Globally Convergent Approximate Dynamic Programming
Applied to an Autolander

J. Murray
Department of Electrical Engineering
State University of New York at Stony Brook
Stony Brook, NY 11790
and
C. Cox, R. Saeks, and G. Lendaris ¹
Accurate Automation Corporation²
7001 Shallowford Rd.,
Chattanooga, TN 37421
March 12, 2001

ABSTRACT

A globally convergent nonlinear Approximate Dynamic Programming algorithm is described, and an implementation of the algorithm in the linear case is developed. The resultant linear Approximate Dynamic Programming algorithm is illustrated via the design of an autolander for the NASA X-43 research aircraft, without a priori knowledge of the X-43's flight dynamics.

1. INTRODUCTION

Unlike the many soft computing applications where it suffices to achieve a "good approximation most of the time", a flight control system must be stable all of the time. As such, if one desires to learn a flight control law in real-time, a fusion of soft computing techniques to learn the appropriate control law with hard computing techniques to maintain the stability constraint and guarantee convergence is required. The objective of this paper is to describe a globally convergent Approximate Dynamic Programming Algorithm which uses soft computing techniques to learn the optimal cost (or return) functional for a stabilizable nonlinear aircraft with unknown dynamics and hard computing techniques to verify the stability and convergence of the algorithm. This algorithm is then specialized to the linear case and used to design an autolander for the NASA X-43 research aircraft. See 9, for the details of the nonlinear case and its applications.

The present work has its roots in the approximate Dynamic Programming / adaptive critic concept where soft computing techniques are used to approximate the solution of a dynamic programming algorithm without the explicit imposition of a stability or convergence constraint, and the authors' stability criteria for these algorithms. Alternatively, a number of authors have combined hard and soft computing techniques to develop tracking controllers. These include Lyapunov synthesis techniques using both neural and fuzzy learning laws, sliding mode techniques, and input-output techniques.

The centerpiece of Dynamic Programming is the Hamilton Jacobi Bellman (HJB) Equation which one solves for the optimal cost functional, \( V^*(x, t) \). This equation characterizes the cost to drive the initial state, \( x_0 \), at time \( t_0 \) to a prescribed final state using the optimal control. Given the optimal cost functional, one may then solve a second partial differential equation (derived from the HJB Equation) for the corresponding optimal control law, \( k^*(x, t) \), yielding an optimal cost functional / optimal control law pair, \( (V^*, k^*) \).

Although direct solution of the Hamilton Jacobi Bellman Equation is computationally untenable (the so-called "curse of dimensionality"), the HJB Equation and the relationship between \( V^* \) and the corresponding control law, \( k^* \), derived therefrom, serves as the basis of the Approximate Dynamic Programming Algorithm developed in the present paper. In this algorithm we start with an initial cost functional / control law pair, \( (V_0, k_0) \), where \( k_0 \) is a stabilizing control law for the plant, and construct a sequence of cost functional / control law pairs, \( (V_k, k_k) \), in real-time, which converge to the optimal cost functional / control law pair, \( (V^*, k^*) \).

Indeed, it is shown in that with the appropriate technical assumptions this process is:

* globally convergent to
* the optimal cost functional, \( V^* \), and the optimal control law, \( k^* \), and is
  * stepwise stable, i.e., \( k_k \) is a stabilizing controller at every iteration with Lyapunov function, \( V_k \).

¹. On Sabbatical Leave from Portland State University.
². This research performed in part on NSF SBIR contract DMI-9983287 and NASA Ames SBIR Contract NAS2-99047.
While one must eventually explore the entire state space (probably repeatedly) in any (truly) nonlinear control problem with unknown dynamics, in the above described Approximate Dynamic Programming Algorithm one must explore the entire state space at each iteration of the algorithm. Unfortunately, this is not feasible and is tantamount to fully identifying the plant dynamics at each iteration of the algorithm. As such, the present paper is devoted to the development of an implementation of the Approximate Dynamic Programming Algorithm in the linear case, which requires only local exploration of the state space, and its application in an X-43 autolander.

2. APPROXIMATE DYNAMIC PROGRAMMING ALGORITHM

In the formulation of the Approximate Dynamic Programming Algorithm and Theorem, we use the following notation for the state and state trajectories associated with the plant. The variable "," denotes a generic state while "x0" denotes an initial state, "t" denotes a generic time and "t0" denotes an initial time. We use the notation, x(t0), for the state trajectory produced by the plant (with an appropriate control) starting at initial state x0 at some implied initial time, and the notation, u(x(t0), for the corresponding control. Finally, the state reached by a state trajectory at time "t" is denoted by x = x(t0, t), while the value of the corresponding control at time "t" is denoted by u = u(x(t0, t)).

