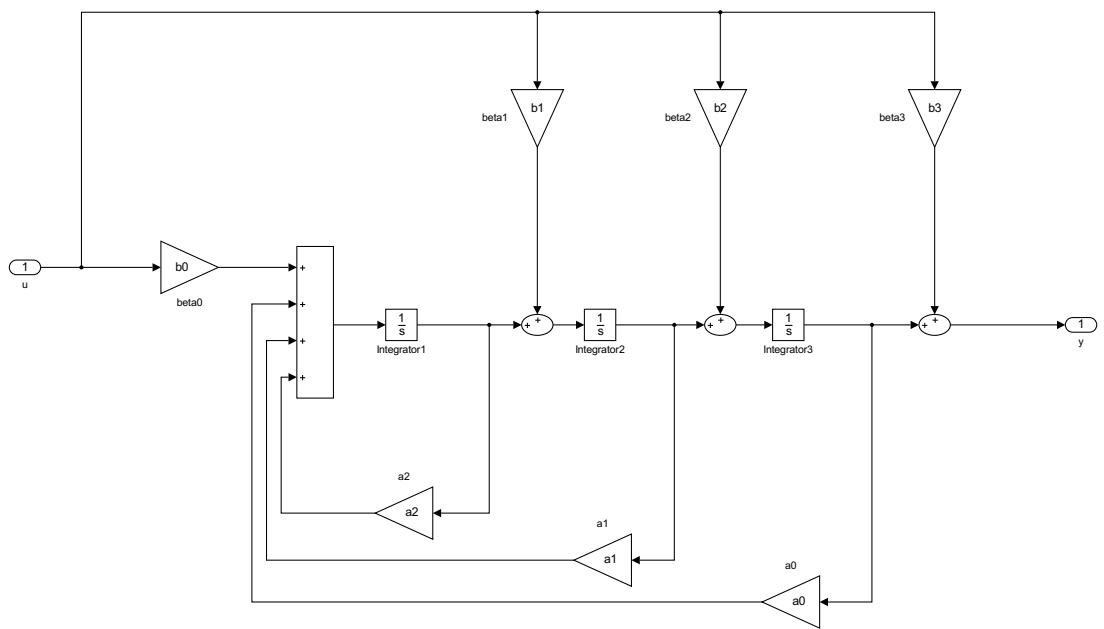
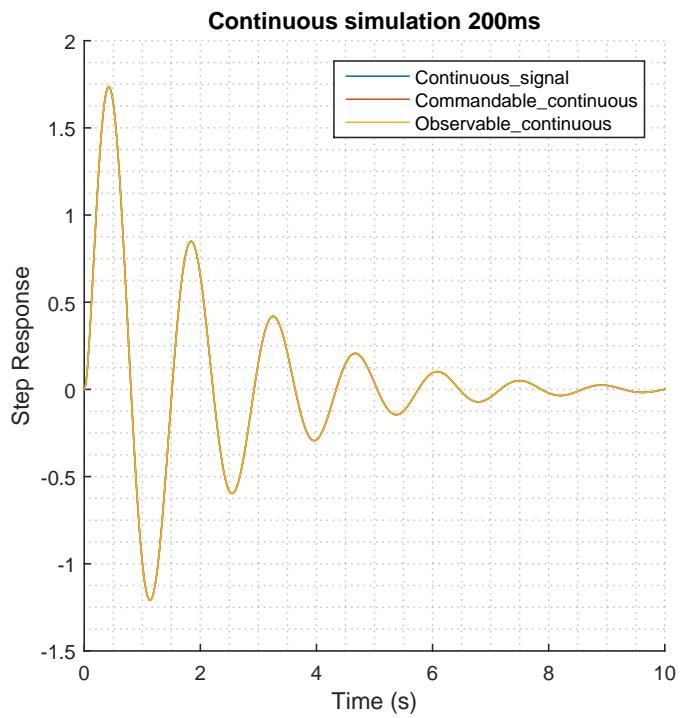


Figure 3: Représentation d'état observable

Le résultat suivant est obtenu:



2 Simulation avec la méthode d'Adam-Bashforth d'ordre 2

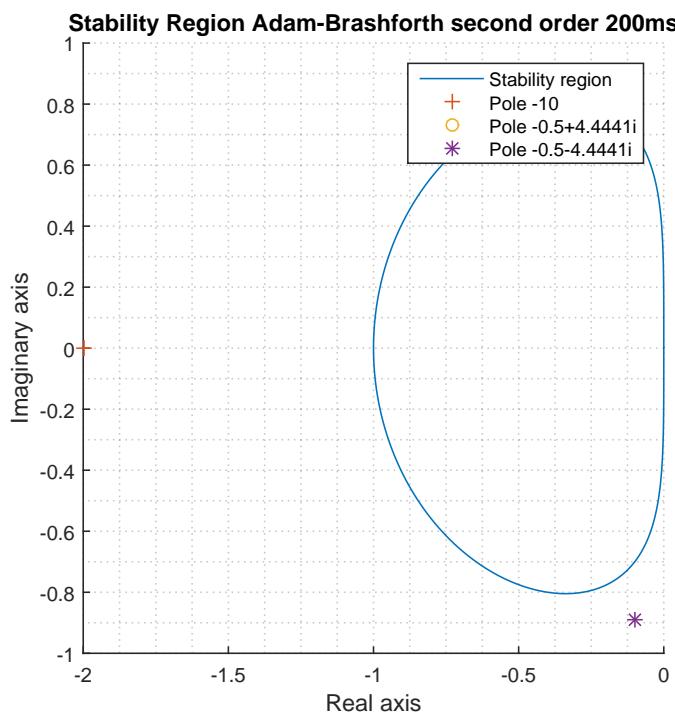
Le modèle choisi est le modèle observable, on remplace l'intégrateur continu par l'intégrateur discret de type AB_2 .

2.1 Étude de la stabilité en boucle fermée

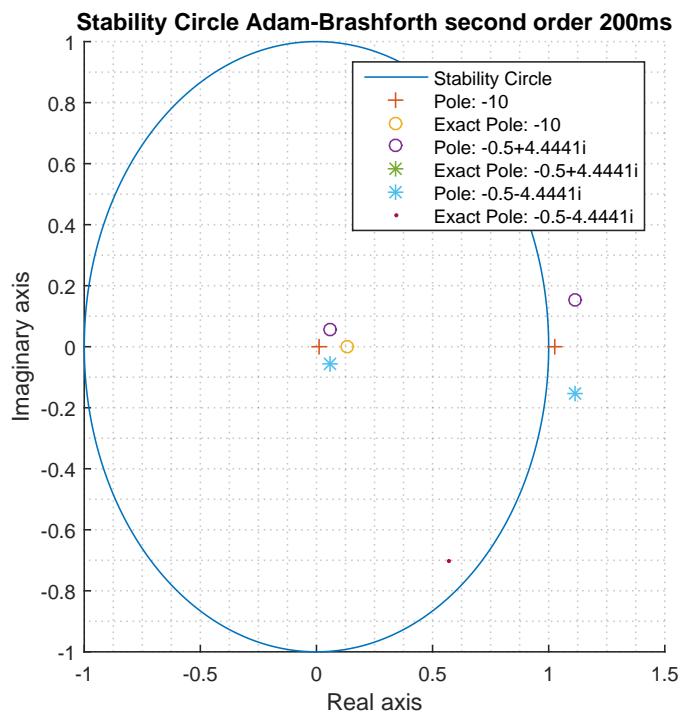
Afin de déterminer la stabilité en boucle fermée, on trace la région de stabilité et les pôles associés à différentes périodes d'échantillonnage, on trace également le cercle unitaire associé.

Note: Les algorithmes utilisés afin de réaliser les tracés sont disponibles dans la classe Discretizer, en annexe.

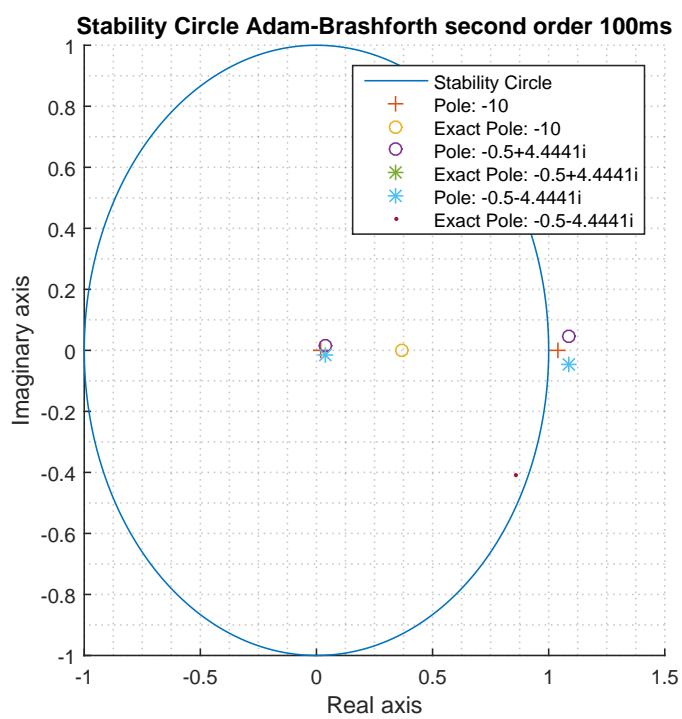
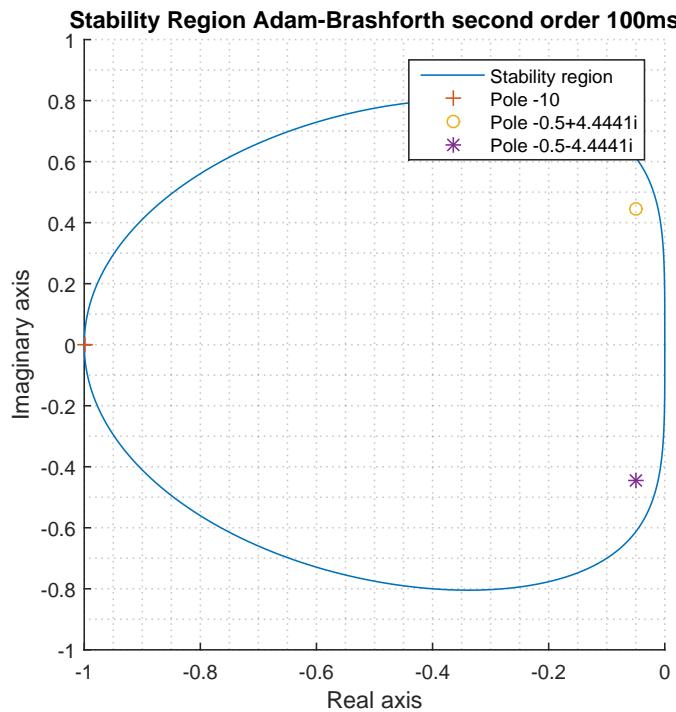
1. 200ms:



À 200ms, il est clair que la simulation n'est pas stable, elle est de plus divergente.



