Proceedings of the American Control Conference
San Diego, California • June 1999

REDUCTION OF THE DIMENSIONALITY OF DYNAMIC PROGRAMMING: A CASE STUDY

A. P. de Madrid, S. Dormido, F. Morilla

Dpto. de Informática y Automática, UNED. Senda del Rey, s/n
28040-Madrid (Spain)
Phone: (+34) 91-398-7160 Fax: (+34) 91-398-6697 E-mail: angel@dia.uned.es

Abstract: This paper deals with the reduction of the computational complexity of dynamic programming, the well known *curse of dimensionality*. In this paper we show how it can be exorcized using different reduction techniques. Three of the most significant ones are introduced and compared with an example. The main conclusion is that the computational load can be reduced several orders of magnitude in an easy and intuitive way.

1 Introduction

Dynamic programming (DP) is a valuable optimization tool (Bellman 1957), with well established and very easy to understand fundamentals. No advanced mathematics are needed and, the most important of all its advantages, a true optimum is always guaranteed. Its applications are many and well known (Bellman and Dreyfus 1962): scheduling, automatic control, artificial intelligence, economics, etc. The cost is an extremely high computational load, the *curse of the dimensionality*. Although for some applications dynamic programming is applied analytically, for most applications the solution has to be found numerically and then the problem of the dimension plays a very important role: the CPU time and the storage requirements can be so high that, in practice, conventional dynamic programming cannot be used numerically except for some trivial problems. For this reason techniques to reduce the computational load have been developed (Bellman and Dreyfus 1962; Larson 1968; Dormido *et al.* 1970; Larson and Korsak 1970; Korsak and Larson 1970; Cooper and Cooper 1981; Larson and Casti 1982; Luus 1990a,b; Moreno *et al.* 1992, 1994; Sniedovich 1992).

In this paper we shall show how some of these techniques work. We were motivated in the field of automatic control, where dynamic programming was used to solve nonlinear constrained control problems. In order to solve nontrivial problems, the techniques to reduce the computational burden were needed. For this reason we studied and compared some of the most significant ones (de Madrid 1995; de Madrid *et al.* 1996). This work summarizes our results.

The paper is organized as follows. Section 2 deals with the fundamentals of dynamic programming and introduces the computational load of the basic algorithms. In section 3 the main techniques to reduce the computational complexity are described. In section 4 three of the main techniques are compared with an illustrative example. Finally, section 5 draws the conclusions of this paper.

2 Dynamic programming

2.1 The basic computational procedure

Dynamic programming is based on Bellman's Principle of Optimality (Bellman 1957; Larson and Casti 1978; Bertsekas 1995). This principle states that all the portions of an optimal trajectory are, themselves, optimal trajectories.

In the following we shall consider one of the most representative DP problems. It is defined as follows:

Minimize (maximize) the separable cost (performance) function

$$J = \sum_{k=0}^{N} L(\mathbf{x}(k), \mathbf{u}(k), k) \tag{1}$$

where

$$\mathbf{x}(k+1) = g(\mathbf{x}(k), \mathbf{u}(k), k) \tag{2}$$

and subject to

$$\mathbf{x} \in X(k) \subset \Re^n, \quad \mathbf{u} \in U(\mathbf{x}(k), k) \subset \Re^m . \tag{3}$$

◊◊◊

In this problem **x** is the *state variable*, **u** is the *decision* or *control variable*, k is the *stage*, g is the *transition function* and, in general, the cost or performance function J is known as *objective function*. In this function, L represents the cost or benefit of a single stage.

If we define I, the *minimum cost function* from stage k to the end of the optimization problem

$$I(\mathbf{x}, k) = \min_{\mathbf{u}(k), \mathbf{u}(k+1), \ldots, \mathbf{u}(N)} \left\{ \sum_{j=k}^{N} L(\mathbf{x}(j), \mathbf{u}(j), j) \right\}, \tag{4}$$

using Bellman's Principle of Optimality it is possible to prove that

$$I(\mathbf{x}, k) = \min_{\mathbf{u}} \{ L(\mathbf{x}, \mathbf{u}, k) + I[g(\mathbf{x}, \mathbf{u}, k), k+1] \} \tag{5}$$

with

$$I(\mathbf{x}, N) = \min_{\mathbf{u}(N)} \{ L(\mathbf{x}, \mathbf{u}(N), N) \} \tag{6}$$

for the final stage N.

