Two-point boundary-value solver for stiff unstable linear systems (suited for application to ILC theory)

Stagerapport 2001.50

Two-point boundary-value solver for stiff unstable linear systems (suited for application to ILC theory)

door Bas Roset

Rapport van een interne stage
uitgevoerd van 23 april 2001 tot 20 september 2001

Begeleider:	M.J.G. van de Molengraft
Afstudeerhoogleraar:	M. Steinbuch

1 Introduction

In this report an algorithm to numerically solve a two-point boundary-value problem is proposed. The algorithm should deal with the differential equations used to compute the control signal (feedforward signal) for ILC (Iterative Learning Control) control strategy applied to linear motion control systems. These differential equations are often very stiff, unstable and contain a non-causal part. The problem of numerically solving the non-causal part is being taken care of in [4].
The remaining stiff and unstable causal part has to be taken care of in this report by means of a suited two-point boundary-value problem solver. By suited we mean a solver that can cope with; linear, stiff and unstable differential equations and approximating its solution within a reasonable accuracy and computing time.
A commonly used algorithm to deal with the instability and non-causality is ZPETC. ZPETC cancels the non-causality by a time shift between the input and output signal, however the method that ZPETC uses to deal with the instability leads to an inaccurate approximation of the amplitude behavior of the solution.
With a suitable two-point boundary-value problem solver this problem can be prevented and it offers also an additional advantage of control over the boundary conditions of the learnt signal. Especially in motion control it is desirable to have a feedforward signal (with its derivatives) to be zero at start and endpoint.
Previous investigations have shown that the current available two-point boundary-value solvers such as solvers based on single or multiple shooting techniques do not perform satisfactory.

1.1 Problem description and approach

We want to obtain a numerical solution of the following *unstable* time-invariant (A=constant) inhomogeneous linear set of n coupled first order differential equations between point t=a and t=b (a<b).

$$\frac{d}{dt}\underline{x}(t) = A\underline{x}(t) + Bu(t) \qquad a \le t \le b \qquad (1)$$

With non-separated boundary-values:

$$B_a \underline{x}(t=a) + B_b \underline{x}(t=b) = \underline{d} \qquad (2)$$

Eq. (2) is the general description in which the initial-value(s) and the end-value(s) in respectively point t=a and t=b are prescribed. B_a and B_b are constant nxn matrices that should satisfy the following condition:

$$\{\, rank(B_a) + rank(B_b) = \dim(A) = n \,\} \qquad (3)$$

If Eq. (3) is not satisfied, Eq. (1) is either underdetermined or over-determined and cannot be solved. Standard numerical MATLAB solvers, such as the ODE-solvers, are only capable in solving initial-value problems (B_b=0). In order to obtain the numerical solution of Eq. (1) with its unstable modes and with Eq. (2) as boundary-values some more computation effort has to be preformed.
This report, in which an attempt is made to create a solver that can cope with the problem as sketched, the following parts can be distinguished:

- Chapter 2
  -Paragraph 2.1: Explains why the two-point boundary-value solvers based on single or multiple shooting techniques do not perform satisfactory.
  -Paragraph 2.2: Introduces a time dependent transformation, which transforms Eq. (1) with state variables x(t) into another set of equations with state variables y(t). The new obtained set of equations consist of a decoupled unstable and stable part. This decoupling property can be used to overcome numerical problems which the shooting techniques are struggling with.

-Paragraph 2.3: Explains how the decoupling technique can be applied to a special class of boundary-value problem namely the *separated boundary-value problem*
-Paragraph 2.4: Explains a technique that is a proposed solution to a complication that is caused by the transformation explained in paragraph 2.2.
-Paragraph 2.5: Shows how the general *non-separated boundary-value problem* defined by Eq. (1) and Eq. (2) can be solved by the superposition of linear independent solutions of the *separated boundary-value problem.*

- Chapter 3: Explains some practical aspects about the implementation of the theory described in chapter 2. Chapter 3 also includes an explanation of how the algorithm should be used.
- Chapter 4,5: Show some examples of *non-separated boundary-value problem*s numerically solved by the proposed solver. Also a comparison is made to other potential suitable solvers from which some conclusions can be drawn.

2 Numerical solution of the Linear Boundary Value Problem

2.1 Shooting method (superposition method)

Because Eq. (1) is linear one can split the solution of Eq. (1) $\underline{x}(t)$ into a homogeneous solution $\underline{x}_h(t)$ and a particular solution $\underline{x}_p(t)$.

$$\underline{x}(t) = \underline{x}_h(t) + \underline{x}_p(t) \tag{4}$$

The particular solution can be evaluated by numerically integrating the following differential equation.

$$\frac{d}{dt}\underline{x}_p(t) = A\underline{x}_p(t) + Bu(t) \qquad a \le t \le b \quad \text{with initial conditions: } \underline{x}_p(t=a) = \underline{0} \tag{5}$$

The homogeneous solution consists of a linear combination of arbitrary linear independent solutions of the homogeneous equation of Eq. (1). So one can write:

$$\underline{x}_h(t) = \Phi_x(t)\underline{v} \tag{6}$$

In which $\Phi_x(t)$ is a nxn matrix function (transition matrix) that contains the n arbitrary linear independent solutions of the homogeneous equation (fundamental solutions). $\underline{v}$ is a superposition vector that contains the coefficients that define the linear relation between the arbitrary linear independent solutions of the homogeneous equation and the homogeneous solution $\underline{x}_h(t)$.
The n arbitrary independent solutions of the homogeneous equation of Eq. (1) are obtained by integrating Eq. (7) n times with n arbitrary linear independent initial conditions.

$$\frac{d}{dt}\underline{x}_{h,i}(t) = A\underline{x}_{h,i}(t) \qquad a \le t \le b \tag{7}$$

For example $[\underline{x}_{h,1}(a),\ \underline{x}_{h,i}(a),\ \cdot,\ \cdot,\ \underline{x}_{h,n}(a)] = I_n$

The superposition vector $\underline{v}$ is obtained by substituting Eq. (6) into Eq. (4) and letting Eq. (4) satisfy the boundary conditions Eq. (2).

$$B_a\left(\Phi_x(a)\underline{v} + \underline{x}_p(a)\right) + B_b\left(\Phi_x(b)\underline{v} + \underline{x}_p(b)\right) = \underline{d}$$
$$\underline{v} = \left(B_a\Phi_x(a) + B_b\Phi_x(b)\right)^{-1}\left(\underline{d} - B_a\underline{x}_p(a) - B_b\underline{x}_p(b)\right) \tag{8}$$

The major problem of the shooting method is that it evaluates Eq. (1), which is unstable, by solving IVP's (Initial Value Problems). So in general, after a certain time between a and b, the numerical solution of the IVP will blow up and the obtained solution will be inaccurate.
A multiple shooting technique also uses IVP's but for just a subinterval $[t_i, t_{i+1}]$ $(i=0,\ldots,m-1)$,where $a = t_a < t_1 < \cdots < t_{m-1} < t_b = b$
The idea is that the IVP's will be evaluated for just a short time, so the unstable increasing modes will not reach the unacceptable large grow factors that lead to numerical inaccuracy. After finding all the sub-solutions of the IVP's one has to solve a linear algebraic equation (one that resembles Eq. (8)). The dimension of that algebraic equation is m. The drawback of this technique, and the reason why it cannot be used for our application, is that for very stiff unstable systems the amount of subintervals and so also the dimension (m) of the linear algebraic equation becomes unacceptably large.

2.2 Continuous decoupling

If the solution space of Eq. (1) would only contain unstable solutions (increasing modes) or only stable solutions (decreasing modes) then the shooting method would probably be effective, because one could obtain the solutions respectively by solving EVP's (End Value Problem) or IVP's. Unfortunately this is not the case in general.
However one can create a decoupling method [2], which is capable of decoupling the increasing and decreasing modes of the solution space of Eq. (1).
Consider a homogeneous ODE of order n Eq (5). Any arbitrary solution consists of linear combinations of exponential functions, which can be reorganized into two vectors (1xn) $\underline{p}_1(t)$ and $\underline{p}_2(t)$.

$\underline{p}_1(t)$ And $\underline{p}_2(t)$ contain the exponential functions that are respectively responsible for the increasing and decreasing modes.

$$\left|\underline{p}_1(t)\right| \sim e^{\psi t} \quad , \qquad \left|\underline{p}_2(t)\right| \sim e^{-\xi t} \qquad (\psi \geq 0 , \xi \geq 0)$$

To make the problem easy to represent geometrically we will now consider the ODE to be 2-dimensional, but the theory also holds for n-dimensional ODE's.
For some arbitrary solution $\underline{x}_{h,1}(t)$ of the ODE it holds that it consist of a linear combination of $\underline{p}_1(t)$ and $\underline{p}_2(t)$ containing respectively the unstable and stable modes, so:

$$\underline{x}_{h,1}(t) = \alpha_1 \underline{p}_1(t) + \alpha_2 \underline{p}_2(t) \tag{9}$$

To obtain the total fundamental solution $\Phi_x(t)$ of the ODE (n=2) one needs another arbitrary solution linear independent of $\underline{x}_{h,1}(t)$ (see shooting method):

$$\underline{x}_{h,2}(t) = \beta_1 \underline{p}_1(t) + \beta_2 \underline{p}_2(t) \tag{10}$$

For the geometric relation between the vectors see figure 1a.

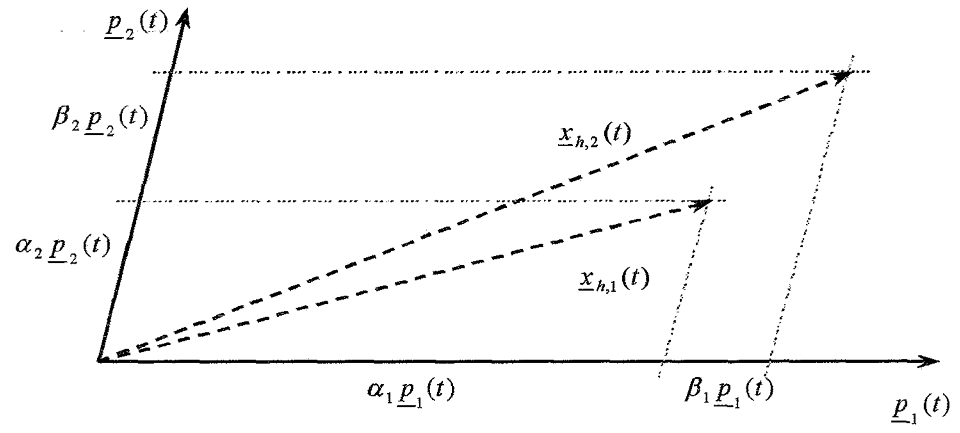

$\underline{p}_2(t)$
$\beta_2 \underline{p}_2(t)$
$\alpha_2 \underline{p}_2(t)$
$\underline{x}_{h,2}(t)$
$\underline{x}_{h,1}(t)$
$\alpha_1 \underline{p}_1(t)$
$\beta_1 \underline{p}_1(t)$
$\underline{p}_1(t)$

figure 1a Geometric interpretation: $\underline{x}_{h,1}(t)$ and $\underline{x}_{h,2}(t)$ as linear combination of $p_1(t)$ and $p_2(t)$

$\underline{x}_{h,1}(t)$ will grow asymptotically like $\underline{p}_1(t)$ (unless $\alpha_1 = 0$) and will asymptotically have the same direction as $\underline{p}_1(t)$. $\underline{x}_{h,2}(t)$ will also grow asymptotically like $\underline{p}_1(t)$ (unless $\beta_1 = 0$) and also will asymptotically have the same direction as $\underline{p}_1(t)$. Conclusion: The obtained solutions $\underline{x}_{h,1}(t)$ and $\underline{x}_{h,2}(t)$ become numerically dependent as t becomes large, this is precisely why superposition used in the shooting method fails.
To prevent this problem one would like to create two (n) vectors instead of $\underline{x}_{h,1}(t)$ and $\underline{x}_{h,2}(t)$ which do not get dependent when t becomes large. A suitable solution would be to create a pair of vectors that stay orthogonal when t proceeds. This can be accomplished as follows:

Let $\underline{t}_1(t)$ be a unit vector in the direction of $\underline{x}_{h,1}(t)$.