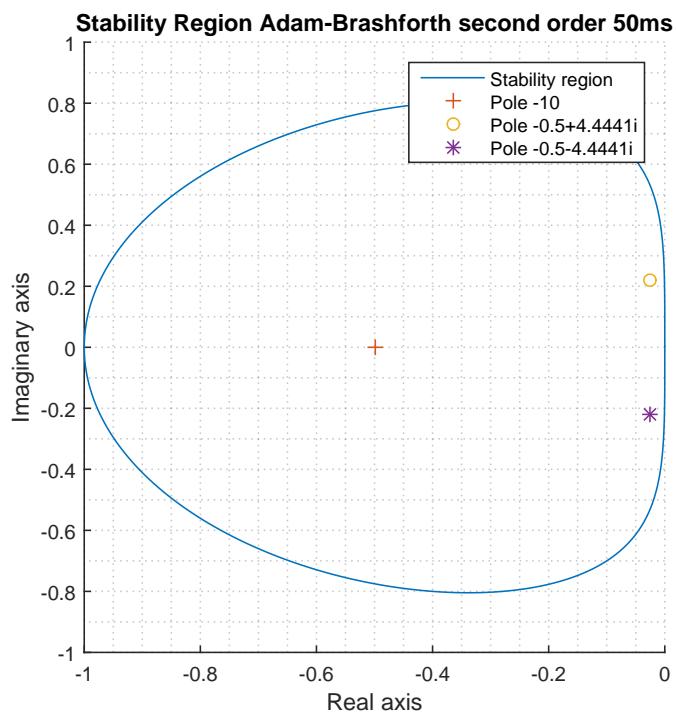
2. 100ms:

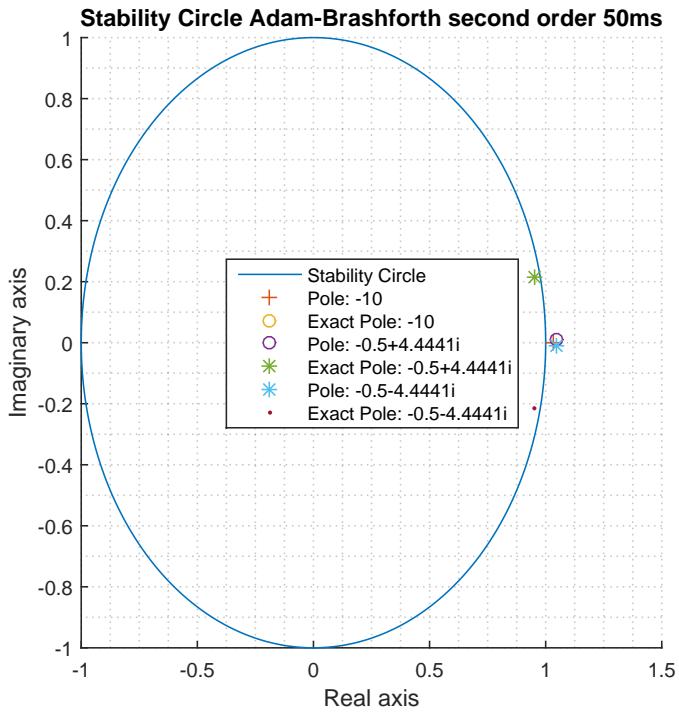


À 100ms, les pôles exacts du système sont compris dans la région de stabilité. Toutefois, on constate que si les pôles exacts sont également inclus dans le cercle unitaire, ce n'est pas le cas des pôles secondaires. De plus, ceux - ci sont éloignés des pôles exacts.

On s'attend donc à avoir une simulation qui ne diverge pas, est relativement stable, mais avec une très faible précision.

3. 50ms:





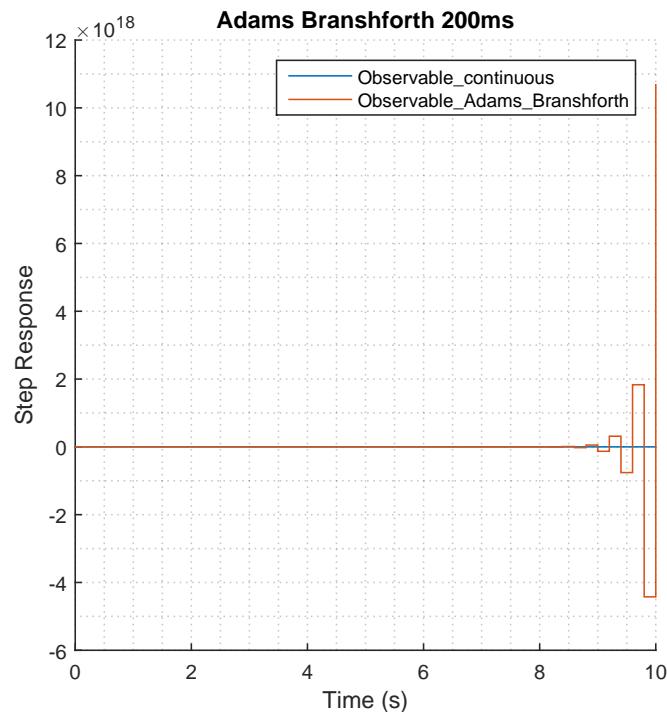
À 50ms, les pôles exacts du système sont compris dans la région de stabilité, les pôles secondaires sont presque localisés dans le cercle unitaire, et ceux-ci sont plus rapprochés des pôles exacts.

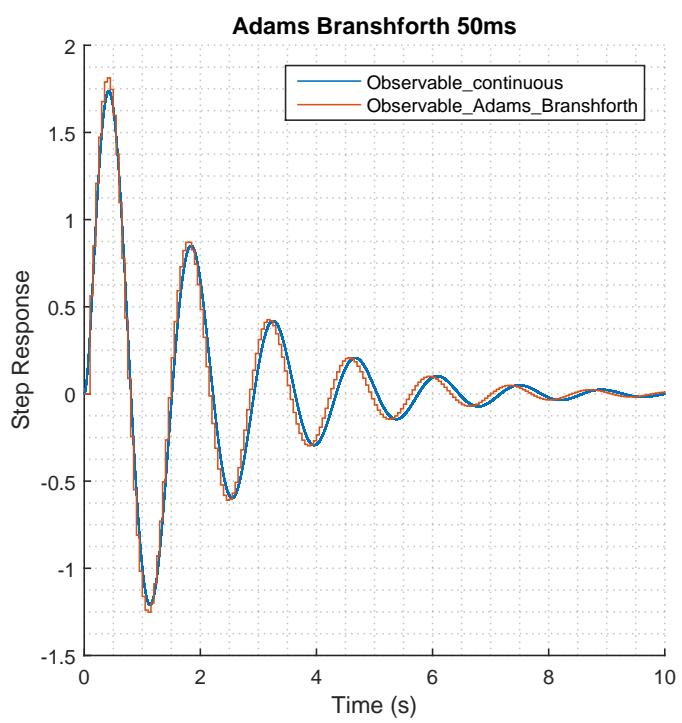
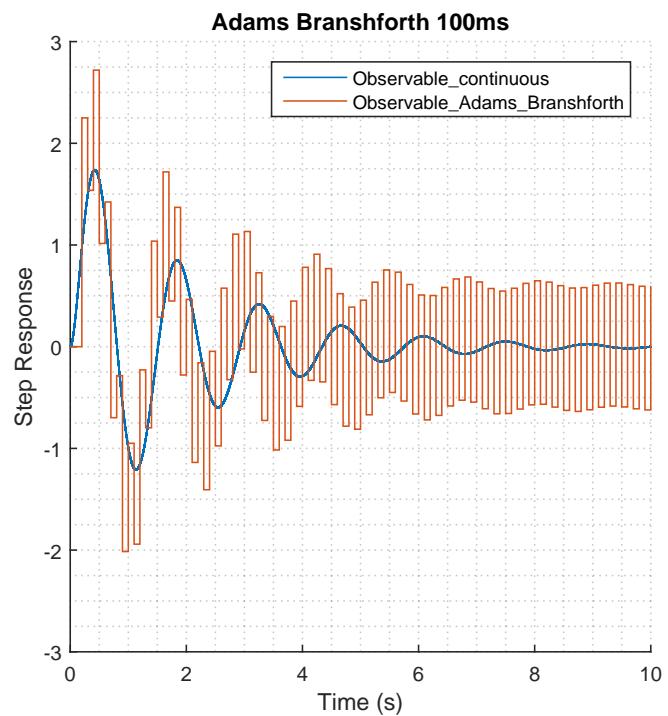
On s'attend donc à avoir une simulation d'une précision relativement correcte, avec toutefois une erreur prononcée. A ce stade, il n'est plus nécessaire de diminuer drastiquement la période d'échantillonnage, aussi nous choisissons 30ms comme période d'échantillonnage.