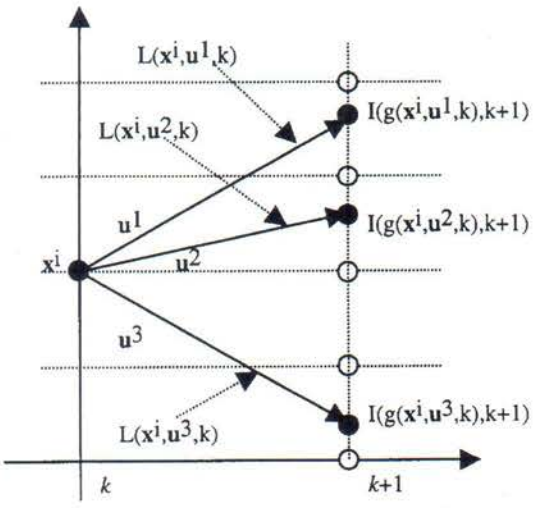

Figure 1.- Backward DP algorithm with interpolation.

American Automatic Control Council

```
initialise I(x,k)=∞, ∀x∈X(k), ∀k
evaluate I(x,N), ∀x∈X(N)
for all the stages from N-1 to 1
 for all the quantified states x^i(k)∈X(k)
  for all the admissible decisions ...
                    ... u^j(k)∈U(x^i(k),k)
   evaluate g(x^i,u^j,k)
   if g(x^i,u^j,k)∈X(k+1)
    interpolate I(g(x^i,u^j,k),k+1)
    if L(x^i,u^j,k)+I(g(x^i,u^j,k),k+1)<I(x^i,k)
     I(x^i,k)=L(x^i,u^j,k)+I(g(x^i,u^j,k),k+1)
     u*(x^i,k)=u^j
    endif
   endif
  endfor
 endfor
endfor
```

Algorithm 1.- Backward DP with interpolation.

In some cases the iterative functional equation (5) can be solved analytically. For example, in the field of automatic control the problem where the system is linear, the objective function is quadratic, the stochastic variables are Gaussian and there are no constraints is of great importance and is solved analytically making use of dynamic programming (Bertsekas 1976; Larson and Casti 1982; Pierre 1986; Mosca 1995). But for most applications (5) is solved numerically.

To solve the iterative functional equation (5) numerically the set of admissible states X and the set of admissible decisions U are quantified

$$\begin{aligned} X(k) &= \left\{\mathbf{x}^1, \mathbf{x}^2, \ldots, \mathbf{x}^{M_X(k)}\right\} \\ U(\mathbf{x}(k),k) &= \left\{\mathbf{u}^1, \mathbf{u}^2, \ldots, \mathbf{u}^{M_U(\mathbf{x}(k),k)}\right\} \end{aligned} \tag{7}$$

defining a computational grid. The computational method is described in the algorithm 1 (see Fig. 1). In this algorithm, called *backward dynamic programming* as it is solved backwardly (there exist other formulations which are solved forwardly, with similar computational load), $\mathbf{u}^*(\mathbf{x}^i,k)$ stands for the optimal decision at the state $\mathbf{x}^i$ at the stage k. The optimal decision policy is obtained for a complete family of optimization problems, i.e., for all the states at all the stages, and it always determines an absolute minimum, within the accuracy of the computational grid (Fig. 2). Another important property of this method is that the optimal initial and final states are obtained if they are not known in advance at the beginning of the optimization process.