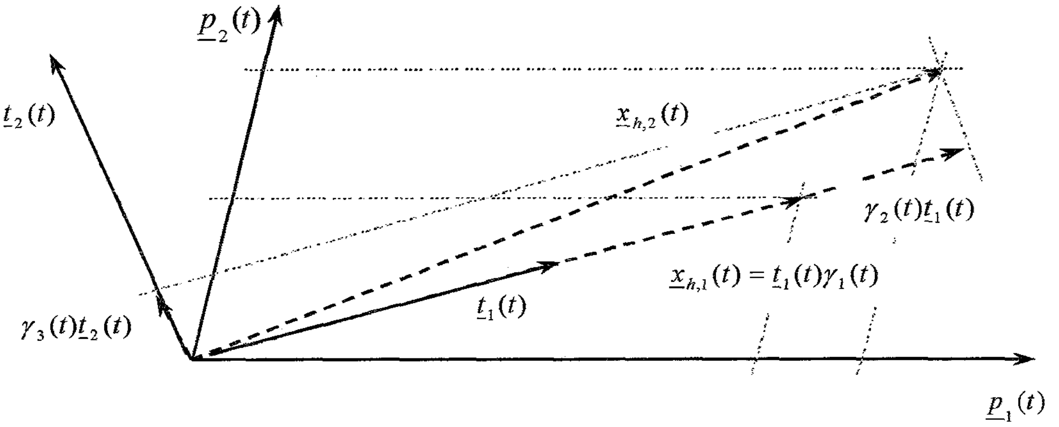
$$\underline{t}_1(t) = \frac{\underline{x}_{h,1}(t)}{\gamma_1(t)} \qquad \underline{\gamma}_1(t) = \left|\underline{x}_{h,1}(t)\right| \tag{11}$$

Then, let $\underline{t}_2(t)$ be a unit vector orthogonal to $\underline{t}_1(t)$. Now we can write the following linear relation of $\underline{t}_1(t)$ and $\underline{t}_2(t)$ with respect to $\underline{x}_{h,2}(t)$:

$$\underline{x}_{h,2}(t) = \gamma_2(t)\underline{t}_1(t) + \gamma_3(t)\underline{t}_2(t) \tag{12}$$

Combining Eq. (9),(10),(11) and Eq. (12) gives:

$$[\underline{x}_{h,1}(t) \quad \underline{x}_{h,2}(t)] = [\underline{p}_1(t) \quad \underline{p}_2(t)]\begin{bmatrix}\alpha_1 & \beta_1\\ \alpha_2 & \beta_2\end{bmatrix} = [\underline{t}_1(t) \quad \underline{t}_2(t)]\begin{bmatrix}\gamma_1(t) & \gamma_2(t)\\ 0 & \gamma_3(t)\end{bmatrix} \tag{13}$$



$\underline{p}_2(t)$
$\underline{t}_2(t)$
$\underline{x}_{h,2}(t)$
$\gamma_2(t)\underline{t}_1(t)$
$\underline{x}_{h,1}(t) = \underline{t}_1(t)\gamma_1(t)$
$\gamma_3(t)\underline{t}_2(t)$
$\underline{t}_1(t)$
$\underline{p}_1(t)$

figure 1b Geometric interpretation: of constructing t1 and t2

Now examining figure 1b it can be seen that by $\gamma_1(t)$ and $\gamma_2(t)$ will increase as $\underline{p}_1(t)$ increases and $\gamma_3(t)$ will decrease as $\underline{p}_2(t)$ is decreasing.
Eq. (13) can in fact be rewritten in terms of the transition (fundamental) matrix defined in section 2.1.

$$\begin{bmatrix}\Phi_{x,11}(t) & \Phi_{x,12}(t)\\ \Phi_{x,21}(t) & \Phi_{x,22}(t)\end{bmatrix} = T(t)\begin{bmatrix}\Phi_{y,11}(t) & \Phi_{y,12}(t)\\ 0 & \Phi_{y,22}(t)\end{bmatrix} \tag{14}$$

Eq. (14) and also Eq. (13) shows that the transition matrix $\Phi_x(t)$ can be decomposed into a matrix T(t) which represents the direction of the solutions and another transition matrix $\Phi_y(t)$ which is (block) upper triangular and has the requested decoupling behavior of the increasing and decreasing modes. The transition matrix $\Phi_y(t)$ is descended from Eq. (1) on which the state variables x(t) where transformed into a set of other variables y(t) by the transformation matrix function T(t).
Because of the decoupling behavior of T(t), the variables of y(t) can be partitioned into two parts $y_1(t)$ (ϵR^{kx1}) and $y_2(t)$ ($\epsilon R^{(n-k)x1}$). Representing respectively the increasing and the decreasing solutions.

$$\frac{d}{dt}\begin{bmatrix}\underline{y}_1(t)\\ \underline{y}_2(t)\end{bmatrix} = \begin{bmatrix}Z_{11} & Z_{12}\\ Z_{21} & Z_{22}\end{bmatrix}\begin{bmatrix}\underline{y}_1(t)\\ \underline{y}_2(t)\end{bmatrix} + \begin{bmatrix}\underline{g}_1(t)\\ \underline{g}_2(t)\end{bmatrix}\begin{matrix}\updownarrow k\\ \updownarrow n-k\end{matrix} \tag{15}$$
$$\underline{x}(t) = T(t)\underline{y}(t) \qquad \underline{g}(t) = T(t)Bu(t)$$

The question we are dealing with now is: What does one choose for the matrix functions T(t) and Z ? This will become clear in the next analysis.
Taking the first derivative with respect of time of the right hand side of Eq. (14) gives:

$$\frac{d}{dt}\left(T(t)\Phi_y(t)\right)=\frac{dT(t)}{dt}\Phi_y(t)+T(t)\frac{d\Phi_y(t)}{dt}=\frac{d}{dt}\Phi_x(t) \qquad (16)$$

A transition matrix function belonging to a homogeneous differential equation corresponding for example to Eq. (7) has the following property:

$$\frac{d}{dt}\Phi_x(t)=A\Phi_x(t)$$

Now applying this property to $\Phi_x(t)$ and $\Phi_y(t)$ gives to the following expressions:

$$\frac{d}{dt}\Phi_y(t)=Z\Phi_y(t) \quad \text{and} \quad \frac{d}{dt}\Phi_x(t)=AT(t)\Phi_y(t)$$

Substituting those into Eq. (16) gives an expression which results in a relation that Z and T(t) have to satisfy when one performs a transformation from Eq. (1) to Eq. (15).

$$\frac{dT(t)}{dt}=AT(t)-T(t)Z \qquad (17)$$

Eq. (17) consists of n^2 equations for $2n^2$ variables (elements of Z and T(t)), so n^2 degrees of freedom are left. With this extra freedom one can demand some extra constraints on Z and T(t).
To decompose a matrix in order to obtain a (block) upper triangular matrix function $\Phi_y(t)$, which is needed to guarantee decoupling of the decreasing and increasing modes, can actually be achieved in two ways.

1) QR-decomposition: This automatically leads to an orthogonal structure for the matrix function T(t) (in figure 1b the geometric representation of this decomposition is presented)
2) LU-decomposition: This leads to a lower triangular matrix function for the matrix function T(t)

Because to obtain a lower triangular matrix function for T(t) is much easier and requires less computational effort than an orthogonal structure for the matrix function T(t), we will go on with the lower triangular form for T(t).
Let T(t) be of the form:

$$T(t)=\begin{array}{cc} \left[\begin{matrix} I_k & 0 \\ R(t) & I_{n-k} \end{matrix}\right] & \begin{matrix} \updownarrow k \\ \updownarrow n-k \end{matrix} \\ \begin{matrix} \leftrightarrow & \leftrightarrow \\ k & n-k \end{matrix} & \end{array}$$

Now the structure of T(t) is prescribed and leaves us with k(n-k) degees of freedom left to prescribe the structure of Z. Because of the structure of $\Phi_y(t)$, our requirement on Z is: $Z_{21}=0$ (k(n-k) d.o.f.'s).
Substituting our choice of T(t) and Z into Eq. (17) gives:

$$\begin{bmatrix} 0 & 0 \\ \frac{d}{dt}R(t) & 0 \end{bmatrix}=\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}\begin{bmatrix} I & 0 \\ R(t) & I \end{bmatrix}-\begin{bmatrix} I & 0 \\ R(t) & I \end{bmatrix}\begin{bmatrix} Z_{11} & Z_{12} \\ 0 & Z_{22} \end{bmatrix}$$

In which A is partitioned in the similar way as Z

$$A=\begin{array}{cc} \left[\begin{matrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{matrix}\right] & \begin{matrix} \updownarrow k \\ \updownarrow n-k \end{matrix} \\ \begin{matrix} \leftrightarrow & \leftrightarrow \\ k & n-k \end{matrix} & \end{array}$$

This leads to the following expressions for R(t) and Z.

$$\frac{d}{dt}R(t) = A_{21} + A_{22}R(t) - R(t)A_{11} - R(t)A_{12}R(t) \qquad \text{with} \qquad R(t=a) = 0 \tag{18}$$

Notice that Eq. (18) has a Riccati equation structure (asymmetric class of Riccati equation).

$$Z(t) = \begin{bmatrix} A_{11} + A_{12}R(t) & A_{12} \\ 0 & A_{22} - R(t)A_{12} \end{bmatrix} \tag{19}$$

Eq. (15) can now be written as:

$$\frac{d}{dt}\underline{y}_1(t) = (A_{11} + A_{12}R(t))\underline{y}_1(t) + A_{12}\underline{y}_2(t) + \underline{g}_1(t) \tag{20a}$$

$$\frac{d}{dt}\underline{y}_2 = (A_{22} - R(t)A_{12})\underline{y}_2(t) + \underline{g}_2(t) \tag{20b}$$

In which:
$y_2(t)$, which will be decreasing, can be integrated in forward direction with some initial value $y_2(t=a)$ without any stability problems. $y_1(t)$,which will be increasing, can be integrated in backward direction with some end value $y_1(t=b)$ without any stability problems.
One major drawback however is the fact that R(t) can become unbounded in general. A way to handle this problem will be discussed in section 2.4.