2.2 Résultats

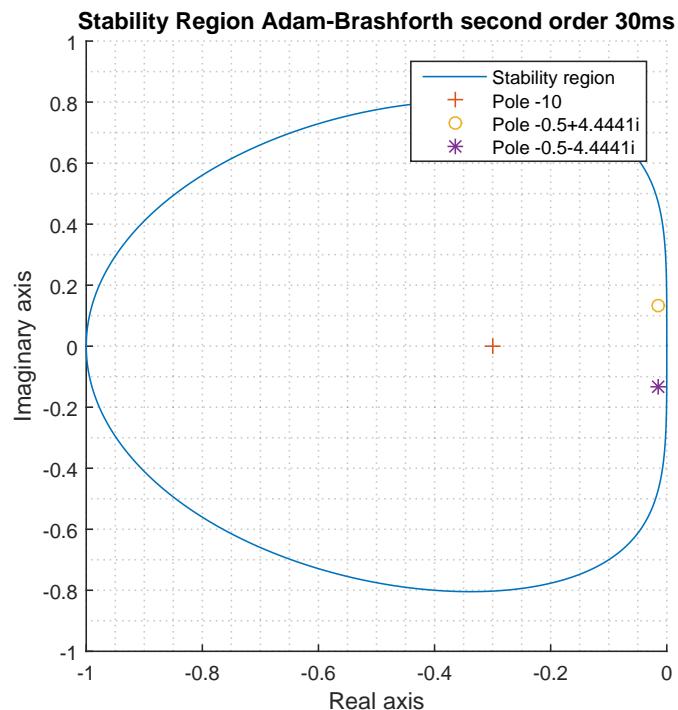
Les résultats de simulation effectués avec les périodes d'échantillonnage précédemment vues confirment les résultats qui étaient indiquées par les courbes obtenues:

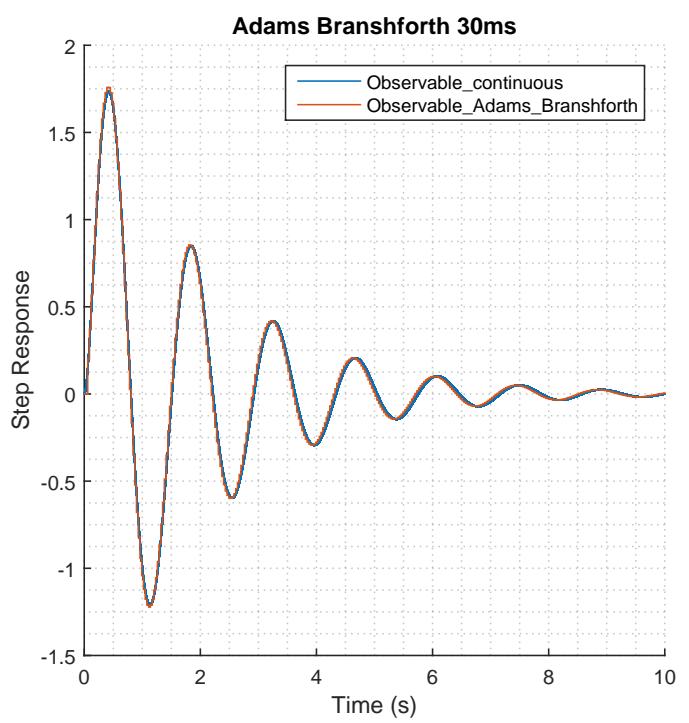
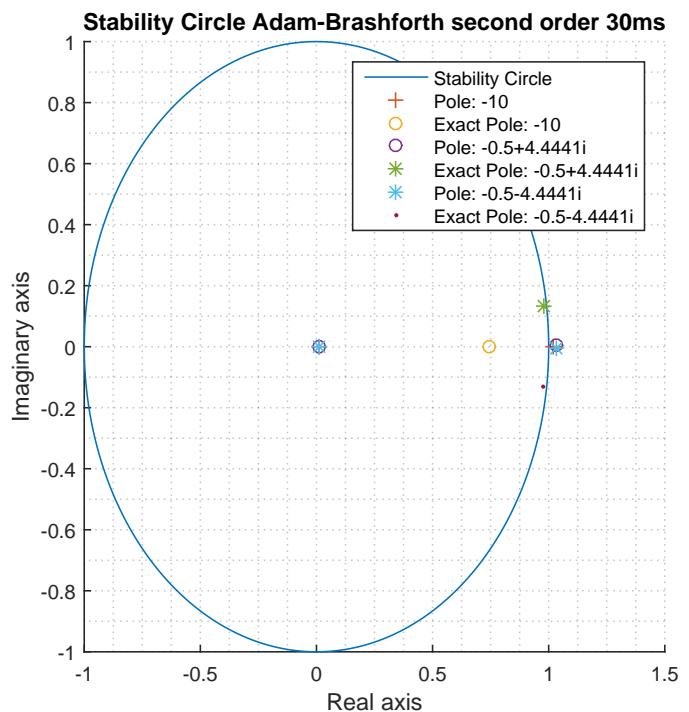
1. Simulation préliminaires:





2. Simulation à 30ms:





2.3 Conclusion:

Pour obtenir un résultat proche de la simulation continue, il est nécessaire de descendre à des périodes d'échantillonage qui sont environ 4 fois plus faibles que celles nécessaires avec la simulation continue. De plus, la simulation est beaucoup plus rapide. Ceci s'explique par le fait qu'un algorithme à pas variable permet d'optimiser la simulation en diminuant le pas lorsque nécessaire, et en l'augmentant lorsque cela est possible. Il est très probable que le pas utilisé pour le calcul en simulation continue soit descendu sous les $30ms$ durant la simulation, mais uniquement lorsque les dérivées varient fortement.

Cependant, ce mode de calcul n'est utilisable que pour les systèmes ne présentant pas ou peu de discontinuités / non-linéarités.

3 Simulation avec la méthode d'Adam-Bashforth d'ordre 2 couplé à un correcteur

3.1 Principe de calcul de la région de stabilité

On a établit que la fonction de transfert en boucle fermée valait:

$$G_{BF}(z) = \frac{T[\tau_c + \lambda T_s \beta_k \tau_p]}{(\rho_c - \lambda T_s \tau_c) + \lambda T_s \beta_k (\rho_p - \lambda T_s \tau_p)} \quad (3.1)$$

Avec:

$$\begin{cases} H_p(z) = \frac{T_s \rho_p(z)}{\tau_p(z)} \\ H_c(z) = \frac{T_s \rho_c(z)}{\tau_c(z)} \\ \beta_k \text{Le coefficient de plus haut degré de } H_c(z) \end{cases}$$

L'équation caractéristique s'écrit:

$$0 = \rho_c + \lambda T_s (\beta_k \rho_p - \tau_c) - (\lambda T_s)^2 \tau_p \equiv \rho_c + X(\beta_k \rho_p - \tau_c) - X^2 \tau_p \quad (3.2)$$

La région de stabilité est tracée en faisant varier z sur $[0; 2\pi]$ et en calculant les deux racines associées à chaque valeur de z .

Note: Voir code en annexe pour l'algorithme exact.

3.2 Détermination des équations de simulation

Ici le modèle choisi est le modèle commandable, les équations récurrentes sont calculées sur la base de cette représentation d'état.

De:

$$\begin{cases} H_p(z) = \frac{T_s(3z - 1)}{T_s(2z^2 - 2z)} \\ H_c(z) = \frac{T_s(z^2 - z)}{T_s(2z^2 - 2z)} \end{cases}$$

Il vient:

$$\begin{cases} Hp_{n+2} = Hp_{n+1} + \frac{T_s}{2} \cdot (3U_{n+1} - U_n) \\ Hc_{n+2} = Hc_{n+1} + \frac{T_s}{2} \cdot (U_{n+2} + U_{n+1}) \end{cases}$$

On en déduit les équations récurrentes suivantes:

$$\begin{cases} x1p_{n+2} = x1c_{n+1} + \frac{T_s}{2} \cdot (3f1c_{n+1} - f1c_n) \\ x2p_{n+2} = x2c_{n+1} + \frac{T_s}{2} \cdot (3f2c_{n+1} - f2c_n) \\ x3p_{n+2} = x3c_{n+1} + \frac{T_s}{2} \cdot (3f3c_{n+1} - f3c_n) \\ \\ f1p_{n+2} = x2p_{n+2} \\ f2p_{n+2} = x3p_{n+2} \\ f3p_{n+2} = -200.x1p_{n+2} - 30.x2p_{n+2} - 100.x3p_{n+2} + 1 \\ \\ x1c_{n+2} = x1c_{n+1} + \frac{T_s}{2} \cdot (f1p_{n+2} + f1c_{n+1}) \\ x2c_{n+2} = x2c_{n+1} + \frac{T_s}{2} \cdot (f2p_{n+2} + f2c_{n+1}) \\ x3c_{n+2} = x3c_{n+1} + \frac{T_s}{2} \cdot (f3p_{n+2} + f3c_{n+1}) \\ \\ f1c_{n+2} = x2c_{n+2} \\ f2c_{n+2} = x3c_{n+2} \\ f3c_{n+2} = -200.x1c_{n+2} - 30.x2c_{n+2} - 100.x3c_{n+2} + 1 \end{cases}$$

D'autre part, il convient de noter que la sortie vaut:

$$Y = (0 \ 100 \ 0) \cdot X + (0) \cdot U \equiv Y_n = 100.x2c_n \quad (3.3)$$

Finalement, les conditions initiales sont:

$$\begin{cases} f1c_0 = 0 \\ f2c_0 = 0 \\ f3c_0 = 1 \\ x1c_0 = 0 \\ x2c_0 = 0 \\ x3c_0 = 0 \end{cases}$$

Enfin, la causalité impose $\forall n < 2, f_n = 0 \wedge X_n = 0$ ce qui permet d'initialiser la récursion en calculant X_1 :

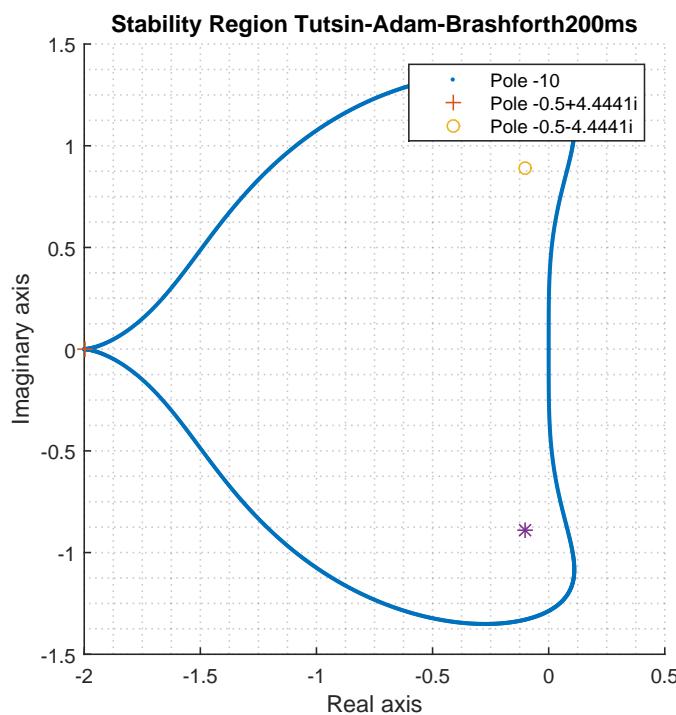
$$\begin{cases} x1p_1 = x1c_0 + \frac{T_s}{2} \cdot (3f1c_{n+1}) \\ x2p_1 = x2c_0 + \frac{T_s}{2} \cdot (3f2c_{n+1}) \\ x3p_1 = x3c_0 + \frac{T_s}{2} \cdot (3f3c_{n+1}) \end{cases}$$

3.3 Étude de la stabilité en boucle fermée

L'ajout d'un correcteur implicite à la méthode Adam-Bashforth explicite implique que la simulation sera stable à des périodes d'échantillonnage plus élevées. Nous faisons donc une simulation à 200ms, puis 100ms (à titre de comparaison), et enfin 50ms. Note: Les algorithmes utilisés afin de réaliser les tracés sont disponibles dans la classe Discretizer, en annexe.