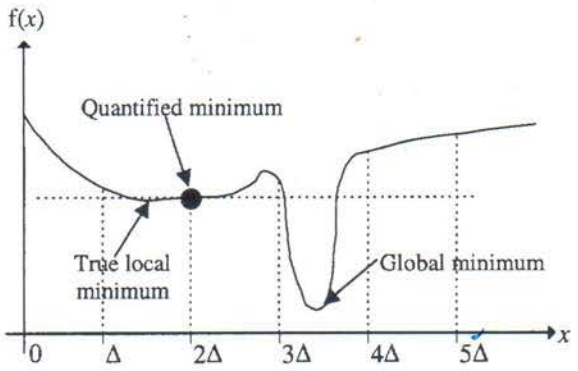

Figure 2.- A global minimum can be lost when a function is evaluated in a too coarse computational grid.

```
initialise I(x,k)=∞, ∀x∈X(k), ∀k
evaluate I(x,N), ∀x∈X(N)
for all the stages from N-1 to 1
 for all the quantified states x^i(k)∈X(k)
  for all the quantified states x^j(k+1)∈X(k+1)
   u=g^-1(x^j(k+1),x^i(k))
   if u∈U(x(k),k)
    if L(x^i,u,k)+I(x^j,k+1)<I(x^i,k)
     I(x^i,k)=L(x^i,u,k)+I(x^j,k+1)
     u*(x^i,k)=u
    endif
   endif
  endfor
 endfor
endfor
```

Algorithm 2.- Backward DP with no interpolation.

Dynamic programming shares all these properties with *exhaustive search*, also known as *direct enumeration*. Exhaustive search builds the complete decision tree whilst DP only builds some branches in a smart manner, with a computational load that can be several orders of magnitude lower (see section 2.2).

We have to take into account that if $g(\mathbf{x}^i,\mathbf{u}^j,k)$ is not a quantified state then $I(g(\mathbf{x}^i,\mathbf{u}^j,k), k+1)$ has to be interpolated. Low order interpolation polynomials are usually used. It has been proven (Guignabodet 1961, 1963) that higher order interpolation procedures do not always lead to a more accurate solution. The interpolation errors tend to increase almost linearly with $(N-k)$. For this reason, the decisions at the first stages are not so good as in the final stages and the solution is corrupted. The only way to be more accurate is the use of more quantified states and decisions, with a higher computational load.

Let us assume that the inverse function g^{-1} exists:

$$\begin{aligned} g^{-1}\left[\mathbf{x}(k+1),\mathbf{x}(k)\right] &= \mathbf{u}(k) \\ g\left[\mathbf{x}(k),g^{-1}\left[\mathbf{x}(k+1),\mathbf{x}(k)\right],k\right] &= \mathbf{x}(k+1). \end{aligned} \tag{8}$$

In this situation we can use an alternative backward dynamic programming computational procedure with no interpolation (algorithm 2 and Fig. 3). As there are no errors due to interpolation it is clear that the only way to obtain an accurate solution is a dense computational grid.

2.2 *The curse of dimensionality*

To estimate the computational complexity let us assume that the optimal initial and final states are not defined in advance and that all the possible decisions and state transitions are allowed by the constraints. Let us also assume that

$$\begin{cases} M_U(k) = M_U, & \forall \mathbf{x}, \quad \forall k \\ M_X(k) = M_X, & \forall k \end{cases} \tag{9}$$

If the grid step for each state variable is Δ_X then the number of quantified states is proportional to $(1/\Delta_X)^n$, where n is, according to (3), the dimension of the state space. In the same way the number of quantified decisions is proportional to $(1/\Delta_U)^m$, where m is the dimension of the decision space.

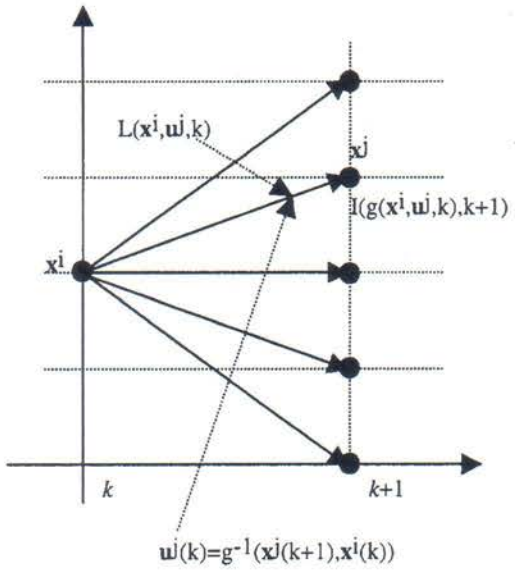

Figure 3.- Backward DP algorithm with no interpolation.