2.3 Separated Boundary-Value problem

The Separated Boundary-Value problem is defined as:

$$\frac{d}{dt}\underline{x}(t) = A\underline{x}(t) + Bu(t) \qquad a \le t \le b \tag{21}$$

With boundary-values

$$B_a\underline{x}(t=a) + B_b\underline{x}(t=b) = \begin{bmatrix} 0 & 0 \\ 0 & B_{2a} \end{bmatrix}\begin{bmatrix} \underline{x}_1(a) \\ \underline{x}_2(a) \end{bmatrix} + \begin{bmatrix} B_{1b} & 0 \\ 0 & 0 \end{bmatrix}\begin{bmatrix} \underline{x}_1(b) \\ \underline{x}_2(b) \end{bmatrix} = \begin{bmatrix} \underline{d}_1 \\ \underline{d}_2 \end{bmatrix}\begin{matrix} \updownarrow k \\ \updownarrow n-k \end{matrix} \tag{22}$$

$$\{\, rank(B_{2a}) + rank(B_{1b}) = \dim(A) = n \,\}$$

The separated boundary-Value problem is very suited to be solved by the continuous decoupling method describes in section 2.2. The continuous decoupling method needs n-k initial values and k end values, which are prescribed by the Separated Boundary-Value problem as:

$$\underline{x}(a) = T(a)\underline{y}(a)$$
$$\underline{x}(b) = T(b)\underline{y}(b)$$

Substituting above equations in Eq. (22) gives:

$$\begin{bmatrix} 0 & 0 \\ B_{2a}R(a) = 0 & B_{2a} \end{bmatrix}\begin{bmatrix} \underline{y}_1(a) \\ \underline{y}_2(a) \end{bmatrix} + \begin{bmatrix} B_{1b} & 0 \\ 0 & 0 \end{bmatrix}\begin{bmatrix} \underline{y}_1(b) \\ \underline{y}_2(b) \end{bmatrix} = \begin{bmatrix} \underline{d}_1 \\ \underline{d}_2 \end{bmatrix}\begin{matrix} \updownarrow k \\ \updownarrow n-k \end{matrix}$$

Which leads to:

$$\underline{y}_1(b) = B_{1b}^{-1}\underline{d}_1$$
$$\underline{y}_2(a) = B_{2a}^{-1}\underline{d}_2$$

With above initial and end values y(t) can be obtained by 20a,b and x(t) is then obtained by

$$\underline{x}(t) = T(t)\underline{y}(t) \tag{23}$$

2.4 Integration restart

As already mentioned in section 2.2, the solution of the non-linear matrix differential equation R(t) is not bounded in general. However in general R(t) grows less catastrophic than the IVP of Eq. (1) would. A way to deal with this problem is interrupting the integration at some time t_i when $||R^i(t)||$ has reached some prescribed value. Then one can create a new basis for the solution subspace of R(t) and integration can proceed, until $||R^i(t)||$ may becomes unbounded again. The sequence can be repeated for a set of subintervals $[t_i\ t_{i+1}]$ (i=0, . . .,m-1) with a set of restart points $\{t_i\}_{i=0}^m$,with $t_a=t_0<t_1< \ldots <t_m=t_b$ until t_b has been reached. The result will be m-1 solutions $x^i(t)$ with respect to m-1 different bases. In order to obtain the solution x(t) with respect to the original basis one transforms all the solutions $x^i(t)$, with their bases, back to the original basis.

1) Obtaining a new basis for $R^{i+1}(t)$:
The new basis is obtained by a transformation of the state variables of Eq. (1) by an orthogonal transformation matrix Q^i, so we get:

$$A^i = \left(Q^i\right)^T AQ^i \tag{24a}$$
$$B^i u(t) = \left(Q^i\right)^T Bu(t) \qquad t\in[t_i \quad t_{i+1}] \qquad (i = 0,\ldots\ldots, m-1) \tag{24b}$$

The initial basis is defined as $Q^0=I_n$. The new transformation matrix Q^{i+1} is obtained by a QR-decomposition of $[-R^i(t_i)\ I_{n-k}]$. According to Appendix A this will give:

$$[-R^i(t_{i+1}) \quad I_{n-k}]U^{i+1} = [\underset{\overset{\leftrightarrow}{k}}{0} \quad \underset{\overset{\leftrightarrow}{n-k}}{V_{22}^{i+1}}] \qquad (V_{22}^{i+1}=\text{non-singular}) \tag{25}$$

Later on it will become clear why $[-R^i(t_i)\ I_{n-k}]$ is a handy choice for decomposition to obtain U^{i+1}.
The obtained orthogonal matrix U^{i+1} ($\in R^{nxn}$) will be used to create the new transformation matrix Q^{i+1}:

$$Q^{i+1} = Q^i U^{i+1} \tag{26}$$

2) Obtaining initial conditions needed to obtain solutions $R^i(t)$ and $y_2^i(t)$ for i=0, . . .,m-1
To proceed the integration after a new basis has been introduced, one needs boundary conditions that are compatible with the new basis. With other words, when one at the end transforms all the solutions $x^i(t)$, belonging to bases i, back to the original basis the different time segments $[t_i\ t_{i+1}]$ (i=0, . . .,m-1) of the solution x(t) will be connected properly and the resulting solution x(t) will be smooth.
Using Eq. (23) the following relation between $x^i(t_i)$ and $y_2^i(t_i)$ can be obtained:

$$\underline{y}_2^i(t_i) = \left[-R^i(t_i) \quad I_{n-k}\right]\underline{x}^i(t_i) \tag{27}$$

The relation between the state variables x(t) (which is the solution of Eq. (21)) and the state variables $x^i(t)$ (which are in fact also solutions of Eq. (21) but with respect to different bases) is:

$$\underline{x}(t) = Q^i \underline{x}^i(t) \qquad t \in [t_i \quad t_{i+1}] \qquad (i = 0,\dots\dots, m-1) \tag{28a}$$

The relation between state variables $x^i(t)$ belonging to the i th basis and the state variables $x^{i+1}(t)$ belonging to the next basis can be obtained by:

$$\underline{x}^i(t) = U^{i+1}\underline{x}^{i+1}(t) \qquad t \in [t_i \quad t_{i+1}] \qquad (i = 0,\dots\dots, m-1) \tag{28b}$$

Substituting Eq. (28b) into Eq. (27) gives:

$$\underline{y}_2^i(t_i) = \left[-R^i(t_i) \quad I_{n-k}\right] U^{i+1}\underline{x}^{i+1}(t_i)$$

Using decomposition Eq. (25) and $x^{i+1}(t_i) = T^{i+1}(t_i)y(t_i)^{i+1}$ (Eq. 23) leads to:

$$\underline{y}_2^i(t_i) = V_{22}^{i+1}R^i(t_i)\underline{y}_1^{i+1}(t_i) + V_{22}^{i+1}\underline{y}_2^{i+1}(t_i) \tag{29}$$

Eq. (29) Shows that a convenient choice for the initial condition to integrate $d/dR^i(t)$ Eq. (18) would be $R^i(t_i)=0$. By doing so we obtain the following recursive relation for the initial condition used to integrate $d/dy_2^i(t)$ Eq. (20b).

$$\underline{y}_2^{i+1}(t_i) = \left(V_{22}^{i+1}\right)^{-1}\underline{y}_2^i(t_i) \tag{30}$$

3) Obtaining end condition(s) needed to obtain solution(s) y_2^i (t) for i=0,. . .,m-1

Evaluating Eq. (28a) for i=m and substituting this relation into Eq. (22) we can obtain the following relation for the end value $x_1(b)$ and the end value belonging to the last basis $x_1^m(b)$.

$$\begin{bmatrix} B_{1b} & 0 \\ 0 & 0 \end{bmatrix}\begin{bmatrix} \underline{x}_1(b) \\ \underline{x}_2(b) \end{bmatrix} = \begin{bmatrix} B_{1b} & 0 \\ 0 & 0 \end{bmatrix}\begin{bmatrix} Q_{11}^m & Q_{12}^m \\ Q_{21}^m & Q_{22}^m \end{bmatrix}\begin{bmatrix} \underline{x}_1^m(b) \\ \underline{x}_2^m(b) \end{bmatrix}$$

$$\underline{x}_1^m(b) = \left(Q_{11}^m\right)^{-1}\left[\underline{x}_1(b) - Q_{12}^m\underline{x}_2^m(b)\right] \tag{31}$$

Notice that because one has chosen $R^i(t_i)=0$, $T^i(t_i) = I_n$ so Eq. (23) leads to the following property:

$$\underline{x}_1^i(t) = \underline{y}_1^i(t) \qquad t \in [t_i \quad t_{i+1}]$$
$$\underline{x}_2^i(t_i) = \underline{y}_2^i(t_i)$$

Now we can rewrite Eq. (31) as:

$$\underline{y}_1^m(b) = \left(Q_{11}^m\right)^{-1}\left[\underline{y}_1(b) - Q_{12}^m\underline{y}_2^m(b)\right] \tag{32}$$

In which $y_1(b)$ is known from the separated boundary condition ($y_1(b)=B_{1b}^{-1}d_1$) and $y_2^m(b)$ is computed in the forward integration sequence.

To transform the begin condition $y_1^m(b)$ $(y_1^i(t_i))$ into an end condition suited to start the backward integration sequence for the previous basis $y_1^{m-1}(b)$ $(y_1^{i-1}(t_i))$ we establish, by using Eq. (28b), the following recursive relation for the end condition needed to integrate $d/dy_1^i(t)$ Eq. (20a).

$$\underline{y}_1^{i-1}(t_i) = \begin{bmatrix} U_{11}^i & U_{12}^i \end{bmatrix} \begin{bmatrix} \underline{y}_1^i(t_i) \\ \underline{y}_2^i(t_i) \end{bmatrix} \tag{33}$$

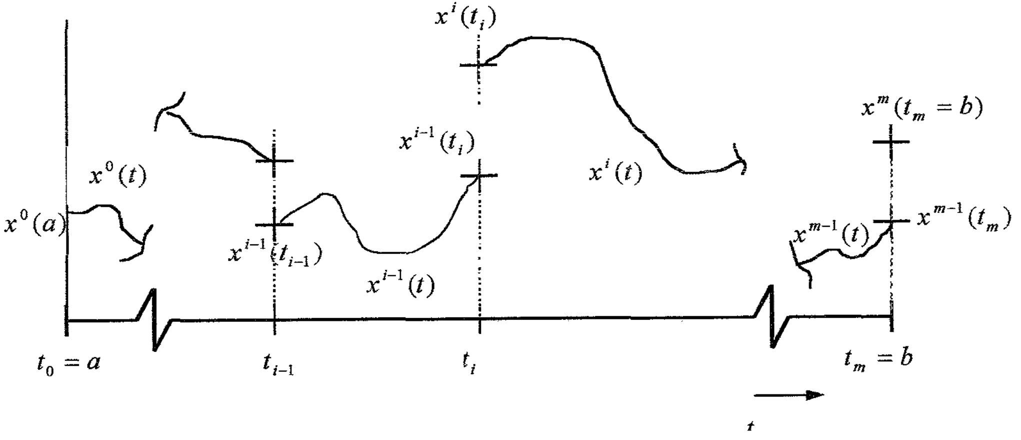


figure 2 Graphical representation of the solutions $x^i(t)$ belonging to the different bases

4) ***Transforming $y^i(t)$ into $x(t)$ for $i=0,\ldots,m-1$***

Summarizing the previous subsections 1,2 and 3 we can now obtain the initial-values (Eq. (30)) and end-values (Eq. (32) and Eq. (33)) necessary to obtain the solutions $\{R^i(t), y_2^i(t)\}_{i=0}^{i=m-1}$ with $t\in[t_i\ t_{i+1}]$ $(i=0,\ldots,m-1)$ and $\{y_2^i(t)\}_{i=m-1}^{i=0}$ with $t\in[t_i\ t_{i+1}]$ $(i=m-1,\ldots,0)$ respectively.
Using $R^i(t)$, $y_1^i(t)$, $y_2^i(t)$ and Eq. (23) we obtain $x^i(t)$:

$$\underline{x}^i(t) = \begin{bmatrix} I_k & 0 \\ R^i(t) & I_{n-k} \end{bmatrix} \begin{bmatrix} \underline{y}_1^i(t) \\ \underline{y}_2^i(t) \end{bmatrix} \qquad t\in[t_i \quad t_{i+1}] \qquad (i = 0,\ldots\ldots, m-1) \tag{34}$$

As mentioned before $x^i(t)$ can be transformed into $x(t)$ by:

$$\underline{x}(t) = Q^i \underline{x}^i(t) \qquad t\in[t_i \quad t_{i+1}] \qquad (i = 0,\ldots\ldots, m-1) \tag{35}$$

2.5 Non-Separated Boundary-Value Problem

A non-separated boundary-value problem is a general boundary value problem as defined by Eq. (1) with its boundary-value conditions. A non-separated boundary can not be partitioned in such a way that one can establish a boundary condition as is defined by Eq. (22) in section 2.3. So the continuous decoupling theory can not be used directly as it can be used by the separated boundary-value problem described in section 2.3. Yet a numerical solution for the non-separated boundary-value problem can be obtained with the continuous decoupling theory, with its favorable property, by applying the superposition theory of linear differential equations. The theory will basically be applied in the same way as it is used in the shooting method explained in section 2.1. The subtle difference is found in the way the particular solution Eq. (5) and the homogenous solution Eq. (6) will be solved. The particular solution and the homogenous solution will be solved by using the separated boundary-value problem of section 2.3.