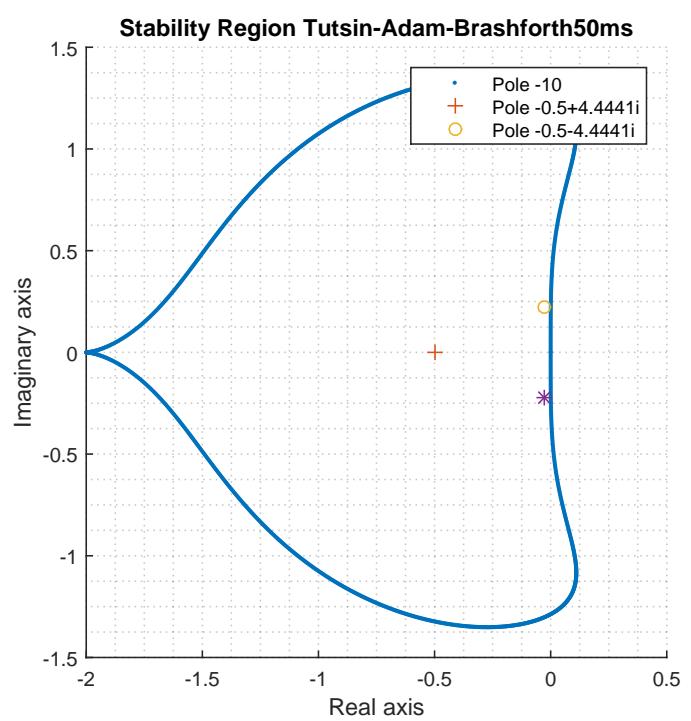
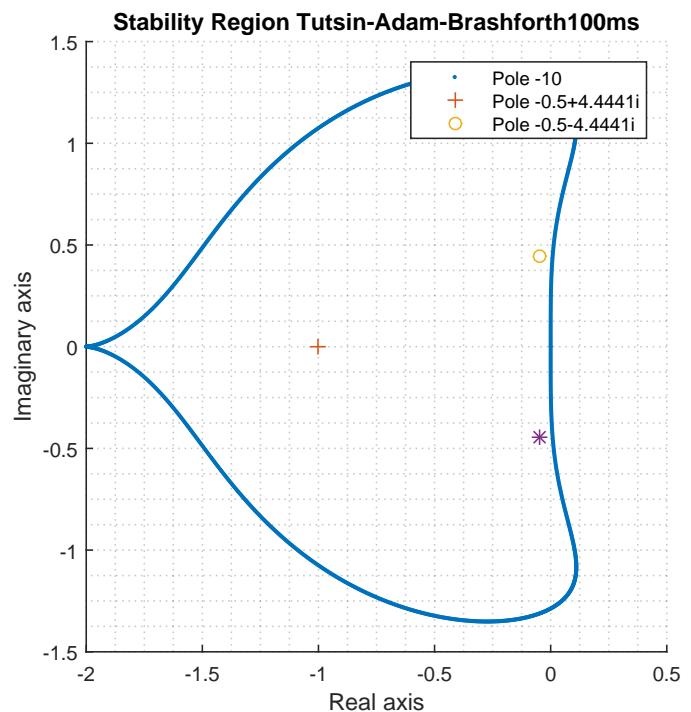
1. 200ms:

À 200ms, l'ensemble des pôles est situé en limite de la région de stabilité.



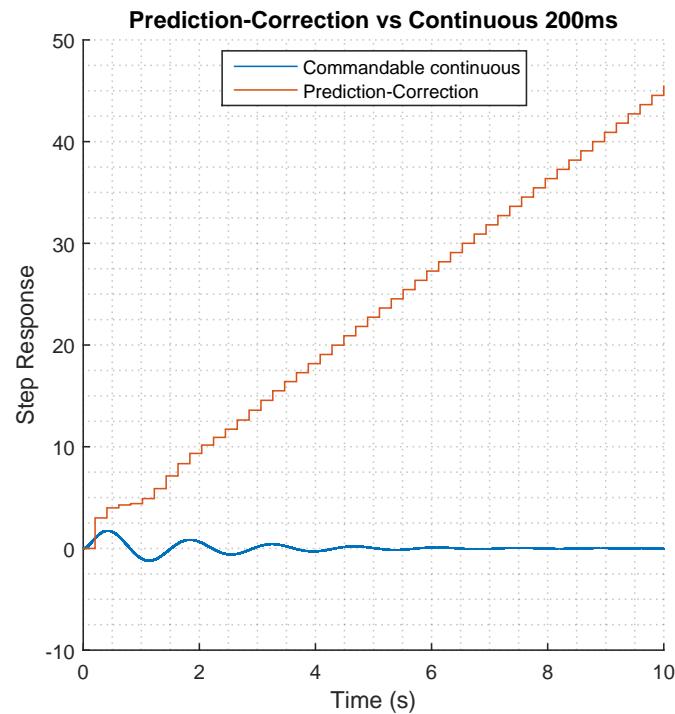
2. 100/50ms:

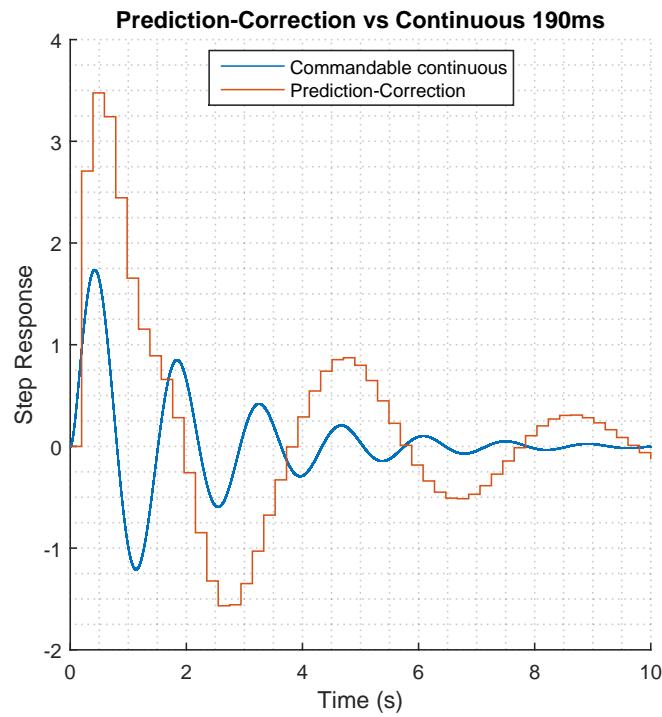
On sait que la simulation était stable avec la méthode AB_2 , on s'attend donc à retrouver une stabilité et à terme une meilleure précision, même si la stabilité n'est pas automatiquement acquise puisqu'elle dépend du choix du correcteur:



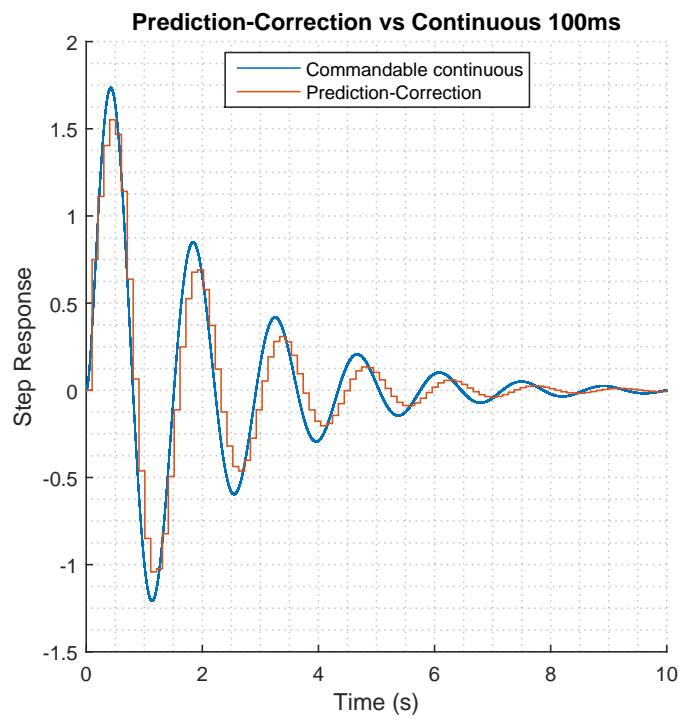
3.4 Résultats

Il est intéressant de constater que, malgré le tracé de la région de stabilité pour 200ms, la simulation diverge. Il apparaît que cette période d'échantillonage est située en limite de stabilité. Une simulation effectuée à 190ms montre de bien meilleurs résultats:

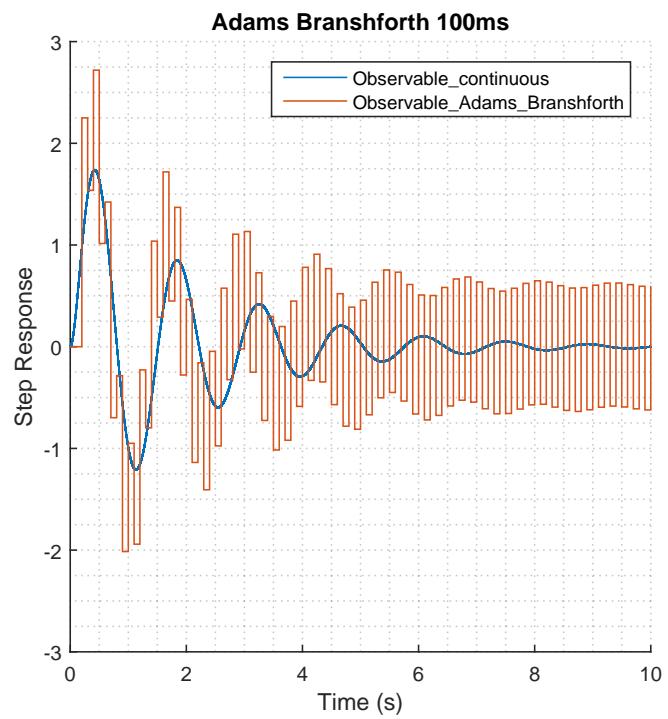




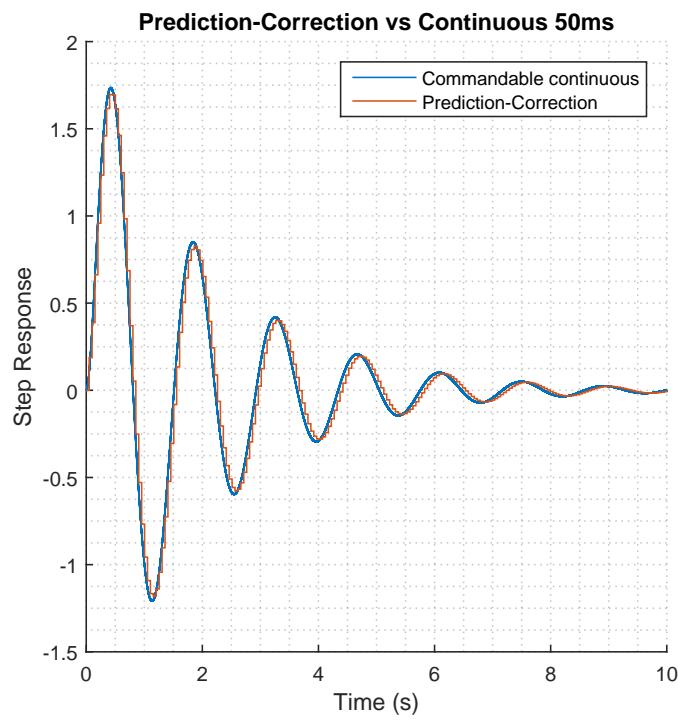
La simulation effectuée à 100ms montre comme attendue une amélioration très nette de la précision:



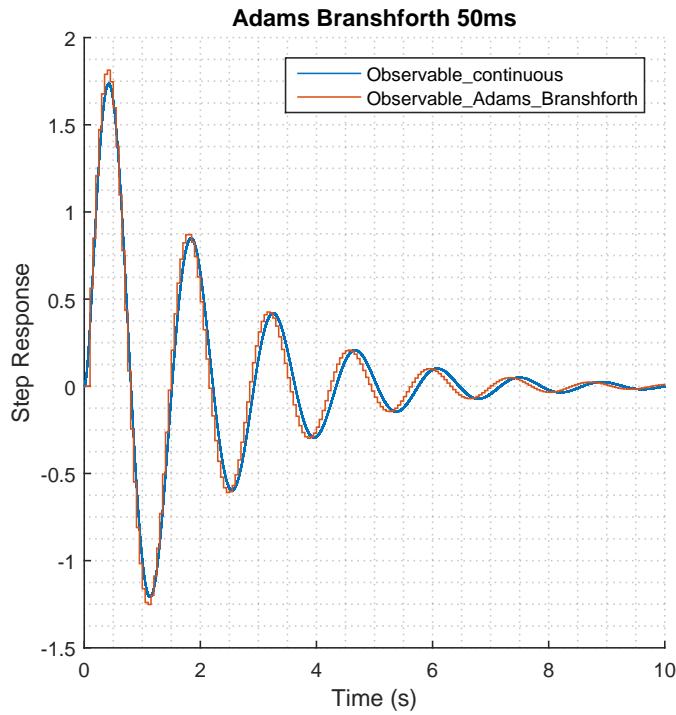
Pour rappel:



Finalement, à 50ms:



Pour rappel:



4 Conclusion

La comparaison effectuée entre la méthode AB_2 nue, et AB_2 couplée à un correcteur implicite met en évidence l'avantage de cette dernière méthode par rapport à la première.

En réduisant considérablement la période d'échantillonnage nécessaire pour obtenir un résultat similaire comparativement à la méthode nue, la méthode Predicteur-Correcteur permet de diminuer le temps de simulation, mais également la puissance de calcul nécessaire au déroulement de la simulation, et donc les coûts associés.

En revanche, il est nécessaire de considérer qu'une faible période d'échantillonnage peut être souhaitée afin de minimiser les ensembles de valeurs au sein desquels les sorties sont inconnues. Sous réserve d'une précision suffisante, la solution nue pourrait alors être plus avantageuse en termes de complexité d'implantation. En effet, à 50ms, on constate que l'écart de précision entre les deux techniques est plus faible, et il tend à se nullifier avec la diminution de T_s .

Le choix d'une technique ou d'une autre sera donc dicté par les contraintes du systèmes à planter, ainsi que de l'environnement de simulation, un arbitrage devra être fait entre une période d'échantillonnage plus faible, causant également des erreurs d'arrondi plus élevés, ou une période plus élevée, qui cause des "gaps" entre chaque point plus large, et qui nécessite l'emploi d'un correcteur.

ANNEXES: Code Matlab

1. Simulation Adam-Bashforth

```

49
50 myopts=simset('SrcWorkspace','current','DstWorkspace','current');
51
52 sim(model,wSimulationTime,myopts);
53 while (strcmp(get_param(model,'SimulationStatus'),'stopped')==0);
54 end
55
56 t_sim = toc;
57 fprintf('\nTemps de simulation => %3.3g s\n',t_sim)
58
59 %Drawing and saving Plots
60
61 wPloter.mDrawTimeseriesPlot([SimOutput.Continuous_signal...
62     ,SimOutput.Commandable_continuous...
63     ,SimOutput.Observable_continuous]...
64     ,['Continuous simulation ...
65     ',strrep(num2str(wSampleTime*1000),'.','.'), 'ms']...
66     ,'Time (s)'...
67     , 'Step Response');
68
68 wPloter.mDrawTimeseriesPlot([SimOutput.Observable_continuous...
69     ,SimOutput.Observable_Adams_Branshforth]...
70     ,['Adams Branshforth ...
71     ',strrep(num2str(wSampleTime*1000),'.','.'), 'ms']...
72     ,'Time (s)'...
73     , 'Step Response',...
74     , 'stairs');
75
75 close all;
76 end
77
78 %Saving Model
79 wPloter.mProcessSaveModel(model);
80 wPloter.mProcessSaveModel(wObsBlock);
81 wPloter.mProcessSaveModel(wComBlock);
82 wPloter.mProcessSaveModel(wAB2Block);

```

2. Simulation Predicteur-Correcteur:

```

1 addpath(genpath('..\..\..'));
2 dbstop if error
3
4 % **** * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * %
5 % * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * %
6 % * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * %
7
8 close all;
9 clear all; %#ok<CLSCR>
10 clc;
11
12 wSampleTimes=[0.3,0.2,0.19,0.18,0.17,0.16,0.15,0.1];
13 wSimulationTime=10;
14
15 wAdamsBashforth = tf([3,-1],[2,-2,0]);
16 wModifiedTutsin = tf([1 1 0],[2 -2 0]);
17
18 wContinuousSystemNum = [100,0];
19 wContinuousSystemDen = [1,11,30,200];
20
21 wPloter = Ploter([0 0 5 5],[5 5]);
22
23 for i=1:length(wSampleTimes)
24
25     close all;
26     java.lang.Runtime.getRuntime().freeMemory();
27
28     clearvars -except wSampleTimes wSystem wSimulationTime ...
29         wContinuousSystemNum...
30         wContinuousSystemDen wAdamsBashforthNum wAdamsBashforthDen ...
31             wPloter...
32         wAdamsBashforth wModifiedTutsin i eval(SimOutput)
33
34     wSampleTime = wSampleTimes(i);
35     wMaxStep=wSampleTime/1000;
36
37     wSystem = Discretizer(wSampleTime, ...
38         wContinuousSystemNum, ...
39         wContinuousSystemDen);
40
41     adamsFamillyModelSetup;
42
43     try
44         load(wSaveFileName);
45         Y = eval(wSaveFileName);
46     catch
47         h = errordlg(['Could not load', wSaveFileName, '...
48             Simulation to be executed once. Exiting.']);
49         waitfor(h);
50         break;
51     end

```

```

50      % **** PREDICTION - CORRECTION **** %
51      % ***** PREDICTION - CORRECTION ***** %
52      % ***** %
53
54      %Parameters
55      clear t f1c f2c f3c x1c x2c x3c f1p f2p f3p x1p x2p x3p;
56      t=linspace(0,wSimulationTime,wSimulationTime*(1/wSampleTime));
57
58      %Stability region
59      wSystem.mComputeStabilityRegion('Tutsin-Adam-Brashforth',wAdamsBashforth,wModif
60      %Initial conditions
61      f1c(1) = 0; f2c(1) = 0; f3c(1) = 1;
62      x1c(1) = 0; x2c(1) = 0; x3c(1) = 0;
63
64      for n = 0:wSimulationTime/wSampleTime-2
65
66          if (n == 0)
67              x1p(n+2) = x1c(n+1) + (wSampleTime/2)*(3*f1c(n+1)); ...
68                  %#ok<*SAGROW>
69              x2p(n+2) = x2c(n+1) + (wSampleTime/2)*(3*f2c(n+1));
70              x3p(n+2) = x3c(n+1) + (wSampleTime/2)*(3*f3c(n+1));
71          else
72              x1p(n+2) = x1c(n+1) + (wSampleTime/2)*(3*f1c(n+1)-f1c(n));
73              x2p(n+2) = x2c(n+1) + (wSampleTime/2)*(3*f2c(n+1)-f2c(n));
74              x3p(n+2) = x3c(n+1) + (wSampleTime/2)*(3*f3c(n+1)-f3c(n));
75          end
76
77          f1p(n+2) = x2p(n+2);
78          f2p(n+2) = x3p(n+2);
79          f3p(n+2) = -200*x1p(n+2) - 30*x2p(n+2) -11*x3p(n+2) + 1;
80
81          x1c(n+2) = x1c(n+1) + (wSampleTime/2)*(f1p(n+2)+ f1c(n+1));
82          x2c(n+2) = x2c(n+1) + (wSampleTime/2)*(f2p(n+2)+ f2c(n+1));
83          x3c(n+2) = x3c(n+1) + (wSampleTime/2)*(f3p(n+2)+ f3c(n+1));
84
85          f1c(n+2) = x2c(n+2);
86          f2c(n+2) = x3c(n+2);
87          f3c(n+2) = -200*x1c(n+2) - 30*x2c(n+2) -11*x3c(n+2) + 1;
88
89      end
90
91      % **** EXTRACTING FIGURES **** %
92      % ***** %
93
94      %Drawing and saving Plots
95
96      wPloter.mDrawStandardPlot({[Y.Observable_continuous.Time...
97          ,Y.Observable_continuous.Data]...
98          ,[t;100*x2c]}...
99          , 'stairs',...
100         , ['Prediction-Correction vs Continuous ...
101             ,strrep(num2str(wSampleTime*1000),'.',''), 'ms']...
102             , 'Time (s)'...
103             , 'Step Response'...