For N stages, the exhaustive search method evaluates

$$T_{ES} = M_X \cdot M_U{}^N \propto \left(\frac{1}{\Delta_X}\right)^n \cdot \left(\frac{1}{\Delta_U}\right)^{mN} \qquad (10)$$

state transitions. On the other hand, dynamic programming with interpolation requires

$$T_{DP} = N \cdot M_X \cdot M_U \propto N \cdot \left(\frac{1}{\Delta_X}\right)^n \cdot \left(\frac{1}{\Delta_U}\right)^m \qquad (11)$$

evaluations. Finally, dynamic programming with no interpolation requires

$$\hat{T}_{DP} = N \cdot M_X{}^2 \propto N \cdot \left(\frac{1}{\Delta_X}\right)^{2n}. \qquad (12)$$

state transition evaluations.

Backward dynamic programming —and all the dynamic programming numerical algorithms in general— is much more efficient than the exhaustive search method. But dense computational grids are needed and this means huge computer memories, to store the optimal policy and the intermediate computations, and long CPU time. Even though constraints do reduce the computational complexity reducing the number of quantified decisions and states, the complexity is still too high to solve most problems of practical interest. It is the so-called *curse of dimensionality*.

3 The reduction of the dimensionality

There exist many ways to reduce the computational burden of dynamic programming, each one with its own advantages and drawbacks. Some of these techniques only reduce the storage requirements (see, for example, (Larson 1968) for a description of state increment dynamic programming), whereas others do reduce the number of state transitions to evaluate and, therefore, the computational load.

We have found that many techniques can be classified in one of the following three families (de Madrid 1995; de Madrid *et al.* 1996):

- **Special partitions:** Both control and decision spaces are partitioned in such a way the computational grid that they define is a compromise between precision and dimensionality. Thus the grid can be dense where more precision is needed and coarse far from the optimal solution. In this case, an initial nominal solution (feasible and suboptimal), if available, is valuable.
- **System dynamics:** All the knowledge available about the system dynamics can be used to reduce the computational load. For example, it is possible to find in advance all the states that are not physically reachable from the initial state $\mathbf{x}(0)$; these states will not be considered by the dynamic programming algorithm as they do not belong to any optimal solution from $\mathbf{x}(0)$. Some properties of some systems, such as the convexity of the set of admissible decisions, can be also used to reduce the number of quantified states and decisions to test. In general, a deep knowledge about the system should be available before using dynamic programming to reduce the computational burden.
- **Nominal trajectories:** If a good initial nominal solution is available, not the whole regions of the state and decision spaces allowed by the constraints (3) are considered. We shall only look for the optimal solution "near" the nominal one. This procedure is repeated iteratively and the solution improves, with low cost, at each iteration.

To illustrate them we have chosen three representative techniques, one from each category:

- **Coarse partitions (*special partitions*):** A coarse partition of the state space is initially defined and a conventional dynamic programming problem is solved with it (Bellman and Dreyfus 1962; de Madrid 1995). Once the solution has been obtained, it is defined a band around it, where a new partition is defined. Then a new dynamic programming problem is solved in this band, and all the procedure is repeated until the desired grid size is reached. If the solution tends to leave the band it means that the band has to be wider.
- **Limit trajectories (*system dynamics*):** If the initial state $\mathbf{x}(0)$ is defined, it is possible to bound a region in the state space by means of the *limit trajectories*: those state space trajectories obtained applying the extreme controls to the system equations. The only states to explore are confined in the region defined by these trajectories and the constraints (Dormido *et al.* 1970). If the final state $\mathbf{x}(N)$ is defined, this region can be reduced even more (Fig. 4).
- **Successive approximations in the state space (*nominal trajectories*):** Let us consider a problem with as many state variables as control variables ($n = m$) (Bellman and Dreyfus