- ***Particular solution:*** $\underline{x}_p(t)$

$$\frac{d}{dt}\underline{x}_p(t) = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} \underline{x}_{1,p}(t) \\ \underline{x}_{2,p}(t) \end{bmatrix} + \begin{bmatrix} B_1 \\ B_2 \end{bmatrix} u(t) \begin{matrix} \updownarrow k \\ \updownarrow n-k \end{matrix} \qquad a \le t \le b \tag{36}$$

With boundary-values:

$$\begin{bmatrix} 0 & 0 \\ 0 & I_{n-k} \end{bmatrix} \begin{bmatrix} \underline{x}_{1,p}(a) \\ \underline{x}_{2,p}(a) \end{bmatrix} + \begin{bmatrix} I_k & 0 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} \underline{x}_{1,p}(b) \\ \underline{x}_{2,p}(b) \end{bmatrix} = \begin{bmatrix} \underline{0} \\ \underline{0} \end{bmatrix} \begin{matrix} \updownarrow k \\ \updownarrow n-k \end{matrix} \tag{37}$$

- ***Homogeneous solution:*** $\underline{x}_h(t) = \Phi_x(t)\underline{v}$

$$\frac{d}{dt}\underline{x}_{h,i}(t) = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} \underline{x}_{1,h,i}(t) \\ \underline{x}_{2,h,i}(t) \end{bmatrix} \begin{matrix} \updownarrow k \\ \updownarrow n-k \end{matrix} \qquad a \le t \le b$$

Solving n times with n arbitrary linear independent boundary-value conditions. For example:

$$\begin{bmatrix} 0 & 0 \\ 0 & I_{n-k} \end{bmatrix} \begin{bmatrix} \underline{x}_{1,h,i}(a) \\ \underline{x}_{2,h,i}(a) \end{bmatrix} + \begin{bmatrix} I_k & 0 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} \underline{x}_{1,h,i}(b) \\ \underline{x}_{2,h,i}(b) \end{bmatrix} = \begin{bmatrix} \underline{d}_{1,i} \\ \underline{d}_{2,i} \end{bmatrix} \begin{matrix} \updownarrow k \\ \updownarrow n-k \end{matrix} \text{ with} \begin{bmatrix} \begin{bmatrix} \underline{d}_{1,1} \\ \underline{d}_{2,1} \end{bmatrix}, \begin{bmatrix} \underline{d}_{1,i} \\ \underline{d}_{2,i} \end{bmatrix}, \ldots\ldots, \begin{bmatrix} \underline{d}_{1,n} \\ \underline{d}_{2,n} \end{bmatrix} \end{bmatrix} = I_n$$

Consequence for the transition (fundamental) matrix:

$$[\Phi_{x,21}(a) \quad \Phi_{x,22}(a)] = [0 \quad I_{n-k}] \qquad \text{and} \qquad [\Phi_{x,11}(b) \quad \Phi_{x,12}(b)] = [I_k \quad 0]$$

To obtain the superposition vector one can use relation Eq. (8) from section 2.1.

Using this superposition method one will not face the problem of numerical dependency of vectors $\underline{x}_{h,i}(t)$ (or $\underline{x}_{h,1}(t)$ and $\underline{x}_{h,2}(t)$) as mentioned in section 2.2, because they will stay bounded as t becomes large.

3 Computational aspects

Computations are being performed in MATLAB. For the written m-files see Appendix C.
Integration in order to obtain the solutions $\{R^i(t),y_2^i(t)\}_{i=0}^{i=m-1}$ with $t\epsilon[t_i\ t_{i+1}]$ (i=0, . . .,m-1) is carried out by one of the variable step size integration schemes, which are present in Matlab (ode-solvers). The choice of what integration scheme one should use is optional and can be chosen by the user. In ILC applications the input signal u(t) (see Eq. (1)) consists of a discrete signal corresponding to a equidistant discrete time interval [a b]. In order use the input equidistant signal u(t) in the variable step size integration schemes interpolation of the input signal $g_1^i(t)$ is unavoidable.
Integration in order to obtain the solution(s) $\{y_2^i(t)\}_{i=m-1}^{i=0}$ with $t\epsilon[t_i\ t_{i+1}]$ (i=m-1, . . .,0) is also carried out by a variable step size integration scheme. The stored variable $y_2^i(t)$ with its variable step size formation and $g_2^i(t)$ with its equidistantial spaced structure are just as $g_1^i(t)$ interpolated during the integration process.
In order to minimize the accuracy of the solver to be a function of the used step size an accurate interpolation method is necessary. The interpolation method is also optional and the user can chose one of the interpolation methods available in MATLAB.
To obtain the transition matrix $\Phi_x(t)$, used to obtain the homogeneous solution, is quite a time consuming job. In ILC application however, one only has to compute the transition matrix $\Phi_x(t)$ once (offline). In fact computation time between the learning iterations will basically consist of computing the particular solution.

3.1 Using the solver in MATLAB

The following m-files are written in order to solve the non-separated boundary-value problem defined by Eq. (1) and Eq. (2):

- RICLBVP.m: Is a function which computes the non-separated boundary-value problem for input variables that prescribe the non-separated boundary-value problem which are: A,B,u,t,B_a,B_b,d (see Eq. (1) and Eq. (2)). Additional parameters are
  - k: Some integer value between 1 and n-1 which specifies how the separated boundary-value problem will be partitioned (see Eq. 36). The integer value k represents the amount of unstable modes that are presented in Eq. (1). If k=0 or k=n it suggests that there are respectively no unstable or no stable modes present in Eq. (1) and using this solver will not be effective because T=I. You rather use IVP's or EVP's (see section 2.2)
  - Rmax: Some maximum value which $||R^i(t)||$ is not allowed to reach. When this value has been reached integration will be restarted (see section 2.4)
  - RelTol: Relative integration error tolerance
  - AbsTol: Absolute integration error tolerance
  - Interp_methode: With the string variable 'interp_methode' the method of interpolation can be chosen.
  - Ode: With string variable 'ode' one can select one of the integration schemes in MATLAB.
- FUNDSOL.m Is a function that computes the particular solution (defined by Eq. (36) and Eq. (37))
- PARTSOL.m Is a function that computes the transition matrix $\Phi_x(t)$ (fundamental solutions).
- SEPBVP.m Is a function that computes the separated boundary value problem including integration restarts.
- RandY2.m Is an odefile which integrates $R^i(t)$ and $y_2^i(t)$
- Y1.m Is an odefile which integrates $y_1^i(t)$

When computing the non-separated boundary-value problem the previous listed m-files can be put into the following computation scheme.

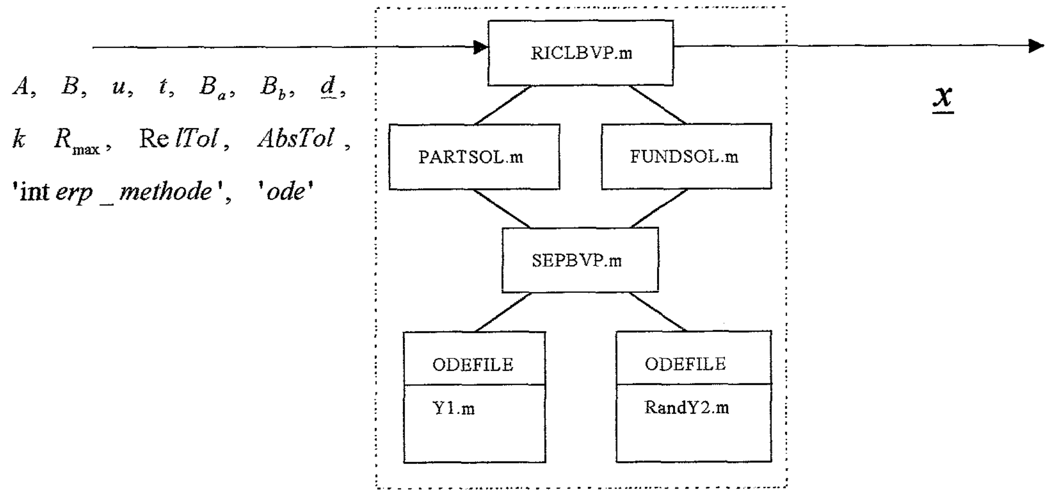

$A,\ B,\ u,\ t,\ B_a,\ B_b,\ \underline{d},$
$k\ \ R_{\max},\ \ RelTol,\ \ AbsTol,$
$'interp\_methode',\ \ 'ode'$
RICLBVP.m
PARTSOL.m
FUNDSOL.m
SEPBVP.m
ODEFILE
Y1.m
ODEFILE
RandY2.m
$\underline{x}$

figure 3 Computation scheme

One example of how to use the solver in MATLAB:

- Make sure the variables `A,B,u,t,Ba,Bb,d,k,Rmax,RelTol,AbsTol` are defined in the MATLAB workspace.
- Select an interpolation method.
  For example the "spline" method, which is a standard MATLAB interpolation method, can be selected as follows:
  Define the string variable `interp_methode` in the workspace by typing;
  `interp_methode='spline'`
- Select an integration scheme.
  For example "ode45", which is a standard MATLAB integration scheme, can be selected as follows:
  Define the string variable `ode` in the workspace by typing; `ode='ode45'`
- If al variables are defined in the workspace and the needed m-files are present in the MATLAB path just type;
  `x=riclbvp(A,B,u,t,Ba,Bb,d,k,Rmax,RelTol,AbsTol,interp_methode,ode)`
  Now x will be a vector corresponding to the given equidistant time vector t and will contain the numerical values of the approximated solution of the prescribed non-separated boundary-value problem.

4 Performance Compared to other potential suitable solvers

The potential suitable solvers to which the numerical routine, described in previous chapters, (from now on we call this numerical routine RICLBVP-solver (RICLBVP stands for RICcati Linear Boundary-Value Problem)) is compared, are:.

- A routine from the *NAG foundation* (DO2JBF) [5] (program: FORTRAN)
  The DO2JBF-routine uses a least squares method to approximate the solution by Chebyshev polynomials.
- A routine from *Institut National Polytechnique De Toulouse* (WASP) [6] (program: MATLAB)
  The WASP-routine (Wavelet Adaptive Solver for boundary value Problems) uses finite difference schemes [1]. After each attempt to obtain the approximate solution the grid is adapted by a wavelet analysis. Starting with a coarse grid (grid of resolution J), a new refined grid (of resolution J+1) is computed by adding points whenever high wavelet coefficients are detected. We refer to [7] for the details.