For the purposes of the present paper, we consider a stabilizable time-invariant input affine plant of the form

\[ x = f(x, u) = a(x) + b(x)u; \quad x(t_0) = x_0 \tag{1} \]

with input quadratic performance measure

\[ J = \frac{1}{2} \int [q(x(x_0, \lambda), u(x_0, \lambda))] dx + 1 \tag{2} \]

Here a(x), b(x), q(x), and r(x) are \( C^\infty \) matrix valued functions of the state where: a(0) = 0, producing a singularity at (x, u) = (0, 0); the eigenvalues of \( \frac{da(0)}{dx} \) have negative real parts, i.e., the linearization of the uncontrolled plant at zero is exponentially stable; q(x) > 0, x = 0; q(0) = 0; r(x) has a positive definite Hessian at x = 0, i.e., any non-zero state is penalized independently of the direction from which it approaches 0; and r(x) > 0 for all x.

The goal of the Approximate Dynamic Programming Algorithm is to adaptively construct an optimal control, \( u^*(x(t_0)) \), which takes an arbitrary initial state, \( x_0 \) at \( t_0 \) to the singularity at \( (0,0) \), while minimizing the performance measure J.

Since the plant and performance measure are time invariant, the optimal cost functional and optimal control law are independent of the initial time, \( t_0 \) which we may, without loss of generality, take to be 0; i.e.

\[ J^*(x_0) = J^*(x(t_0)) \]

Even though the optimal cost functional is defined in terms of the initial state, it is a generic function of the state, \( J^*(x) \), and is used in this form in the Hamilton Jacobi Bellman Equation and throughout the paper. Finally, we adopt the notation \( J^*(x) = a(x) + b(x)k^*(x) \), for the optimal closed loop feedback system. Using this notation, the Hamilton Jacobi Bellman Equation then takes the form

\[ \frac{dJ^*(x)}{dx} = -J(x, k^*(x)) = -q(x) - k^{T}(x)r(x)k(x) \tag{3} \]

in the time-invariant case.

Differentiating the HBJ Equation (3) with respect to \( u^* = k^*(x) \) now yields

\[ \frac{dJ^*(x)}{dx} b(x) = -2k^{T}(x)r(x) \tag{4} \]

or equivalently

\[ u = k^*(x) = \frac{1}{2} -1(x) b^{T}(x) \left[ \frac{dJ^*(x)}{dx} \right] \tag{5} \]

which is the desired relationship between the optimal control law and the optimal cost functional. Note, that an input quadratic performance measure is required to obtain the explicit form for \( k^* \) in terms of \( J^* \) of Equation 5, though a similar implicit relationship can be derived in the general case. (See12 for a derivation of this result).

Given the above preparation, we may now formulate the desired Approximate Dynamic Programming Algorithm as follows.

Approximate Dynamic Programming Algorithm:

1. Initialize the algorithm with a stabilizing cost functional / control law pair \( (V_0, k_0) \), where \( V_0(x) \) is a \( C^\infty \) function, \( V_0(x) > 0, x \neq 0; V_0(0) = 0 \), with a positive definite Hessian at \( x = 0, \frac{d^2 V_0}{dx^2}(0) > 0 \); and \( k_0(x) \) is the \( C^\infty \) control law, \( u = k_0(x) = \frac{1}{2} -1(x) b^{T}(x) \left[ \frac{dV_0(x)}{dx} \right] \).

2. For \( i = 0, 1, 2, \ldots \) run the system with control law, \( k_i \), from an array of initial conditions, \( x_0 \) at \( t_0 = 0 \), recording the resultant state trajectories, \( x_i(t_0, t) \), and control inputs.
The Approximate Dynamic Programming Algorithm is characterized by the following Theorem.

**Approximate Dynamic Programming Theorem:** Let the sequence of cost functional / control law pairs \( (V_i, k_i) \); \( i = 0, 1, 2, \ldots \); be defined by, and satisfy the conditions of the Approximate Dynamic Programming Algorithm. Then,

1. \( V_{i+1}(x) \) and \( k_{i+1}(x) \) exist, where \( V_{i+1}(x) \) and \( k_{i+1}(x) \) are continuous functions with
   \[
   V_{i+1}(x) > 0, x \neq 0; \quad V_{i+1}(0) = 0; \quad \frac{d^2V_{i+1}(0)}{dx^2} > 0 ;
   \]
   \( i = 0, 1, 2, \ldots \)

2. The control law, \( k_{i+1} \), stabilizes the plant (with Lyapunov function \( V_{i+1}(x) \)) for all \( i = 0, 1, 2, \ldots \), and that the eigenvalues of \( \frac{d^2V_{i+1}(0)}{dx^2} \) have negative real parts.