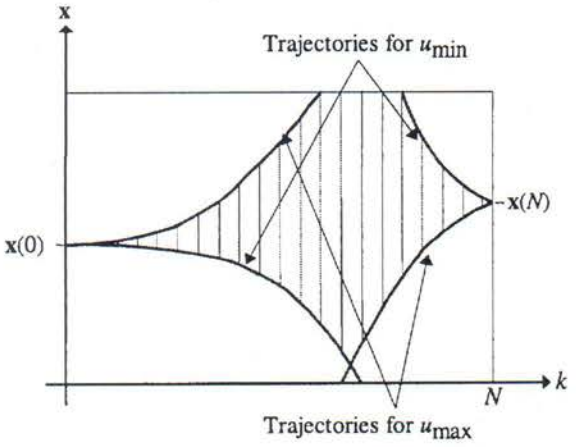

Figure 4.- The limit trajectories method.

1962; Larson 1968; Larson and Korsak 1970; Korsak and Larson 1970; Cooper and Cooper 1981; Larson and Casti 1982). Let $\{\mathbf{x}^{(0)}(k)\}$ be a nominal solution. One of the n state variables is selected, $\mathbf{x}_{i1}$, and a DP problem is solved, in such a way that $\{\mathbf{x}_i^{(0)}(k), i \neq i_1\}$ does not change. It means that there exist $(n - 1)$ equality constraints in the control variables, and therefore the problem has only one degree of freedom. The solution of this problem generates a new nominal trajectory, $\{\mathbf{x}^{(1)}(k)\}$. Then a new state variable is selected, $\mathbf{x}_{i2}$, and the method is repeated. This is done for all the n state variables. Then the method is repeated, until it converges to the same solution for all the state variables. This method reduces the complexity to linear, and can be generalized for the general case ($n \neq m$). But convergence to the true optimum —even to a local one— can be only guaranteed for few cases that are fully described in the references.

In practice, more than one method can be used at the same time to solve a problem. In this way, the advantages of each method are combined and the reduction is better. But in the following section they are used separately to illustrate how they work.

4 Comparative study

To illustrate the techniques described above, in the following we shall analyze an example from de Madrid (1995):

Minimize

$$J = \sum_{k=0}^{N-1} \left(x_1(k)^2 + 0.5x_2(k)^2 + u_1(k)^2 + u_2(k)^2 \right) \qquad (13)$$

subject to

$$\begin{pmatrix} x_1(k+1) \\ x_2(k+1) \end{pmatrix} = \begin{pmatrix} 1 & 0.6321 \\ 0 & 0.3679 \end{pmatrix} \cdot \begin{pmatrix} x_1(k) \\ x_2(k) \end{pmatrix} + \begin{pmatrix} 0.3679 & 1 \\ 0.6321 & 0 \end{pmatrix} \cdot \begin{pmatrix} u_1(k) \\ u_2(k) \end{pmatrix} \qquad (14)$$

and

$$0 \le x_1 \le 2; \quad 0 \le x_2 \le 2;$$
$$|u_1| \le 1; \quad |u_2| \le 1; \qquad (15)$$
$$\mathbf{x}(0) = \mathbf{x_0} = \begin{pmatrix} 1 \\ 1 \end{pmatrix}; \quad \mathbf{x}(T) = \mathbf{x_T} = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$$

Assume that $T = 10$ *and* $N = 10$.

◊◊◊

Equation (14) defines a linear time invariant (LTI) system with an inverse function $\mathbf{u}(k) = \mathbf{u}(\mathbf{x}(k+1), \mathbf{x}(k))$ that is defined and easy to compute. Hence we can use dynamic programming with no interpolation.

4.1 The basic computational procedure

Firstly we shall consider the resolution of the example using the basic algorithm with no interpolation. For this reason the partition of set U is not necessary. We have chosen two step sizes for the computational grid defined for X: $\Delta x_i = 0.5$ and $\Delta x_i = 0.1$, $i = 1, 2$, to illustrate the effect of the grid step on the computational load.