4.1 Achievable accuracy

Assume we have the following unstable 2nd-order differential equation:

$$\frac{d^2}{dt^2}x(t)+ax(t)=u(t) \quad \text{with } a<0 \quad \text{or}$$

$$\frac{d}{dt}\underline{x}(t)=A\underline{x}(t)+Bu(t) \quad \text{with } A=\begin{bmatrix}0 & 1\\ -a & 0\end{bmatrix} \text{ and } B=\begin{bmatrix}0\\ 1\end{bmatrix} \tag{38}$$

With non-separated boundary conditions:

$$\begin{bmatrix}1 & 0\\ 0 & 0\end{bmatrix}\underline{x}(t=0)+\begin{bmatrix}0 & 0\\ 1 & 0\end{bmatrix}\underline{x}(t=1)=\begin{bmatrix}0\\ 0\end{bmatrix}$$

Eq. (38) is approximated by the three solvers and compared to the analytical solution. Results are shown in figure 4 and table 1

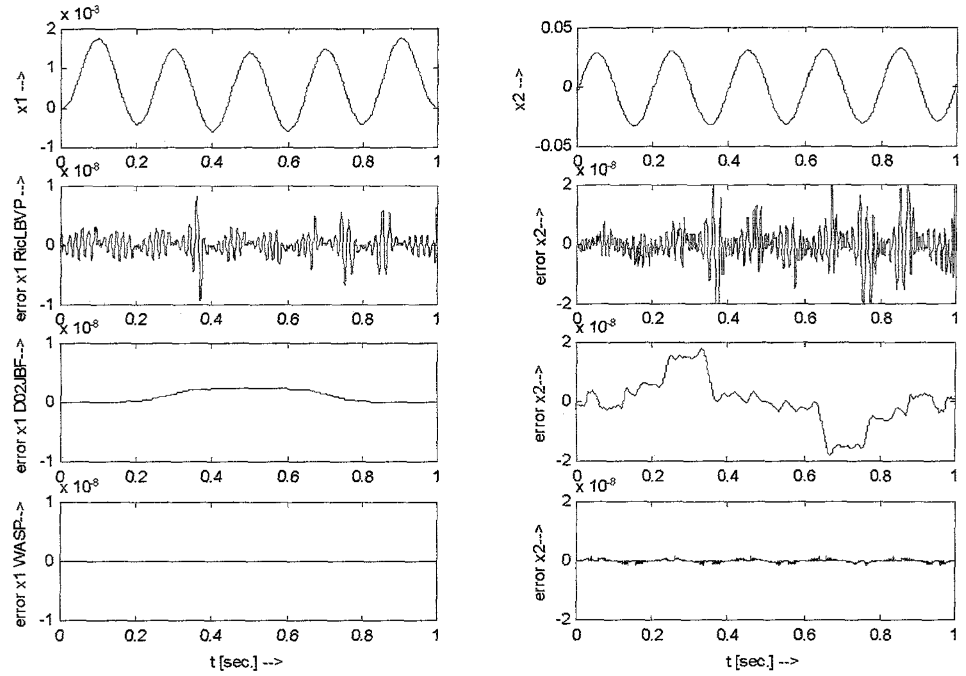


Figure 4 Result of evaluation of Eq. (38) by RICLBV-solver, DO2JBF and WASP :a=-10; u(t)=$A_m$cos(2*π*f_0*t) (discrete); A_m=1; f_0=5Hz; dt=0.001 sec.. Because the error x_1 of the WASP-solver is presented in the same scale as the rest of the figures it seems to be zero, however numerically it is not zero.

Input variables	**RICLBVP-solver**	**DO2JBF**	**WASP**
Integration scheme	Ode45	-	
Interpolation method	Spline	-	
Integration tolerance RelTol and AbsTol	1e-9		
Order of Chebyshev polynomials	-	144	
Number of collocation points	-	1000 (maximum)	
$\|\|R\|\|$	10 (Amount of integration restarts 0x)	-	
Coarseness of grid	-	-	dt
Numerical value boundary condition			
$x_1(t=0)$	0	2.168e-019	8.470e-022
$x_1(t=1)$	0	2.168e-019	1.082e-019
RMS-value of error	4.194e-009	4.641e-009	2.266e-010
Computation time	286 sec.	33 sec.	202 sec.

Table 1 Used parameters and numerical results

Our RICLBVP-solver solves the problem with an error of approximately the same order as the commanded tolerance. The accuracy of the WASP routine depends on the coarseness of the commanded grid. Using the same coarseness for the grid as the resolution of the discrete input signal u(t) we reach a higher accuracy as the other two solvers. However to obtain a higher accuracy for these two solvers one could further decreasing the integration accuracy of the RICLBVP-solver and increasing the Chebyshev polynomial order of the D02JBF routine. See figure 5 and table 2 for the obtained results.

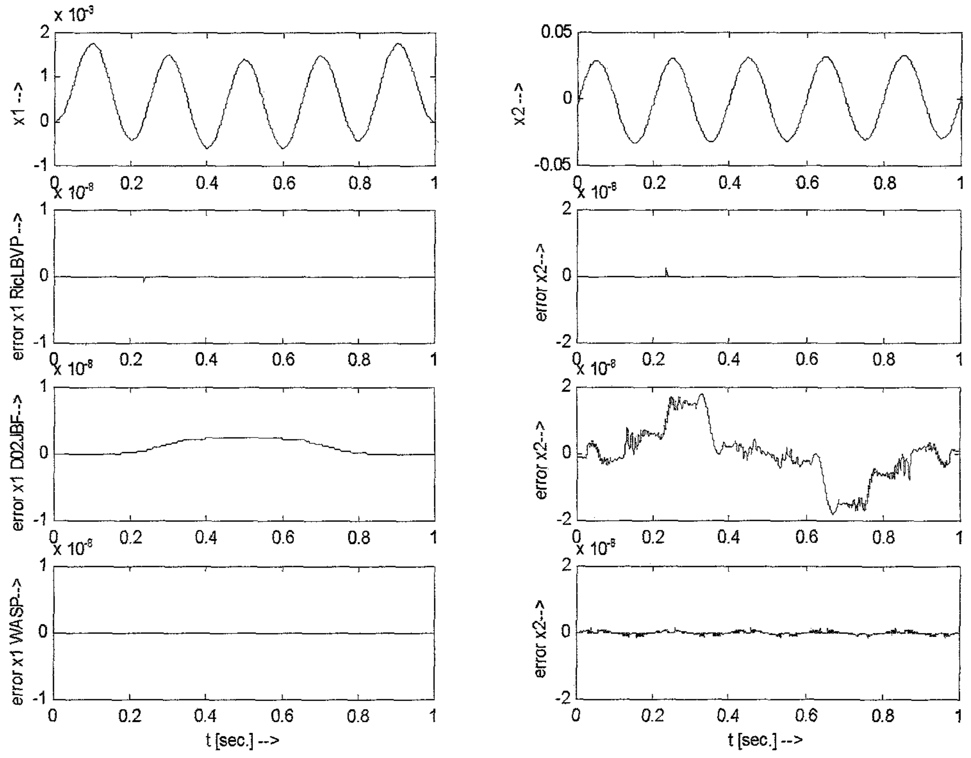


Figure 5 Result of evaluation of Eq. (38) by RICLBV-solver, DO2JBF and WASP: a=-10; $u(t)=A_m\cos(2*\pi*f_0*t)$; $A_m=1$; $f_0=5$Hz; dt=0.001 sec.; Tol=1e-12

Input variables	**RICLBVP-solver**	**DO2JBF**	**WASP**
Integration scheme	Ode45	-	
Interpolation method	Spline	-	Spline
Integration tolerance RelTol and AbsTol	1e-12		
Order of Chebyshev polynomials	-	844	
Number of collocation points	-	1000	
$\|\|R\|\|$	2 (amount of integration restarts 1x)	-	
Coarseness of grid	-	-	dt
Numerical value boundary condition			
$x_1(t=0)$	0	2.168e-019	8.470e-022
$x_1(t=1)$	0	2.168e-019	1.082e-019
RMS-value of error	5.449e-011	4.658e-009	2.266e-010
Computation time	1292 sec.	267 sec.	202 sec.

Table 2

The error of the RICLBV-solver again has almost reduced till the same value prescribed by the integration tolerance. The little deviation is mainly caused by interpolation errors and numerically errors introduced by solving the linear algebraic equations needed to obtain the superposition vector and initial and end-values needed for the integration restarts.
Notice that increasing the Chebyshev polynomial order of routine D02JBF does not introduce an improvement of the accuracy. Apparently no further improvement of the accuracy can be obtained. To further improve accuracy of the WASP routine one can refine the grid.

4.1 Stiff systems

Assume we have an unstable relatively stiff 3rd-order system with the following zero-pole-gain representation:
Zeros: none
Poles : 0.1, -0.5, 10000
Gain: 1
Input: $u(t)=A_m\cos(2*\pi*f_0*t)$; $A_m=1, f_0=3\text{Hz}$
With boundary conditions:

$$\begin{bmatrix}1&0&0\\0&0&0\\0&0&0\end{bmatrix}\underline{x}(t=0)+\begin{bmatrix}0&0&0\\0&1&0\\0&0&1\end{bmatrix}\underline{x}(t=0.6)=\begin{bmatrix}0\\0\\0\end{bmatrix}$$

The three solvers are again used to approximate the solution of the problem. Results are shown in figure 6 and table 3

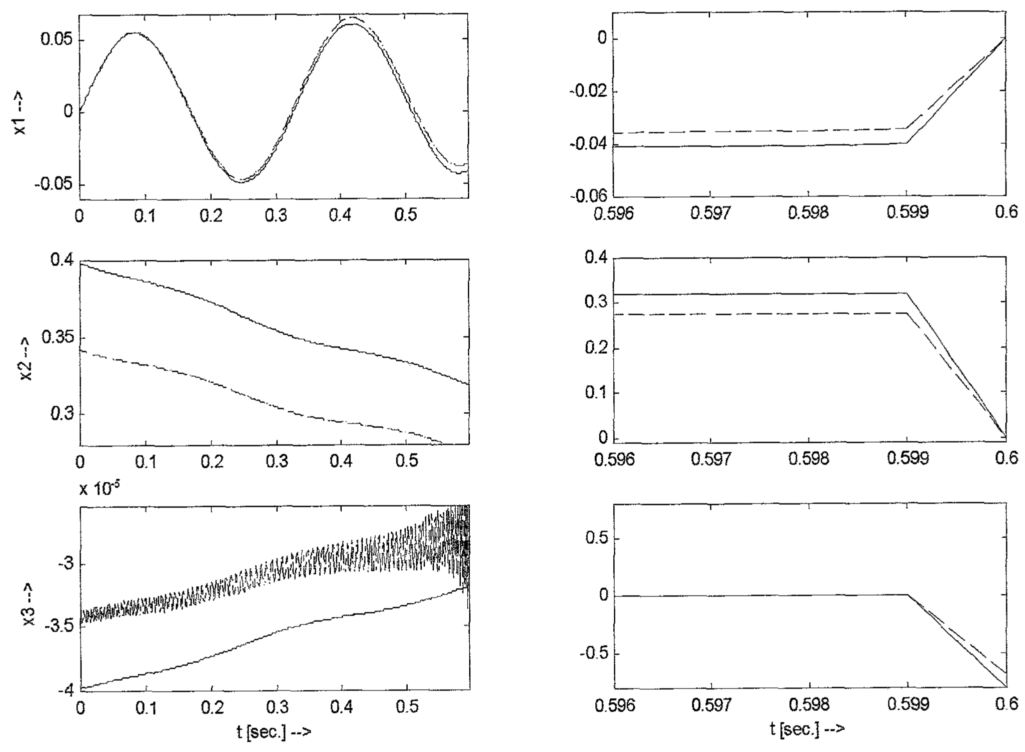


Figure 6 Approximations by RICLBV-solver [____], DO2JBF [_ _ _] and WASP [.......]. Notice that the set of figures at the right represent the end part of the solution. This part is being zoomed in with respect to the time axis in order to represent the stiffness of the solution.

Input variables	**RICLBVP-solver**	**DO2JBF**	**WASP**
Integration scheme	Ode15s	-	
Interpolation method	Spline	-	Spline
Integration tolerance RelTol and AbsTol	1e-9		
Order of Chebyshev polynomials	-	240	
Number of collocation points	-	1000	
||R||	10000 (amount of integration restarts 0x)	-	
Coarseness of grid	-	-	Adaptable
Numerical value boundary condition			
$x_1(t=0)$	0	2.385e-016	0
$x_2(t=1)$	0	-1.778e-016	0
$x_3(t=1)$	0	7.216e-016	0
Computation time	756 sec.	216 sec.	1143 sec. (7x grid adaptation)

Table 3

Notice that routine D02JBF fails. The Wasp routine does also seem to fail after computing the first solution with the initial grid (see Appendix B). Due to the efficient adaptation of the grid by a wavelet analysis it succeeds to obtain a meaningful solution. The grid is refined on the locations where the solution profile is stiff and the integration scheme has trouble getting a meaningful solution. The adaptation does consume some more computation time. It is also uncertain how much adaptation cycles there will be necessary to obtain a particular accuracy.