```

```
103      , {'Commandable continuous'; 'Prediction-Correction'});  
104  
105 close all;  
106 end
```

3. Code du traceur de courbes:

```
1 classdef Ploter < handle
2
3     properties (SetAccess = private, GetAccess = private)
4
5         mPaperPos;
6         mPaperSize
7         mHoldAll;
8         mSaveAfterDraw;
9
10    end
11
12    methods %Public
13
14        %Constructors
15        function oInstance = Ploter(iPaperPos, iPaperSize, varargin)
16
17            oInstance.mPaperPos = iPaperPos;
18            oInstance.mPaperSize = iPaperSize;
19
20            oInstance.mHoldAll = true;
21            oInstance.mSaveAfterDraw = true;
22
23        end
24
25        %Public drawers
26        function oHandle = ...
27            mDrawTimeseriesPlot(iThis, iValues, iTitle, iXlabel, iYlabel, varargin)
28
29            if isempty(varargin)
30
31                oHandle = ...
32                    iThis.mProcessTimeseriesPlot(@plot, iValues, iTitle, iXlabel, iYlabel);
33
34            else
35
36                oHandle = ...
37                    iThis.mProcessTimeseriesPlot(str2func(varargin{1}), iValues, iTitle, iXlabel, iYlabel);
38
39            end
40
41        function oHandle = ...
42            mDrawStandardPlot(iThis, iValues, iPlot, varargin)
43
44            oHandle = ...
45                iThis.mProcessStandardPlot(str2func(iPlot), iValues, varargin{:});
46
47        end
48
49        %Accessors
50        function mSetPaperPos(iThis, iPaperPos)
```

```

48         iThis.mPaperPos = iPaperPos;
49     end
50
51     function mSetPaperSize(iThis,iPaperSize)
52         iThis.mPaperSize = iPaperSize;
53     end
54
55     function mSetSaveAfterDraw(iThis,iValue)
56         iThis.mSaveAfterDraw = iValue;
57     end
58
59     %Save methods
60     function mProcessSaveDraw(iThis,iHandle)
61
62         set(iHandle, 'PaperPosition', iThis.mPaperPos);
63         set(iHandle, 'PaperSize', iThis.mPaperSize);
64         %waitfor(iHandle);
65         saveas(iHandle, iHandle.Name, 'pdf');
66
67     end
68
69     function mProcessSaveModel(iThis,iBlock)
70
71         saveas(get_param(iBlock,'Handle'), ...
72             strcat('SimBlock_',get_param(iBlock,'Name')), 'pdf');
73
74     end
75 end %Public methods
76
77 methods (Access = private)
78
79     function mSetHold(iThis)
80
81         if (iThis.mHoldAll)
82             hold all;
83         end
84
85     end
86
87     function mSetSaveDraw(iThis,iValue)
88
89         iThis.mSaveAfterDraw = iValue;
90     end
91
92     function mSaveDraw(iThis,iHandle)
93
94         if (iThis.mSaveAfterDraw)
95             iThis.mProcessSaveDraw(iHandle);
96         end
97
98     end
99
100    function oHandle = ...
101        mProcessTimeseriesPlot(iThis,iFuncHandle,iValues,iTitle,iXlabel,iYlabel);

```

```

101
102     oHandle=figure();
103
104     for i=1:length(iValues)
105
106         iThis.mSetHold();
107
108         iFuncHandle(iValues(i).Time,iValues(i).Data);
109         wLegend = ...
110             legend(get(legend(gca), 'String'),iValues(i).Name);
111
112     end
113
114     set(wLegend, 'Interpreter', 'none');
115
116     iThis.mProcessLabels(oHandle,iTitle,iXlabel,iYlabel);
117     grid minor;
118     iThis.mSaveDraw(oHandle);
119
120 function oHandle = ...
121     mProcessStandardPlot(iThis,iFuncHandle,iValues,varargin)
122
123     oHandle=figure();
124     oHandle.Name = func2str(iFuncHandle);
125     wPlotData = 0; %#ok<NASGU>
126
127     for i=1:length(iValues)
128
129         iThis.mSetHold();
130
131         wPlotData = iValues{i};
132
133         if (any(size(wPlotData) == 2))
134
135             if(isa(wPlotData,'cell'))
136                 %Case where we have a matrix and a character
137                 %specifier
138                 wMarker = wPlotData{2};
139                 wPlotData = wPlotData{1};
140
141             else
142                 wMarker = '-';
143             end
144
145             if (size(wPlotData,1) == 2)
146                 iFuncHandle(wPlotData(1,:),wPlotData(2,:),wMarker);
147             else
148                 iFuncHandle(wPlotData(:,1),wPlotData(:,2),wMarker);
149             end
150
151             elseif (size(wPlotData,2) == 1)
152                 iFuncHandle(wPlotData(1));
153             end
154
155     end

```

```

154
155     if (length(varargin) > 3)
156         legend(varargin{4}, 'Location', 'best');
157         iThis.mProcessLabels(oHandle, varargin{1:3});
158     else
159         iThis.mProcessLabels(oHandle, varargin{:});
160     end
161
162     grid minor;
163     iThis.mSaveDraw(oHandle);
164 end
165
166 function mProcessLabels(iThis, iHandle, varargin) %#ok<INUSL>
167
168     set(0, 'currentfigure', iHandle);
169
170     if (length(varargin) == 1)
171         title(varargin{1});
172         iHandle.Name = varargin{1};
173     elseif (length(varargin) == 2)
174         xlabel(varargin{1});
175         ylabel(varargin{2});
176     elseif (length(varargin) == 3)
177         title(varargin{1});
178         iHandle.Name = varargin{1};
179         xlabel(varargin{2});
180         ylabel(varargin{3});
181     end
182 end
183
184 end %Private methods
185 end %Class

```

4. Code du gestionnaire de systèmes:

```
1 classdef Discretizer < handle
2
3     properties (SetAccess = private, GetAccess = private)
4
5         mPlotMarkersList = ...
6             ['+', 'o', '*', '.', 'x', 's', 'd', '^', 'v', '>', '<', 'p', 'h'];
7
8         mT;
9         mContinuousTf;
10
11         %Matrices de transfer:
12         mQHalijak;
13         mQBoxer;
14
15     end
16
17     methods %Public
18
19         %Constructors
20         function oInstance = Discretizer(iT, varargin)
21             if nargin == 2
22                 oInstance.mContinuousTf = varargin{1};
23             elseif nargin == 3
24                 oInstance.mContinuousTf = tf(varargin{1}, varargin{2});
25             else
26                 oInstance.mContinuousTf = 0;
27                 h = errordlg('Invalid constructor');
28                 waitfor(h);
29             end
30             oInstance.mSetSampleTime(iT);
31
32         end
33
34         %Public methods
35
36         %Accessors
37
38         function mSetSampleTime(iThis, iSampleTime)
39             iThis.mT = iSampleTime;
40             mUpdateMatrix(iThis);
41         end
42
43         function oSampleTime = mGetSampleTime(iThis)
44             oSampleTime = iThis.mT;
45         end
46
47         function oMatrix = mGetHalijakMatrix(iThis, iRank)
48             oMatrix = iThis.mQHalijak{iRank};
49         end
50
51         function oMatrix = mGetBoxerThalerMatrix(iThis, iRank)
```

```

52     oMatrix = iThis.mQBoxer{iRank};
53 end
54
55 function oTf = mGetTf(iThis)
56     oTf = iThis.mContinuousTf;
57 end
58
59 %Compute discrete TF
60 function varargout = mGetDiscreteTf(iThis,iType)
61
62     wDiscreteTf = mProcessTf(iThis,iThis.mContinuousTf,iType);
63
64     if (nargout == 0) || (nargout == 1)
65         varargout{1} = wDiscreteTf;
66     elseif nargout == 2
67         [wHnum,wHden] = tfdata(wDiscreteTf,'v');
68         varargout{1} = wHnum./wHden(1);
69         varargout{2} = wHden./wHden(1);
70     else
71         h = errordlg('Invalid number of output arguments, ...
72                         valids outputs are (Tf) or (NumTf,DenTf)');
73         waitfor(h)
74     end
75
76
77 %Get poles for the specified model of system
78 function oSystemPoles = mGetPoles(iThis,iType)
79
80     wSystem = mGetDiscreteTf(iThis,iType);
81
82     oSystemPoles = pole(wSystem);
83
84 end
85
86 %Compute closed loop TF.
87 function varargout = mGetClosedLoop(iThis,iFeedBackTf,iType)
88
89     wDiscreteTf = mProcessTf(iThis,iThis.mContinuousTf,iType);
90     wDiscreteFeedBackTf = mProcessTf(iThis,iFeedBackTf,iType);
91
92     wCLTF = feedback(wDiscreteTf,wDiscreteFeedBackTf);
93
94     if (nargout == 0) || (nargout == 1)
95         varargout{1} = wCLTF;
96     elseif nargout == 2
97         [wHnum,wHden] = tfdata(wCLTF,'v');
98         varargout{1} = wHnum;
99         varargout{2} = wHden;
100    else
101        h = errordlg('Invalid number of output arguments, ...
102                         valids outputs are (Tf) or (NumTf,DenTf)');
103        waitfor(h)
104    end