Table 1 summarizes the results for both grids. The number of state transitions is the theoretical maximum (12), as constraints (15) have not been considered. We can see how a dense grid improves the cost J, but the number of transitions to evaluate grows dramatically. Even though there are only two state variables and two decision variables the computational complexity of the method is very high.

Δx_i	Partitions in x_i	States per stage	Transitions according (12)	J
0.5	5	25	6,250	3.4081
0.1	21	441	1,944,810	3.1566

Table 1.- Results for the basic DP algorithm with no interpolation.

4.2 The coarse partitions method

Firstly we solve the problem with $\Delta x_i = 0.5$, which means 5,050 state transitions to evaluate (we shall only consider those transitions that are allowed by the set of constraint (15)). Secondly we define a new partition with $\Delta x_i = 0.1$ in a region around the initial solution with a width of 0.5 (constraints in the states, if exist, must be taken into account). Now a new dynamic programming problem is solved in this computational grid, with a load of 12,660 transitions to evaluate. Hence the total number of transitions to evaluate is 17,710, only 0.9% of the total number using the conventional approach (section 4.1) with $\Delta x_i = 0.1$.

This method can be iterated several times to reach the desired precision with low additional cost.

4.3 The limit trajectories method

For the system defined by (14) the limit curves are

$$\begin{cases} x_1(t) = (u_1 + u_2)t + (u_1 - 1)e^{-t} - u_1 + 2 \\ x_2(t) = u_1 + (1 - u_1)e^{-t} \end{cases} \qquad (16)$$

maximum acceleration/deceleration trajectories that verify $\mathbf{x}(0) = \mathbf{x_0}$, and

$$\begin{cases} x_1(t) = u_1 e^{T-t} - (u_1 + u_2)(T - t) - u_1 + 1 \\ x_2(t) = u_1\left(1 - e^{T-t}\right) \end{cases} \qquad (17)$$

maximum acceleration/deceleration trajectories that verify $\mathbf{x}(T) = \mathbf{x_T}$. The intersection of the state space region defined by the limit curves and the state constraints defines the set of states to test.

When a dynamic programming problem is solved with $\Delta x_i = 0.1$ the number of state transitions to evaluate is 419,694, just 21.6% of the total.

4.4 The successive approximations in the state space method

In this example the cost function is quadratic and the decision variable **u** is bounded. This is one of the situations where convergence is guaranteed. Results at the third and fourth iteration are identical so convergence is then achieved.

When we solve this problem for $\Delta x_i = 0.5$ and $\Delta x_i = 0.1$ we have to evaluate 1,000 and 26,460 transitions, respectively. This means 16.0% vs. 1.4% of their respective theoretical maximums: the dense grid obtains a much better relative cost (but, in absolute terms, the computational burden is higher).

4.5 Result analysis

Once the different methods have been illustrated we shall analyze their advantages, drawbacks and computational load. As a reference we shall consider the computational load of the conventional backward dynamic programming with no interpolation for $\Delta x_i = 0.1$.

Table 2 summarizes the results. The limit trajectories techniques achieves the worst reduction as it does not solve the problem of

Method	Transitions	%
Conventional	1,944,810	100.0
Coarse partitions	17,710	0.9
Limit trajectories	419,694	21.6
Succ. approximations	26,460	1.4

Table 2.- Comparative analysis for $\Delta x_i = 0.1$.

combinatory explosion. However, the band that it defines is a good starting point for an initial search using the other techniques.

The successive approximations method performs fine as it reduces the complexity to linear. But the applicability of the method is limited as its convergence is not guaranteed for all the problems.

Finally, the coarse partitions method achieves the best computational load. No matter the kind of problem the convergence to a true optimum is guaranteed, unless a too coarse grid led to a local optimum.

We can conclude that the coarse partitions method is a good starting point as it obtains a good initial guess that can be improved with other techniques. The nature of our problem should be analyzed in depth before a given technique is selected. In general, the better computational load reduction will be obtained combining several ones simultaneously or iteratively.