3. The sequence of cost functionals, \( V_{i+1} \), converge to the optimal cost functional, \( V^* \).

Note that in ii), the existence of the Lyapunov function \( V_{i+1}(x) \) together with the eigenvalue condition on \( \frac{d^2V_{i+1}(0)}{dx^2} \) implies that the closed loop system, \( F_{i+1}(x) \), is exponentially stable, rather than asymptotically stable, as implied by the existence of the Lyapunov function alone. A proof of this theorem appears in reference 9.

**3. THE LINEAR CASE**

The purpose of this section is to develop an implementation of the Approximate Dynamic Programming algorithm for the linear case, where local exploration of the state space at each iteration of the algorithm is sufficient, yielding a computationally tractable algorithm. For this purpose we consider a linear time-invariant plant

\[
\dot{x} = Ax + Bu; \quad x(t_0) = x_0 \quad (7)
\]

with the quadratic performance measure

\[
J = \int_{t_0}^{t_1} [x^T(x_0, \lambda)Q(x_0, \lambda) + u^T(x_0, \lambda)Ru(x_0, \lambda)]dt \quad (8)
\]

where \( Q \) is a positive matrix, while \( R \) is positive definite. For this case \( P^*(x) = x^TP^*x \) is a quadratic form, where \( P^* \) is a
positive definite matrix. As such, \( \frac{dV}{dx}(x) = 2x^TP\) and 
\[ u = K^T x = -R^{-1}B^TP\]

To implement the Approximate Dynamic Programming Algorithm in the linear case, we initialize the algorithm with a quadratic cost functional, \( V_P(x) = x^TPx \) and 
\[ K_0 = -R^{-1}B^TP_0 \]. Now, assuming that \( V_2(x) = x^TPx \) is quadratic and 
\[ K_i = -R^{-1}B^TP_i \,… \text{ for both } \]
\[ F_i(x) = [A-BK_i]x = [A-BR^{-1}B^TP_i]x \]

such, the state trajectories for the plant with control law \( K_i \) can be expressed in the exponential form \( x_i(x_0,t) = e^{F_i}x_0 \), while the corresponding control is 
\[ u_i(x_0,t) = K_i e^{F_i}x_0 \]. As such,
\[ V_{i+1}(x_0) = \int_0^T \left[ e^{T\lambda}x_0 + u_i(x_0,\lambda)Ru_i(x_0,\lambda) \right] d\lambda \]

\[ = \int_0^T e^{T\lambda}e^{\lambda \sigma}_{xx} + x_0^T e^{\lambda \sigma}K_i Q e^{\lambda \sigma}x_0 d\lambda \]

\[ = x_0^T \left[ \int_0^T e^{F_i} \left( Q + K_i^T R K_i \right)e^{F_i} \right] x_0 = x_0^T P_{i+1} x_0 \]

Now, since \( F_i \) is asymptotically stable the integral of Equation exists, confirming that \( V_{i+1}(x_0) = x^{T}\tilde{P}_{i+1}x \) is also quadratic. Moreover, the integral defining \( P_{i+1} \) is the “well known” integral form of the solution of the Linear Lyapunov Equation
\[ P_{i+1} + F_i^T P_{i+1} = -[Q + P_i BR^{-1}B^TP_i] \]

As such, in the linear case, rather than directly evaluating the integral of Equation one can iteratively solve for \( P_{i+1} \) in terms of \( P_i \) by solving the Linear Lyapunov Equation (10). Although the \( A \) matrix for the plant is implicit in \( F_i = [A-BR^{-1}B^TP_i] \), one can estimate \( F_i \) directly from measured data without a priori knowledge of \( A \). To this end, one runs the system using control law \( K_i \) over some desired time interval, and observes the state at \( n \) (the dimension of the state space) or more points, \( , j = 1, 2, ..., n \); while (numerically) estimating the time derivative of the state at the same set of points; \( \dot{x}_j = \dot{x}_j, j = 1, 2, ..., n \). Now, since \( F_i \) is the closed loop system matrix for the plant with control law \( K_i, \dot{x}_j = F_j^T x_j \); or equivalently \( \dot{x}_j = F_j^T x_j \), where \( X_i = [x_1 x_2 ... x_n] \). Assuming that the points where one observes the state are linearly independent, one can then solve for \( F_i \) from the observations via the equality \( F_i = \dot{x}_j^T X_j^{-1} \), yielding the alternative representation of the Linear Lyapunov equation
\[ P_{i+1} \left[ \dot{x}_j^T X_j^{-1} \right] + \left[ \dot{x}_j^T X_j^{-1} \right] P_{i+1} = -[Q + P_i BR^{-1}B^TP_i] \]

which can be solved for \( P_{i+1} \) in terms of \( P_i \) without a priori knowledge of \( A \). Moreover, one can circumvent the requirement that \( B \) be known via the pre-compensation technique of reference 12.