5 Conclusions

5.1 Stiffness

The WASP routine and the RICLBVP-solver can both deal with relative stiff unstable systems. Limitation of the RICLBVP-solver concerning the stiffness of a problem depends on the integration scheme that is used. But in spite of the relative stiff equations that one has to tackle, the actual equations that have to be integrated ($R(t), y_2(t), y_1(t)$) are relatively tame as stiffness is concerned. However because of the adaptive character of the WASP routine it can also solve very stiff problems. If grid refinement by the wavelet analysis of the WASP routine will in general lead to convergence it can handle unlimited stiff systems.

5.2 Accuracy

The disadvantage of the WASP routine is the uncertainty of the amount of grid adaptations that will lead to a specific accuracy.
The RICLBVP-solver does not have this disadvantage and will approximate the solution by a given accuracy, if a proper integration scheme is used and an accurate interpolation method is used to minimize the interpolation errors.

5.3 Computation time

The computing times which are given in tables 1,2,and 3 for the RICLBVP-solver are the total computation times. When using the solver in ILC-application only the computation time of the particular solution has to be taken into account (see chapter 3). This property makes the RICLBVP-solver in spite of its "expensive" computations a relatively fast solver compared to the WASP routine.

6 Further investigation on the RICLBVP-solver

- Further investigation could be preformed on the possibility to make the RICLBVP-solver more efficient to reduce computing time. Especially when one want to compute higher order problems the computing times are still relatively long. (Particularly when compared to ZPETC)
- Solvers designed especially to solve the matrix Riccati differential Eq. (18) could be implemented in stead of the standard ode-solvers from MATLAB
- Some method to prevent the interpolation of the stored variable $y_2^i(t)$ (see chapter 3),which will cancel the interpolation error, could be investigated.
  "Invariant imbedding" [3] appeared to be a solution for this problem but it made the solver less suitable for stiff problems. The exact reason why invariant imbedding creates this unwanted complication is not fully understood. More attention to this matter might lead to some advance.
- More accurate linear algebraic solvers to reduce errors introduced by solving the equations needed to obtain the superposition vector and initial and end-values needed for the integration restarts.

References

[1] Uri M. Ascher , Robert M.M. Mattheij, Robert D.Russell
Numerical Solution of Boundary Value Problems for Ordinary Differential Equations

[2] P.M. van Loon
Continuous decoupling transformations for linear boundary value problems
CWI Tract 52, Centrum for Mathematics and Computer Science

[3] Melvin R. Scott
Invariant Imbedding and its Applications to Ordinary Differential Equations

[4] Maurice Schneiders
Evaluation of (unstable) non-causal systems applied to iterative learning control
Eindhoven Universiteit of Technology, Faculty of Mechanical Engineering, Systems and Control Group, trainee report 2001.08

[5] NAG Fortran library Mark 18

[6] http://www.enseeiht.fr/lima/apo/wasp/
WASP: a Wavelet Adaptive Solver for boundary value Problems

[7] J.B. Caillau, J. Noailles
A *wavelet adaptive solution for boundary value problems*
Technical Report RT/APO/01/1, ENSEEIHT-IRIT, January 2001.

Appendix A QR-decomposition

For any k-dimensional subspace $S_1 \subset R^n$ there exists a column orthogonal, matrix Q_1 such that $S_1=\Re(Q_1)$. This is a result of the following:
Let $A_1 \epsilon R^{n\times k}$, with $k \leq n$. Then there exist a column orthogonal matrix $Q_1 \epsilon R^{n\times k}$ and an upper triangular matrix $R_{11} \epsilon R^{k\times k}$ such that

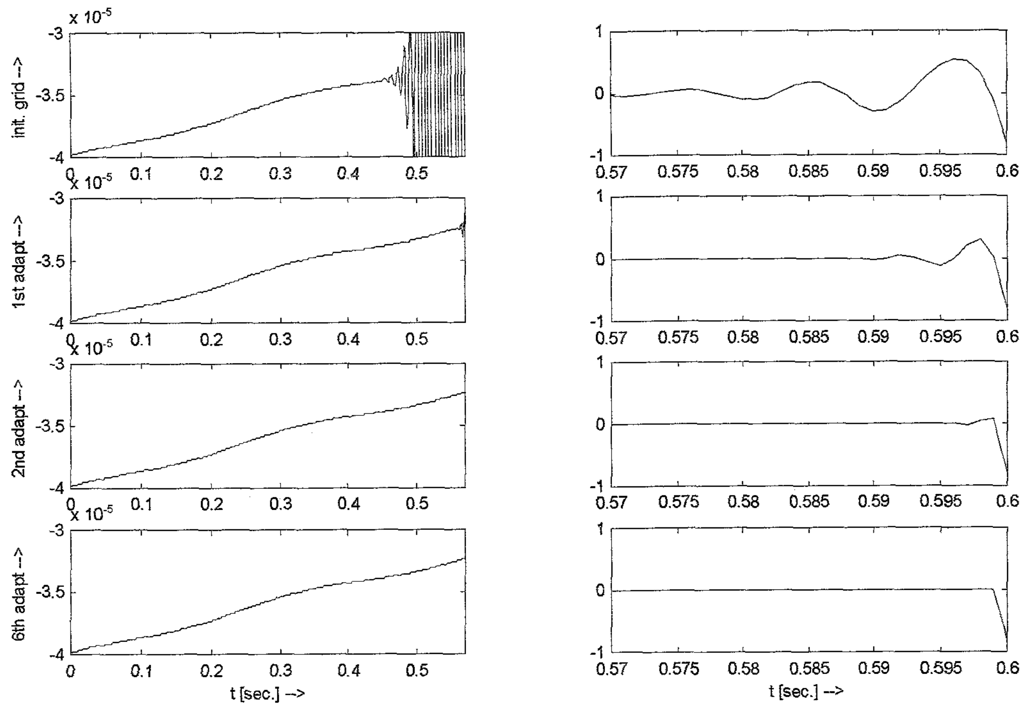
$$A_1=Q_1R_{11}$$

This is called a QR-*decomposition* of A_1.
If A_1 has full rank and we moreover require that the diagonal elements of R_{11} are positive, then Q_1 and R_{11} are uniquely determined.
If $A_1 \epsilon R^{k\times n}$, with $k<n$, then a QR-decomposition of A_1 is obtained by

$$A_1Q=[0\ R_{11}],$$

Where $Q \epsilon R^{n\times n}$ is orthogonal and $R_{11} \epsilon R^{k\times k}$ is upper triangular.

Appendix B Approximation of $x_3(t)$ after each grid adaptation



Appendix C M-files RICBVP-solver

C1 riclbvp.m

```
function
X=riclbvp(A,B,U,t,Ba,Bb,d,k,Rmax,RelTol,AbsTol,interp_methode,ode)
%X=RICLBVP(A,B,U,t,Ba,Bb,d,k,Rmax,RelTol,AbsTol,'interp_methode','ode')
% RICLBVP is a routine, which numerically computes the solution of n
% coupled first-order linear time-invariant differential equations with
% non-separated boundary-value conditions. The algorithm is especially
% suited to deal with stiff differential equations containing unstable
% modes.
%
% d/dtx(t)=A*x(t)+B*u(t)
%
% With Boundary conditions:
%
% Ba*x(t=a)+Bb*x(t=b)=d
%
%
% The routine uses fundsol.m and partsol.m, which respectively compute
% the fundamental solution (PHI(t)) and particular solution (p(t)) of a
% separated boundary-value problem (see sepbvp.m). These solutions are
% used to obtain the solution for the specified non-separated boundary-
% value problem by means of superposition.
%
% x(t)=PHI(t)*v+p(t)  in which v is the superposition vector.
%
% v=inv((Ba*PHI(t=a)+Bb*PHI(t=b))*(d-Ba(t=a)*p(t=a)-Bb(t=b)*p(t=b))
%
%
%-input variables: ------------------------------------------------------
----
%
%A        := system matrix      (nxn)
%B        := inputmatrix        (1xn) (SISO)
%U        := input
%t        := equidistant time vector
%Ba       := (nxn) matrix which specifies the linear boundary-value
%            condition
%Bb       := (nxn) matrix which specifies the linear boundary-value
%             condition
%d        := (1xn) colomvector which specifies the linear boundary-value
%            condition
%k        := some integer value between 1 and n-1 (specifies how the
%            separated boundary-value problem, calculated in sebvp.m,
%            will be partitioned)
%Rmax     := maximum absolute value of R(t) at which numerical
%            integration will be restarted (used because of the
%            unboundness of the solution of the Riccati differential
%            equation, which is used to obtain a continuous decoupling
%            of de state variables necessary to obtain a numerical
%            stable algorithm (see sepbvp.m))
%RelTol   := relative error tolerance
%AbsTol   := absolute error tolerance
%interp_methode    := with string variable 'interp_methode' the method
%                     of interpolation can be chosen see interp1.m for
%                    the different methods


%ode      := with string variable 'ode' one can select one of the
%            integretion schemes in Matlab that will be used for
%            numerical integration
%
%-output variables: ---------------------------------------------------
%
% X       := vector which contains the solutions of the non-seperated
%            boundary-value problem
%
%-used m-files: -------------------------------------------------------
%
% fundsol.m   (not standard MATLAB)
% partsol.m   (not standard MATLAB)
%
%
% Author: Bas Roset
% Date:   July 2001


n=length(A);

%--------------------------------------------------------------------%

[PHI,PHI_t0,PHI_te]=fundsol(A,t,k,Rmax,RelTol,AbsTol,interp_methode,ode
);%Fundemental solution of sep. bvp
p=partsol(A,B,U,t,k,Rmax,RelTol,AbsTol,interp_methode,ode);
%Particular solution of sep. bvp
v=(inv(Ba*PHI_t0+Bb*PHI_te))*(d-Ba*p(:,1)-Bb*p(:,end));  %Superposition
vector

X=[];
n=length(t);
for k=1:length(t),
   PHI_k=[];
   for i=0:(length(A)-1),
      phi_i=PHI(:,k+i*n);
      PHI_k=[PHI_k,phi_i];
   end,
   x=PHI_k*v+p(:,k);     %Solution for nonsep. BVP
   X=[X x];
end
```