```

```

105     end
106
107     %Compute discrete TF and apply retard
108     function varargout = ...
109         mGetRetardedDiscreteTf(iThis,iType,iRetard)
110
111         wHnum      = 0; %#ok<NASGU>
112         wHden      = 0; %#ok<NASGU>
113         wRetard    = ones(1,iRetard);
114
115         [wHnum,wHden] = mGetDiscreteTf(iThis,iType);
116         wHden          = [wRetard,wHden];
117
118         if (nargout == 0) || (nargout == 1)
119             varargout{1} = tf(wHnum,wHden,iThis.mT);
120         elseif nargout == 2
121             varargout{1} = wHnum;
122             varargout{2} = wHden;
123         else
124             h = errordlg('Invalid number of output arguments, ...
125                           valids outputs are (Tf) or (NumTf,DenTf)');
126             waitfor(h)
127         end
128     end
129
130     %Compute discrete TF and apply retard
131     function [A,B,C,D] = mGetStateSpaceMatrix(iThis,iType)
132
133         switch (iType)
134             case 'observable'
135
136                 [A,B,C,D] = iThis.mProcessObservableState();
137
138             case 'commandable'
139
140                 [A,B,C,D] = iThis.mProcessCommandableState();
141
142             otherwise
143                 h = errordlg('Invalid Type, returning null ...
144                               matrixes');
145                 waitfor(h)
146                 A = 0;
147                 B = 0;
148                 C = 0;
149                 D = 0;
150             end
151         end
152
153         %Execute recursion equation with the specified input, on the
154         %specified typed discrete function.
155
156         function Y = mComputeRecursion(iThis,U,iType)
157
158             [wNum,wDen] = iThis.mGetDiscreteTf(iType);
159             Y = iThis.mProcessRecursion(wNum,wDen,U);
160
161

```

```

157     end
158
159     %Stability study
160     function [oStabRegionHdl,oStabCircleHdl] = ...
161         mComputeStabilityRegion(iThis,iTitle,varargin)
162
163         if isempty(varargin)
164             h = errordlg('Error, this methods needs a transfer ...
165                         function in argument. Specify as a Tf object');
166             waitfor(h);
167             return;
168         end
169
170         if(length(varargin) == 1)
171
172             wIntegrator = Discretizer(1,varargin{:});
173             [wTauCoefficients,wLCoefficients] = ...
174                 wIntegrator.mGetDiscreteTf('continuous');
175
176             oStabRegionHdl = ...
177                 iThis.mProcessStabilityRegion(iTitle,wTauCoefficients,wLCoefficients);
178             oStabCircleHdl = ...
179                 iThis.mProcessStabilityCircle(iTitle,wTauCoefficients,wLCoefficients);
180
181         elseif(length(varargin) == 2)
182
183             wPredictor = Discretizer(1,varargin{1});
184             wCorrector = Discretizer(1,varargin{2});
185
186             [wTauPredictor,wRhoPredictor] = ...
187                 wPredictor.mGetDiscreteTf('continuous');
188             [wTauCorrector,wRhoCorrector] = ...
189                 wCorrector.mGetDiscreteTf('continuous');
190
191             oStabRegionHdl = ...
192                 iThis.mProcessPredictorCorrectorStabilityRegion(iTitle,wTauPredictor,wRhoPredictor);
193             oStabCircleHdl = 0;
194
195         end
196
197     end %Public methods
198
199     %Private methods
200     methods (Access = private)
201
202         %Updates all matrixes when requested
203         function mUpdateMatrix(iThis)
204             mUpdateBoxerThalerMatrix(iThis);
205             mUpdateHalijakMatrix(iThis);
206         end
207
208         %Halijak substitution matrix

```

```

204     function mUpdateHalijakMatrix(iThis)
205
206     mQ1=...
207         [iThis.mT,  0;...
208          1, -1];
209
210     mQ2=...
211         [0, iThis.mT^2, 0;...
212          iThis.mT, -iThis.mT, 0;...
213          1, -2, 1];
214
215     mQ3=...
216         [0, iThis.mT^3/2, iThis.mT^3/2, 0;...
217          0, iThis.mT^2, -iThis.mT^2, 0;...
218          iThis.mT, -2*iThis.mT, iThis.mT, 0;...
219          1, -3, 3, -1];
220
221     mQ4 =...
222         [0, iThis.mT^4/4, 2*iThis.mT^4/4, iThis.mT^4/4 ...
223          , 0;...
224          0, iThis.mT^3/2, 0, -iThis.mT^3/2, 0;...
225          0, iThis.mT^2, -2*iThis.mT^2, iThis.mT^2 ...
226          , 0;...
227          iThis.mT, -3*iThis.mT, 3*iThis.mT, ...
228          -iThis.mT, 0;...
229          1, -4, 6, -4, 1];
230
231
232 %Boxer Thalor substitution matrix
233 function mUpdateBoxerThalerMatrix(iThis)
234
235     mQ1=...
236         [iThis.mT/2, iThis.mT/2;...
237          1, -1];
238
239     mQ2=...
240         [iThis.mT^2/12, 10*iThis.mT^2/12, iThis.mT^2/12;...
241          iThis.mT/2, 0, -iThis.mT/2;...
242          1, -2, 1];
243
244     mQ3=...
245         [0, iThis.mT^3/2, iThis.mT^3/2, ...
246          0;...
247          iThis.mT^2/12, 9*iThis.mT^2/12, -9*iThis.mT^2/12, ...
248          -iThis.mT^2/12;...
249          iThis.mT/2, -1*iThis.mT/2, -1*iThis.mT/2, ...
250          iThis.mT/2;...
251          1, -3, 3, -1];
252
253     mQ4 =...
254         [-iThis.mT^4/720, 124*iThis.mT^4/720, ...

```

```

        474*iThis.mT^4/720, 124*iThis.mT^4/720, ...
        -iThis.mT^4/720;...
252          0 , iThis.mT^3/2 , 0 ...
                    , -iThis.mT^3/2 , 0;...
253          iThis.mT^2/12 , 8*iThis.mT^2/12 , ...
                    -18*iThis.mT^2/12 , 8*iThis.mT^2/12 , ...
                    iThis.mT^2/12;...
254          iThis.mT/2 , -iThis.mT , 0 ...
                    , iThis.mT , ...
                    -iThis.mT/2;...
255          1 , -4 , 6 ...
                    , -4 , 1];
256
257      iThis.mQBoxer = {mQ1, mQ2, mQ3, mQ4};
258
259  end
260
261 %Execute recursion equation with the specified input.
262 function Y = mProcessRecursion(iThis,num,den,U)
263
264     iThis; %#ok<VUNUS>
265
266     wTrimedDen = den(find(den,1):size(den,2));
267     wTrimedNum = num(find(num,1):size(num,2));
268
269     %How many iterations of Y are not computable.
270     %If the system is implicit, then wUΔ = 0
271     wUΔ = size(wTrimedDen,2)-size(wTrimedNum,2);
272     if (wUΔ < 0)
273         h = errordlg('Error, non-causal system');
274         waitfor(h);
275         return;
276     end
277
278     Y = zeros(1,wUΔ);
279
280     for i = size(Y,2)+1:size(U,2)
281
282         wY = 0;
283         wU = 0;
284
285         %Building input sum according to matlab 1-based ...
286         %indexing
287         for k = 1:size(wTrimedNum,2)
288             if(i-k+1-wUΔ > 0)
289                 wU = wU + wTrimedNum(k)*U(i-k+1-wUΔ);
290             else
291                 wU = wU + 0;
292             end
293         end
294
295         %Building output sum according to matlab 1-based ...
296         %indexing
297         for k = 1:size(wTrimedDen,2)-1
298             if((i-k > 0) && (i-k ≤ size(Y,2)))

```

```

297             wY = wY + wTrimedDen(k+1)*Y(i-k);
298         else
299             wY = wY + 0;
300         end
301     end
302
303     %Sum must be pondered by the highest numerator ...
304     %coefficient
305     Y(i) = 1/wTrimedDen(1) * (-wY + wU);
306
307 end
308
309 %Process TF conversions.
310 function oTf = mProcessTf(iThis,iTf,iType)
311
312     wTf      = iTf;
313     wHnum   = 0; %#ok<NASGU>
314     wHden   = 0; %#ok<NASGU>
315
316     if (strcmp(get(iTf,'Variable'),'s'))
317         switch (iType)
318             case 'zoh'
319
320                 wTf = c2d(iTf,iThis.mT,'zoh');
321
322             case 'tutsin'
323
324                 wTf = c2d(iTf,iThis.mT,'tutsin');
325
326             case {'halijak','boxerThaler'}
327
328                 [wHnum,wHden] = tfdata(iTf,'v');
329
330                 if(strcmp(iType,'halijak'))
331                     wH = ...
332                         [fliplr(wHnum);fliplr(wHden)]*iThis.mGetHalijakMatr
333                 else
334                     wH = ...
335                         [fliplr(wHnum);fliplr(wHden)]*iThis.mGetBoxerThalerM
336                 end
337
338                 wTf = tf(wH(1,:),wH(2,:),iThis.mT);
339
340             case 'continuous'
341                 %Return input
342
343             otherwise
344                 warning(['mProcessTf returning input ... ...
345
346             case 'zoh'

```

```

348
349             wTf = d2d(iTf,iThis.mT,'zoh');
350
351         case 'tutsin'
352             wTf = d2d(iTf,iThis.mT,'tutsin');
353
354         otherwise
355
356             warning('mProcessTf returning input ...
357                     (discrete) TF')
358         end
359
360     end
361
362     oTf = wTf;
363
364 end
365
366 %Process state space
367 function [A,B,C,D] = mProcessObservableState(iThis)
368
369     wOneBaseBias = 1;
370
371     [wNum,wDen,wOrder] = ...
372         iThis.mGetMatlabOrderedCoefficients();
373
374     A = iThis.mProcessCanonicalAMatrix();
375
376     %Compute beta values
377     wBetaMatrix = zeros(wOrder+1,1);
378     for k=wOrder:-1:0
379         s= 0;
380         for i=1:wOrder-k
381             s = s + ...
382                 wDen(wOrder-i+wOneBaseBias)*wBetaMatrix(k+i+wOneBaseBias);
383         end
384
385         wBetaMatrix(k+wOneBaseBias) = wNum(k+wOneBaseBias) ...
386             - s;
387     end
388
389     %Extract B & D matrixes from beta values. Note that ...
390     %beta matrix
391     %is ordered in matlab order (aka: ...
392     %[b(0);b(1);...b(n-1);b(n)])
393     %B needs to be inverted to be in the following order:
394     %[b(n-1),b(n-2),...b(0)]
395     B = wBetaMatrix(length(wBetaMatrix)-1:-1:1);
396     D = wBetaMatrix(length(wBetaMatrix));
397
398     %Compute C matrix.
399     C = [1,zeros(1,wOrder-1)];
400 end
401
402