5 Conclusions

This paper has shown how the curse of dimensionality in dynamic programming can be exorcized with techniques that reduce its computational load. Three techniques have been introduced and compared with an illustrative example. The numerical results show that the load can be reduced drastically with techniques that are intuitive and easy to use.

These techniques (as many others not discussed in this paper) can be used in a wide range of problems, making possible to solve complex dynamic programming problems with an acceptable computational load.

Acknowledgements

The authors wish to acknowledge the economical support of the Spanish CICYT, under grant TAP 95-0790.

References

BELLMAN, R. E. (1957). *Dynamic programming*. Princeton University Press, New Jersey.

BELLMAN, R. E. and DREYFUS, S. E. (1962). *Applied dynamic programming*. Princeton University Press, New Jersey.

BERTSEKAS, D. P. (1976). *Dynamic programming and stochastic control*. Academic Press, New York.

BERTSEKAS, D. P. (1995). *Dynamic programming and optimal control*. Athena Scientific, Belmont, Mass.

COOPER, L. and COOPER, M. W. (1981). *Introduction to dynamic programming*. Pergamon Press, Oxford.

DE MADRID, A. P. (1995). *Aplicación de técnicas de programación dinámica a control predictivo basado en modelos*. PhD Thesis. UNED, Madrid. (In Spanish.)

DE MADRID, A. P., DORMIDO, S., MORILLA, F. and GRAU, L. (1996). *Dynamic programming predictive control*. 13th World Congress of IFAC - IFAC'96, vol. D, 279-284. San Francisco.

DORMIDO, S., MELLADO, M. and GUILLEN, J. M. (1970). Consideraciones sobre la regulación de sistemas mediante técnicas de programación dinámica. *Revista de Automática*, **10**, 29-34. Madrid. (In Spanish.)

GUIGNABODET, J. J. G. (1961). *Analysis of some process control aspects of dynamic programming*. PhD Thesis. Washington University.

GUIGNABODET, J. J. G. (1963). Dynamic programming: cumulative errors in the evaluation of an optimal policy. *J. Basic Eng.*, June, 151-156.

KORSAK, A. J. and LARSON, R. E. (1970). A dynamic programming successive approximations technique with convergence proofs. Part II. *Automatica*, **6**, 253-260.

LARSON, R. E. (1968). *State increment dynamic programming*. American Elsevier Publishing Company, Inc., New York.

LARSON, R. E. and CASTI, J. L. (1978). *Principles of dynamic programming. Part I: Basic analytic and computational methods*. Marcel Dekker, Inc., New York.

LARSON, R. E. and CASTI, J. L. (1982). *Principles of dynamic programming. Part II: Advanced theory and applications*. Marcel Dekker, Inc., New York.

LARSON, R. E. and KORSAK, A. J. (1970). A dynamic programming successive approximations technique with convergence proofs. Part I. *Automatica*, **6**, 245-252.

LUUS, R. (1990a). Optimal control by dynamic programming using systematic reduction in grid size. *Int. J. Control*, **51**, 5, 995-1013.

LUUS, R. (1990b). Application of dynamic programming to high-dimensional non-linear optimal control problems. *Int. J. Control*, **52**, 1, 239-250.

MORENO, L., ACOSTA, L. and SÁNCHEZ, J. L. (1992). Design of Algorithms for Spacial-time Reduction Complexity of Dynamic Programming. *IEE Proc.-D*, **139**, 2, 172-180.

MORENO, L., ACOSTA, L., HAMILTON, A., MÉNDEZ, J. A., SÁNCHEZ, J. L. and PIÑEIRO, J. D. (1994). Dynamic Programming Approach for Nonlinear Systems. *IEE Proc.-Control Theory Appl.*, **141**, 6, 409-417.

MOSCA, E. (1995). *Optimal, predictive and adaptive control*. Prentice Hall, New Jersey.

PIERRE, D. A. (1986). *Optimization theory with applications*. Dover Publications, Inc., New York.

SNIEDOVICH, M. (1992). *Dynamic programming*. Marcel Dekker, Inc., New York.