C2 fundsol.m

```
function
[PHI,PHI_a,PHI_b]=fundsol(A,t,k,Rmax,RelTol,AbsTol,interp_methode,ode)

%[PHI,PHI_a,PHI_b] = FUNDSOL(A,t,k,Rmax,RelTol,AbsTol,'interp_methode')
% FUNDSOL evaluates the fundamental solution of n coupled first order
% linear time-invariant differential equations with separated
% boundary-value conditions.
%
% d/dtx(t)=A*x(t)      A=[A_11  A_12] (k)
%                        [A_21  A_22] (n-k)
%
%                         (k)    (n-k)
%
% With Boundary conditions:
%
% Ba*x(t=a)+Bb*x(t=b)=d
%
% Ba=[0  0 ] (k)         Bb=[Bb1 0] (k)        d=[d1] (k)
%    [0 Ba2] (n-k)          [0   0] (n-k)        [d2] (n-k)
%
%
% To obtain the fundamental matrix (PHI(t)) with n linear independent
% solutions, the differential equation is solved n times by using
% linear independent separated boundary-value conditions.
%
%-input variables: ---------------------------------------------------
%
%t        := equidistant time vector
%A        := system matrix          (nxn)
%k        := Some integer value between 1 and n-1
%Rmax     := absolute value at which numerical integration will be
%            restarted(used because of the unboundness of the solution
%            of the riccati differential equation, which is used to
%            obtain a continuous decoupling of de state variables
%            necessary to obtain a numerical stable algorithm)
%RelTol   := relative error tolerance
%AbsTol   := absolute error tolerance
%interp_methode   := with string variable interp_methode the method of
%                    interpolation can be chosen see interp1.m for the
%                    different methods
%ode      := with string variable 'ode' one can select one of the
%            integration schemes in Matlab that will be used for
%            numerical integration
%
%-output variables: -------------------------------------------------
%
% PHI   := vector nx(n*t) which contains de fundamental solutions
%           PHI= [PHI_11 PHI_12]    (k)
%                [PHI_21 PHI_22]    (n-k)
%                  (k)    (n-k)
%
% PHI_a  := fundamental matrix at t=a  [PHI_21(t=a) PHI_22(t=a)]=
%                                      [0 I_n-k]
% PHI_b  := fundamental matrix at t=b  [PHI_11(t=b) PHI_12(t=b)]=
%                                      [I_k 0]
```

```
%
%-used m-files: ---------------------------------------------------
%
% sepbvp.m  (numerical algorithm to solve separated boundary value
%            problem)
%
%
% Author: Bas Roset
% Date:   July 2001


n=length(A);

I=eye(n);                                   %n linear independent
boundary-values

%% solving the homogeneous separated boundary-value problem (n times)
PHI=[];
for i=1:n,
   d=I(:,i);                                % boundary-values
   d1=d(1:k);
   d2=d(k+1:n);

[t,phi]=sepbvp(A,zeros(length(A),1),t,zeros(1,length(t)),d2,d1,Rmax,Rel
Tol,AbsTol,interp_methode,ode);
   PHI=[PHI,phi];
end,

%% select PHI_a and PHI_b %%
PHI_a=[];
PHI_b=[];
m=length(t);
for i=0:length(A)-1,
   phi_a=PHI(:,i*m+1);
   phi_b=PHI(:,i*m+m);
   PHI_a=[PHI_a,phi_a];
   PHI_b=[PHI_b,phi_b];
end,
```

C3 partsol.m

```
function p=partsol(A,B,U,t,k,Rmax,RelTol,AbsTol,interp_methode,ode);

%p = PARTSOL(A,B,U,t,k,Rmax,RelTol,AbsTol,'interp_methode')
% PARTSOL evaluates the particular solution of n coupled first order
% linear time invariant differential equations with separated
% boundary-value conditions.
%
% d/dtx(t)=A*x(t)+B*u(t)    A=[A_11  A_12] (k)          B=[B_1] (k)
%                             [A_21  A_22] (n-k)           [B_2] (n-k)
%
%                              (k)    (n-k)                  (1)
%
% With Boundary conditions:
%
% Ba*x(t=a)+Bb*x(t=b)=d
%
% Ba=[0  0 ] (k)         Bb=[Bb1 0] (k)        d=[d1] (k)
%    [0 Ba2] (n-k)          [0   0] (n-k)        [d2] (n-k)
%
%
% To obtain the particular solution (p(t)) the differential equation is
% solved by using the following boundary-value conditions: p2(t=a)=0
% and p1(t=b)=0
%
%-input variables: -------------------------------------------------
%
%A        := system matrix     (nxn)
%B        := input             (1xn) (SISO)
%k        := some integer value between 1 and n-1
%t        := equidistant time vector
%U        := input
%Rmax     := absolute value at which numerical integration will be
%            restarted (used because of the unboundness of the solution
%            of the riccati differential equation, which is used to
%            obtain a continuous decoupling of de state variables
%            necessary to obtain a numerical stable algorithm)
%RelTol   := relative error tolerance
%AbsTol   := absolute error tolerance
%interp_methode   := with string variable interp_methode the method
%                    of interpolation can be chosen see interp1.m for
%                    the different methods
%ode      := with string variable 'ode' one can select one of the
%            integration schemes in Matlab that will be used for
%            numerical integration
%
%-output variables: ------------------------------------------------
%
% p       := vector which contains the particular solutions
%
%-used m-files: ----------------------------------------------------
%
% sepbvp.m  (not standard MATLAB)
%
%
% Author: Bas Roset
% Date:   July 2001
```

```
n=length(A);

d=zeros(length(A),1);                          % boundary-values
d1=d(1:k);
d2=d(k+1:n);

%% solving the inhomogeneous separated boundary-value problem %%
[t,p]=sepbvp(A,B,t,U,d2,d1,Rmax,RelTol,AbsTol,interp_methode,ode);
```

C4 sepbvp.m

```
function
[t,X]=sepbvp(A,B,t,u,d2,d1,Rmax,RelTol,AbsTol,interp_methode,ode)

%[t,X] = SEPBVP(A,B,t,u,d2,d1,Rmax,RelTol,AbsTol,'interp_methode')
% SEPBVP is an algorithm, which numerically computes the solution of n
% coupled first-order linear time-invariant differential equations with
% separated boundary-value conditions. The algorithm is especially
% suited to deal with stiff differential equations containing unstable
% modes.
%
% d/dtx(t)=A*x(t)+B*u(t)    A=[A_11  A_12] (k)          B=[B_1] (k)
%                              [A_21  A_22] (n-k)          [B_2] (n-k)
%
%                               (k)    (n-k)                (1)
%
% With Boundary conditions:
%
% Ba*x(t=a)+Bb*x(t=b)=d
%
% Ba=[0   0 ] (k)        Bb=[Bb1 0] (k)        d=[d1] (k)
%    [0 Ba2] (n-k)          [0    0] (n-k)        [d2] (n-k)
%
%     (n)                    (n)
%
% In this function Ba2 and Bb1 cannot be specified by the user
% (Ba2=I_n-k,Bb=I_k). The algorithm is based on a continuous decoupling
% transformation (x(t)=T(t)*y(t))(riccati transformation) of the state
% variables. The transformation leads to the following set of
% differential equations.
%
% d/dty1(t)=Z_11(t)*y1(t)+Z_12(t)*y2(t)+g1(t)   (k)
% d/dty2(t)=                 Z_22(t)*y2(t)+g2(t)   (n-k)
%
% With Boundary conditions:
%
% Ba*T(t=a)*y(t=a)+Bb*T(t=b)*y(t=b)=d
%
% Ba=[0   0 ] (k)        Bb=[Bb1 0] (k)        d=[d1] (k)
%    [0 Ba2] (n-k)          [0    0] (n-k)        [d2] (n-k)
%
%     (n)                    (n)
%
% Transformation matrix:
%
% T(t)=[I_k     0  ]
%      [R(t)  I_n-k]
%
% with R(t) satisfying: d/dtR(t)=A_21+A_22*R(t)-R(t)*A_11-
%                                         R(t)*A_12*R(t) with R(t=a)=0
%
% The transformation will decouple the stable decreasing modes (y2(t))
% and the unstable increasing modes (y1(t)). The obtained property
% makes it possible to integrate d/dty2(t) in forward direction
% followed by a backward integration of d/dty1(t) without leading to
% stability problems during integration. A disadvantage however, is
% that R(t) is not bounded in general. In order to prevent that R(t)
% blows up between a<t<b, integration is canceled after some
% time when ||R(t)||=Rmax is reached. After a reorthogonalization of
% the solution basis of d/dtx(t) the integration will be restarted this
% procedure is repeated until t=b is reached.
%
%
%-input variables: --------------------------------------------------
%
%A        := system matrix     (nxn)
%B        := input matrix      (1xn) (SISO)
%d1       := boundary condition at t=b
%d2       := boundary condition at t=a
%time     := equidistant time vector
%u        := input
%Rmax     := absolute value at which numerical integration will be
restarted
%RelTol   := relative error tolerance
%AbsTol   := absolute error tolerance
%interp_methode    := with string variable interp_methode the method of
%                     interpolation can be chosen see interp1.m for the
%                     different methodes
%ode      := with string variable 'ode' one can select one of the
%            integration schemes in Matlab that will be used for
%            numerical integration
%
%-output variables: -------------------------------------------------
%
% X       := numerical solution of the separated boundary-value problem
%
%-used m-files: ------------------------------------------------------
----
%
%
% Randy2.m (not standard MATLAB (ode-file)
% y1.m     (not standard MATLAB (ode-file)
% spline.m (standard MALAB)
%
%
% Author: Bas Roset
% Date:   July 2001

n=length(A);
k=length(d1);
t0=t(1);
dt=t(2)-t(1);
te=t(end);

%----------------------------- Initialization ------------------------%
Bu=B*u;
t_Bu=t;
Qi=eye(n);                          %Q0=I
Ui=eye(n);                          %U0=I
y2_0=d2;                            %Initial value for solving
d/dty_2(t)
R_0=zeros(n-k,k);                   %Initial value for solving d/dtR(t)
i=0;
Nint=0;                             %Amount of time steps from discrete
                                    %time vector that have been
                                    %integrated
```

```
%-----------------  solving {Ri(t),y_2i(t)} i=0,...,m-1 ------------%
Y2_0=y2_0;
while t0<te,
   i=i+1;
   %% Transformation of the state variables %%
   Ai=Qi'*A*Qi;
   Bui=Qi'*Bu(:,[Nint+1:length(Bu)]);
   %-------------- Integration of d/dtR(t) and d/dty_2(t) ---------%
   R_0=mat2col2(R_0);  %Converting initial matrix R_0 into initial
                       %colomvector
   R0_y20=[R_0;y2_0];  %Initial vector
   options = odeset('Events','on','RelTol',RelTol,'AbsTol',AbsTol);
   [ti,R_y2]=feval(ode,'Randy2',[t0 te],R0_y20,options,t_Bu,Ai,Rmax,
                             Bui,k,interp_methode);%solving R and y2
   if ti(end)-ti(1)<dt,
      error('Integration time interval becomes smaller than the given
             discrete time vector (t) steps: => increase Rmax or
             decrease discrete time steps')
   end,
   R_y2=R_y2.';
   Rmat=[];
   for s=1:size(R_y2,2),   %Converting vector solution of R into matrix
      rmat=col2matmk(R_y2([1:(n-k)*k],s),n-k,k);
      Rmat=[Rmat,rmat];
   end,
   y2=R_y2([(n-k)*k+1:(n-k)*k+(n-k)],:);
   %-----------------------------------------------------------------%
   %% Interval of input (Bu) used for integration %%
   Bui=Qi'*Bu(:,[Nint+1:fix(ti(end)/dt)+1]);
   t_Bui=t(Nint+1:fix(ti(end)/dt)+1);
   %% Storing variables %%
   eval(['Rmat' num2str(i) ' = Rmat;']);
   eval(['y2' num2str(i) ' = y2;']);
   eval(['ti' num2str(i) ' = ti;']);
   eval(['Qi' num2str(i) ' = Qi;']);
   eval(['Ui' num2str(i) ' = Ui;']);
   eval(['Bui' num2str(i) ' = Bui;']);
   eval(['t_Bui' num2str(i) ' = t_Bui;']);
   %% New Qi en y2_0 %%
   [Ui,R]=qr([eye(k);Rmat(:,[end-k+1:end])]);
   y2_0=(Ui([k+1:n],[k+1:n]))'*y2(:,end);
   Qi=Qi*Ui;
   Y2_0=[Y2_0,y2_0]; % Storing initial conditions y2_0i
   %% Total amount of steps which have been integrated %%
   Nint=fix(ti(end)/dt);
   %% Discrete time vector t left to integrate %%
   t_Bu=[t(Nint+1):dt:te];
   %% New starting time for integration %%
   t0=ti(end);
end,
m=i+1;
i=m;
eval(['Ui' num2str(i) ' = Ui;']);
%-----------------------------------------------------------------------%

%--------------  solving {y_1i(t)=x_1i} i=m-1,...,0 ----------------%
Qm=Qi;
Qm11=Qm([1:k],[1:k]);
```

```
Qm12=Qm([1:k],[k+1:n]); %Transformation of end-value (d1) to the
                        %appropriate solution basis


xi1_0=inv(Qm11)*(d1-Qm12*Y2_0(:,m));
for z=1:m-1,
   %% loading variables %%
   Rmat=eval(['Rmat' num2str(i-1)]);
   y2=eval(['y2' num2str(i-1)]);
   Qi=eval(['Qi' num2str(i-1)]);
   ti=eval(['ti' num2str(i-1)]);
   t_Bui=eval(['t_Bui' num2str(i-1)]);
   Bui=eval(['Bui' num2str(i-1)]);
   Ui=eval(['Ui' num2str(i)]);
   Ai=(Qi)'*A*Qi;
   %--------------- Backward-integration of d/dty_1(t) -----------%
   Rcol=[];
   for j=1:length(ti),       %Transforming the matrix riccati solution
      rcol=mat2col2(Rmat(:,[(j-1)*k+1:j*k]));%into a colomvector
                                            %solution (nessecery
      Rcol=[Rcol,rcol];                     %to abtain compatibility
                                            %with the ode-solver)
   end,
   U11i=Ui([1:k],[1:k]);                    %Recursive relation which
                                            %computes the new
   U12i=Ui([1:k],[k+1:n]);                  %end-value used for
                                            %Backward-integration
   xi1_0=[U11i,U12i]*[xi1_0;Y2_0(:,i)];     %of d/dty_1(t)

   options = odeset('RelTol',RelTol,'AbsTol',AbsTol);
   [t2i,xi1]=feval(ode,'y1',[ti(end) ti(1)],xi1_0,options,t_Bui,ti,
                           Rcol,Ai,Bui([1:k],:),y2,interp_methode);
   xi1=flipud(xi1)';                        %x1=y1
   xi1_0=xi1(:,1);
   %-----------------------------------------------------------------
   % Interpolate x1=y1, y2 and R to the same equidistantial spaced
   % time vector
   y2_inter=[];
   for h=1:size(y2,1),
      Y2_inter=interp1(ti,y2(h,:),t_Bui,interp_methode);
      y2_inter=[y2_inter;Y2_inter];
   end,
   xi1_inter=[];
   for h=1:size(xi1,1),
      Xi1_inter=interp1(flipud(t2i),xi1(h,:),t_Bui,interp_methode);
      xi1_inter=[xi1_inter;Xi1_inter];
   end,
   %xi1_inter=xi1
   Rcol_inter=[];
   for h=1:size(Rcol,1),
      rcol_inter=interp1(ti,Rcol(h,:),t_Bui,interp_methode);
      Rcol_inter=[Rcol_inter;rcol_inter];
   end,
   %% Compute x2,  x2=[R,In-k]*[y1;y2]  (x=Ty)%%
   xi2=zeros(n-k,length(t_Bui));
   X=zeros(n,length(t_Bui));
   for j=1:length(t_Bui),
      xi2(:,j)=[col2matmk(Rcol_inter(:,j),n-k,k),
                              eye(n-k)]*[xi1_inter(:,j);y2_inter(:,j)];
      %Compute the total solution by transforming all separate
      %solutions from the different solution basis's to the original
      %basis.
      X(:,j)=Qi*[xi1_inter(:,j);xi2(:,j)];
   end,
   %% Storing the variables %%
   eval(['X' num2str(i-1) ' = X;']);
   i=i-1;
end
%--------------------------------------------------------------------%

X=[];
for i=1:m-2,
   x=eval(['X' num2str(i)]);
   X=[X,x(:,[1:end-1])];
end
x=eval(['X' num2str(m-1)]);
X=[X,x];

mess = sprintf('Amount of integration restarts : %4d',m-2);
disp(mess)
```