```

```

397     function [A,B,C,D] = mProcessCommandableState(iThis)
398
399         wOneBaseBias = 1;
400
401         [wNum,wDen,wOrder] = ...
402             iThis.mGetMatlabOrderedCoefficients(); %#ok<ASGLU>
403
404         A = iThis.mProcessCannonicalAMatrix();
405
406         %Compute C values
407         C = zeros(1,wOrder);
408         for i=0:wOrder-1
409             C(1,i+wOneBaseBias) = wNum(i+wOneBaseBias);
410         end
411
412         %Compute B and D
413         B = [zeros(1,wOrder-1),1]';
414         D = 0;
415
416     end
417
418     function [A] = mProcessCannonicalAMatrix(iThis)
419
420         wOneBaseBias = 1;
421
422         [wNum,wDen,wOrder] = ...
423             iThis.mGetMatlabOrderedCoefficients(); %#ok<ASGLU>
424
425         %Compute A matrix
426         A = diag(ones(1,wOrder-1),1);
427         for i=0:wOrder-1
428             A(wOrder,i+wOneBaseBias) = -wDen(i+wOneBaseBias);
429         end
430
431     end
432
433     function [oNum,oDen,oOrder] = ...
434         mGetMatlabOrderedCoefficients(iThis)
435
436         %Get transfert function polynomials and set them in a ...
437             proper
438         %matlab order, opposed to polynomial order (aka ...
439             [a(0),a(1),...,a(n-1),a(n)])
440
441         %Also normalize the coefficients
442         [oNum,oDen] = tfdata(iThis.mGetTf());
443
444         oDen = fliplr(oDen{1});
445         oNum = fliplr(oNum{1});
446
447         oDen = oDen/oDen(length(oDen));
448         oNum = oNum/oDen(length(oDen));
449
450         oOrder = length(oDen)-1;
451
452

```

```

447     end
448
449     function ...
450         mAddPolesToStabilityRegion(iThis,iFigureHandle,iPloter)
451
452         set(0,'currentfigure',iFigureHandle);
453         wSystemPoles = iThis.mGetPoles('continuous');
454         wSampleTime = iThis.mGetSampleTime();
455
456         for k=1:length(wSystemPoles)
457
458             hold all;
459             wPlotMarkersListIndex = ...
460                 mod(k,length(iThis.mPlotMarkersList));
461             plot(real(wSystemPoles(k))*wSampleTime,imag(wSystemPoles(k))*wSampleTime);
462             legend(get(legend(gca),'String'),[{'Pole' ...
463                 ,num2str(wSystemPoles(k))}]);
464
465         end
466
467         iPloter.mProcessSaveDraw(iFigureHandle);
468     end
469
470     function oHandle = ...
471         mProcessStabilityRegion(iThis,iTitle,iTauCoefficients,iLCoefficients)
472
473         wZvalues = (exp(li*(0:.01:2*pi)))';
474         wPloter = Ploter([0 0 5 5],[5 5]);
475         wPloter.mSetSaveAfterDraw(false);
476
477         wStabilityValues = zeros(1,length(wZvalues));
478         for k = 1:length(wZvalues)
479
480             wStabilityValues(k) = ...
481                 polyval(iLCoefficients,wZvalues(k)) ./ ...
482                 polyval(iTauCoefficients,wZvalues(k));
483
484         end
485
486         oHandle = ...
487             wPloter.mDrawStandardPlot({[real(wStabilityValues);imag(wStabilityValues)];
488                 'plot',...
489                 ,['Stability Region' ...
490                     ,iTitle,strrep(num2str(iThis.mGetSampleTime()*1000),'.','.'), ...
491                     ,['Real axis',...
492                     ,['Imaginary axis',...
493                     ,['Stability region']]);
494
495         iThis.mAddPolesToStabilityRegion(oHandle,wPloter);
496     end
497
498     function oHandle = ...
499         mProcessPredictorCorrectorStabilityRegion(iThis,iTitle,iTauPredictor,iRho);
500
501         wZvalues = (exp(li*(0:.01:2*pi)))';

```

```

493 wPloter = Ploter([0 0 5 5],[5 5]);
494 wPloter.mSetSaveAfterDraw(false);
495
496 wBetaK = iTauCorrector(1);
497
498 wStabilityRoots = zeros(1,2*length(wZvalues));
499 for k=1:length(wZvalues)
500
501     wP1 = -wBetaK * polyval(iTauPredictor,wZvalues(k));
502     wP2 = wBetaK * polyval(iRhoPredictor,wZvalues(k)) ...
503         - polyval(iTauCorrector,wZvalues(k));
504     wP3 = polyval(iRhoCorrector,wZvalues(k));
505
506     wRoots = roots([wP1,wP2,wP3]);
507     wStabilityRoots(2*k-1) = wRoots(1);
508     wStabilityRoots(2*k) = wRoots(2);
509
510 end
511
512 oHandle= ...
513     wPloter.mDrawStandardPlot ({{[real(wStabilityRoots);imag(wStabilityRoots)]}}, ...
514         , 'plot'...
515         , ['Stability Region' ...
516             , iTitle,strrep(num2str(iThis.mGetSampleTime()*1000),'.','.'), 'm',...
517             , 'Real axis'...
518             , 'Imaginary axis']);
519
520 iThis.mAddPolesToStabilityRegion(oHandle,wPloter);
521
522 end
523
524 function oHandle = ...
525     mProcessStabilityCircle(iThis,iTitle,iTauCoefficients,iLCoefficients)
526
527     %Unit circle param
528     Wteta=0:0.001:2*pi;
529     wX=1*cos(Wteta);
530     wY=1*sin(Wteta);
531     wSampleTime = iThis.mGetSampleTime();
532
533     wLegendString = {'Stability Circle'};
534
535     wPloter = Ploter([0 0 5 5],[5 5]);
536     wSystemPoles = iThis.mGetPoles('continuous');
537
538     wExactSystemPoles = exp(wSystemPoles*wSampleTime);
539
540     wClosedLoopPoles = cell(1,2*length(wExactSystemPoles));
541
542     for k=1:1:length(wExactSystemPoles)
543
544         %Storing discret poles
545         wClosedLoopRoots = ...
546             roots(iLCoefficients-wExactSystemPoles(k)*wSampleTime*iTauCoefficients);
547
548         wClosedLoopPoles{k} = exp(wClosedLoopRoots);
549
550     end
551
552 end

```

```

543         wPlotMarkersListIndex = ...
544             mod(k,length(iThis.mPlotMarkersList));
545
546         wRealPart = zeros(1,length(wClosedLoopRoots));
547         wImagPart = zeros(1,length(wClosedLoopRoots));
548
549         for j=1:length(wClosedLoopRoots)
550             wRealPart(j) = real(wClosedLoopRoots(j));
551             wImagPart(j) = imag(wClosedLoopRoots(j));
552         end
553
554         wClosedLoopPoles{2*k-1} = ...
555             {[wRealPart;wImagPart],iThis.mPlotMarkersList(wPlotMarkersListIndex)};
556         wLegendString{2*k} = ['Pole: ...
557             ',num2str(wSystemPoles(k))];
558
559         %Storing exact pole on next index
560
561         wPlotMarkersListIndex = ...
562             mod(k+1,length(iThis.mPlotMarkersList));
563         wClosedLoopPoles{2*k} = ...
564             {[real(wExactSystemPoles(k));imag(wExactSystemPoles(k))],iThis.mPlotMarkersList(wPlotMarkersListIndex)};
565         wLegendString{2*k+1} = ['Exact Pole: ...
566             ',num2str(wSystemPoles(k))];
567
568         end
569
570     end
571
572     end %Private methods
573
574 end %Class

```

5. Configuration Simulink:

```
1 addpath(genpath('..\..\..\..\..'));
2 dbstop if error
3
4 % ***** SIMULINK INITIALIZATION *****
5 % ***** SIMULINK INITIALIZATION *****
6 % ***** SIMULINK INITIALIZATION *****
7
8 model='adamsFamilyModel';
9 load_system(model)
10 tic
11
12 wSaveFileName      = 'SimOutput';
13 wContBlock = strcat(model,'/Continuous');
14 wObsBlock = strcat(model,'/Observable continuous');
15 wComBlock = strcat(model,'/Commandable continuous');
16 wAB2Block = strcat(model,'/AB_2');
17
18 set_param(model,'StopFcn','save(wSaveFileName,wSaveFileName)');
19 set_param(strcat(model,'/Output'),'VariableName',wSaveFileName);
20
21 %Parametrazione simulation continue
22 set_param(wContBlock,'Numerator','wContinuousSystemNum');
23 set_param(wContBlock,'Denominator','wContinuousSystemDen');
24
25 set_param(wObsBlock,'a0','Ao(size(Ao,1),1)');
26 set_param(wObsBlock,'a1','Ao(size(Ao,1),2)');
27 set_param(wObsBlock,'a2','Ao(size(Ao,1),3)');
28
29 set_param(wObsBlock,'b0','Bo(3,1)');
30 set_param(wObsBlock,'b1','Bo(2,1)');
31 set_param(wObsBlock,'b2','Bo(1,1)');
32
33 set_param(wObsBlock,'b3','Do(1,1)');
34
35 set_param(wComBlock,'a0','Ac(size(Ac,1),1)');
36 set_param(wComBlock,'a1','Ac(size(Ac,1),2)');
37 set_param(wComBlock,'a2','Ac(size(Ac,1),3)');
38
39 set_param(wComBlock,'b0','Cc(1,1)');
40 set_param(wComBlock,'b1','Cc(1,2)');
41 set_param(wComBlock,'b2','Cc(1,3)');
42
43 %Parametrazione AB_2
44 set_param(wAB2Block,'a0','Ao(size(Ao,1),1)');
45 set_param(wAB2Block,'a1','Ao(size(Ao,1),2)');
46 set_param(wAB2Block,'a2','Ao(size(Ao,1),3)');
47
48 set_param(wAB2Block,'b0','Bo(3,1)');
49 set_param(wAB2Block,'b1','Bo(2,1)');
50 set_param(wAB2Block,'b2','Bo(1,1)');
51
52 set_param(wAB2Block,'b3','Do(1,1));
```

```
53
54 set_param(wAB2Block, 'T', 'wSampleTime');
55
56 set_param(wAB2Block, 'HzNum', 'wAdamsBashforthNum');
57 set_param(wAB2Block, 'HzDen', 'wAdamsBashforthDen');
58
59 set_param(model, 'StopTime', 'wSimulationTime');
60
61 set_param(model, 'MaxStep', 'wMaxStep');
```

————— *Fin du devoir* ————