C5 Randy2.m

```
function varargout =
Randy2(t,Rcol,flag,t_Bu,A,Rmax,Bu,k,interp_methode)

%varargout = RandY2(t,y1,flag,t_Bu,A,Rmax,Bu,k,'interp_methode')
% RandY2 is an ODEFILE, which is used to integrate the matrix Riccati
% equation
%
% dR(t)/dt and the dy2(t)/dt in forward direction.
%
% dR(t)/dt=A_21+A_22*R(t)-R(t)*A_11-R(t)*A_12*R(t);
% dy2(t)/dt=Z_22(t)*y2(t)+g2(t);
%
% Z_22(t)=A_22-R(t)*A_12
%
%%-input variables: -----------------------------------------------------%
%
% A    :=System matrix     (nxn)
% Bu   :=Input
% Rcol :=Solution of Riccati differential equation R(t) converted
%        to colomvector form
% g1   :=Input
% y2   :=Solution obtained with forward integration (see odede
% t_Bu :=Equidistantial spaced time vector which is used to interpolate
%        g1(t_g1) to g1(t)
% ti   :=In general a non-equidistantial spaced time vector which is
%        used to interpolate Rcol(ti) and y2(ti) to Rcol(t) and y2(t)
% interp_methode   := with string variable interp_methode the method of
%                     interpolation can be chosen see interp1.m for the
%                     different methods.
%
%
%-output variable: ----------------------------------------------------
%
% R(t) (columvector)
% y2(t)
%
%-used m-files: -------------------------------------------------------
%
% interp1.m     (standard MATLAB)
% col2matmk.m   (not standard MATLAB)
% mat2col2      (not standard MATLAB)
%

switch flag
case ''                        % Return dRcol/dt = f(t,Rcol).
   varargout{1} = f(t,Rcol,t_Bu,A,Bu,k,interp_methode);
case 'events'                  % Return [value,isterminal,direction].
  [varargout{1:3}] = events(t,Rcol,Rmax,A,k);
otherwise
  error(['Unknown flag ''' flag '''.']);
end
```

```
% ---------------------------------------------------------------------
function dRcol = f(t,Rcol,t_Bu,A,Bu,k,interp_methode)
n=length(A);
A_11=A(1:k,1:k);
A_12=A(1:k,k+1:n);
A_21=A(k+1:n,1:k);
A_22=A(k+1:n,k+1:n);

Bu_t=[];
for i=1:size(Bu,1),
   bu_t=interp1(t_Bu,Bu(i,:),t,interp_methode);
   Bu_t=[Bu_t;bu_t];
end,

Rmat=col2matmk(Rcol(1:(n-k)*k),n-k,k);
dRmat=A_21+A_22*Rmat-Rmat*A_11-Rmat*A_12*Rmat;

Z_22=A_22-Rmat*A_12;
y2=Rcol((n-k)*k+1:(n-k)*k+(n-k));
g2=-Rmat*Bu_t([1:k],:)+Bu_t([k+1:n],:);
dy2=Z_22*y2+g2;
dRcol=[mat2col2(dRmat);dy2];

% ---------------------------------------------------------------------
function [value,isterminal,direction] = events(t,Rcol,Rmax,A,k)
n=length(A);
A_11=A(1:k,1:k);
A_12=A(1:k,k+1:n);
A_21=A(k+1:n,1:k);
A_22=A(k+1:n,k+1:n);

value = Rmax-abs(Rcol);

isterminal = ones(length(value),1);    % stop at local minimum
direction = zeros(length(value),1);    % [local minimum, local maximum]
% ---------------------------------------------------------------------
```

C6 y1.m

```
function dy1 = y1(t,y1,flag,t_g1,ti,Rcol,A,g1,y2,interp_methode);

%dy1 = Y1(t,y1,flag,t_g1,ti,Rcol,A,g1,y2,'interp_methode')
% Y1 is an ODEFILE, which is used to integrate dy1(t)/dt in backward
direction.
%
% dy1(t)/dt=Z_11(ti)*y1(t)+Z_12(ti)*y2(ti)+g1(t_g1);
%
%%-input variables: --------------------------------------------------
%
% A    :=System matrix      (nxn)
% Rcol :=Solution of Riccati differential equation R(t) converted
%        to colomvector vorm
% g1   :=Input
% y2   :=Solution obtained with forward integration (see odede
% t_g1 :=Equidistantial spaced time vector which is used to interpolate
%        g1(t_g1) to g1(t)
% ti   :=In general a non-equidistantial spaced time vector which is
%        used to interpolate Rcol(ti) and y2(ti) to Rcol(t) and y2(t)
%interp_methode    := with string variable interp_methode the method of
%                     interpolation can be chosen see interp1.m for the
%                     different methods.
%
%
%-output variable: ----------------------------------------------------
%
% y1(t)
%
%-used m-files: -------------------------------------------------------
%
% interp1.m     (standard MATLAB)
% col2matmk.m   (not standard MATLAB)
%

k=size(g1,1);
n=length(A);
A_11=A(1:k,1:k);
A_12=A(1:k,k+1:n);
A_21=A(k+1:n,1:k);
A_22=A(k+1:n,k+1:n);
Z_12=A_12;

g1_t=[];
for i=1:size(g1,1),
   G1_t=interp1(t_g1,g1(i,:),t,interp_methode);
   g1_t=[g1_t;G1_t];
end,

rcol=[];
for i=1:size(Rcol,1),
   rcol_i=interp1(ti,Rcol(i,:),t,interp_methode);
   rcol=[rcol;rcol_i];
end,
Rmat=col2matmk(rcol,n-k,k);
```

```
y2_t=[];
for i=1:size(y2,1),
   Y2_t=interp1(ti,y2(i,:),t,interp_methode);
   y2_t=[y2_t;Y2_t];
end,

dy1=(A_11+A_12*Rmat)*y1+Z_12*y2_t+g1_t;
```

C7 col2matmk

```
function Pmat=col2matmk(Pcol,M,K);

%Pmat = COL2MATMK(Pcol,M,K)
% COL2MATNM converts a colomvector (Nx1) into a MxK matrix as follows
%
% [c(1)]                  [c(1)           c(i)    .  c(1*K)]
% [c(i)]  with N=M*K  ==>[c((m-1)*(k+1))  .       .  c(m*K)]  (m=1,..M)
% [ .  ]                  [  .             .       .    .   ]
% [ .  ]   (i=1,..N)      [  .             .       .    .   ]
% [ .  ]                  [c((M-1)*(k+1))  .       .  c(M*K)]
% [c(N)]
%
%-input variables: ---------------------------------------------------
%
%Pcol     := Colomvector (M*Kx1)
%M        := Some positive integer-value
%K        := Some positive integer-value
%
%-output variables: --------------------------------------------------
%
%Pmat    := MxK matrix
%

%-------------------------------- check inputs ----------------------%
if size(Pcol,2)~=1
   error('Pcol has to be a colomvector')
elseif M<1 | (fix(M))/M<1 | size(M,1)~=1 | size(M,2)~=1
   error('M has to be a scalar integer-value >0')
elseif K<1 | (fix(K))/K<1 | size(K,1)~=1 | size(K,2)~=1
   error('K has to be a scalar integer-value >0')
elseif size(Pcol,1)~=M*K
   error('colomvector Pcol must have length M*K')
end


Pmat=zeros(M,K);
l=1;
for i=1:M,
   for j=1:K,
      Pmat(i,j)=Pcol(l);
      l=l+1;
   end,
end,
```

C8 mat2col2

```
function Pcol=mat2col2(Pmat)

%Pcol = MAT2COL2(Pmat)
% MAT2COL2 converts a general matrix Pmat into a colomvector Pcol as
follows
%
% [1  2  3  4]       [1 2 3 4 5 6 7 8 9 10 11 12]'
% [5  6  7  8]  ==>
% [9 10 11 12]

%-input variables: ----------------------------------------------------
----
%
%Pmat     := Matrix
%
%-output variables: ---------------------------------------------------
-----
%
%Pcol    := colomvector
%

m=size(Pmat,1);
n=size(Pmat,2);
Pcol=zeros(m*n,1);
l=1;
for i=1:m,
   for j=1:n,
      Pcol(l)=Pmat(i,j);
      l=l+1;
   end
end
```