As such, Equation can be used to implement the Approximate Dynamic Programming Algorithm without a priori knowledge of the plant. Moreover, since \( F_i = \dot{x}_j^T X_j^{-1} \) is asymptotically stable, Equation 11 always admits a well defined positive definite solution, \( P_{i+1} \), while there are numerous numerical solution techniques for solving this class of Linear Lyapunov Equations. Moreover, unlike the full nonlinear algorithm, this implementation of the Approximate Dynamic Programming Algorithm requires only local information at each iteration. Finally, if one implements the above algorithm off-line to construct the optimal controller for a system with known dynamics, using \( F_i \) at each iteration in lieu of \( \dot{x}_j^T X_j^{-1} \), then the algorithm reduces to the Newton-Raphson iteration for solving the matrix Riccati Equation.

As an alternative to the above Linear Lyapunov Equation implementation, one can formulate an alternative implementation of linear Approximate Dynamic Programming Algorithm using local information along a single state trajectory, \( x_i(x_0,\lambda) \), and the corresponding control, \( u_i(x_0,\lambda) = K_i x_i(x_0,\lambda) \), starting at initial state \( x_0 \) and converging to the singularity at \( (0,0) \). Indeed, for this trajectory one may evaluate \( V_{i+1}(x_0) \) via
\[ V_{i+1}(x_0) = \int_0^T \left[ e^{T\lambda}x_0 + u_i(x_0,\lambda)Ru_i(x_0,\lambda) \right] d\lambda \]

\[ = x_0^T P_{i+1} x_0 \]

since the plant and control law are time-invariant. More generally, for any initial state, \( x_j = x_j(x_0,\lambda) \), along this trajectory
Now, since the positive definite matrix $P_{i+1}$ has only $q = n(n+1)/2$ independent parameters, one can select $q$ (or more) initial states along this trajectory; $x_j$, $j = 1, 2, ..., q$, and solve the set of simultaneous equations

$$x_j^T P_{i+1} x_j = v_i^+(x_j); \quad j = 1, 2, ..., q$$

for $P_{i+1}$. Equivalently, applying the matrix Kronecker product formula,

$$\text{vec}(ABC) = [C^T \otimes A] \text{vec}(B)$$

where the "vec" operator maps a matrix into a vector by stacking its columns on top of one another, one may transform Equation into a $q \times n^2$ matrix equation

$$
\begin{bmatrix}
  x_1^T \
  x_2^T \
  \vdots \
  x_q^T
\end{bmatrix}
\begin{bmatrix}
  x_1^T \
  x_2^T \
  \vdots \
  x_q^T
\end{bmatrix}
= \begin{bmatrix}
  v_1^+(x_1) \
  v_1^+(x_2) \
  \vdots \
  v_1^+(x_q)
\end{bmatrix}
\begin{bmatrix}
  x_1^T \
  x_2^T \
  \vdots \
  x_q^T
\end{bmatrix}
\begin{bmatrix}
  x_1^T \
  x_2^T \
  \vdots \
  x_q^T
\end{bmatrix}
\begin{bmatrix}
  x_1^T \
  x_2^T \
  \vdots \
  x_q^T
\end{bmatrix}
$$

Now, let "vec" be the operator that maps an $n \times n$ matrix, $B$, to a $q = n(n+1)/2$ vector, vec$(B)$, by stacking the upper triangular part of its columns, $b_{ij}$, $i \leq j$, on top of one another. Now, if $B$ is symmetric, vec$(B)$ fully characterizes $B$ and, as such, one may define an $n^2 \times q$ matrix, $S$, which maps vec$(B)$ to vec$(B)$ for any symmetric matrix, $B$. As such, one may express Equation 14 in the form of a $q \times q$ matrix equation in the unknown vec$(P_{i+1})$,

$$
\begin{bmatrix}
  x_1^T \
  x_2^T \
  \vdots \
  x_q^T
\end{bmatrix}
\begin{bmatrix}
  x_1 \
  x_2 \
  \vdots 
\end{bmatrix}
S \text{vec}(P_{i+1}) = \begin{bmatrix}
  v_1^+(x_1) \
  v_1^+(x_2) \
  \vdots 
\end{bmatrix}
\begin{bmatrix}
  x_1 \
  x_2 \
  \vdots 
\end{bmatrix}
\begin{bmatrix}
  x_1 \
  x_2 \
  \vdots 
\end{bmatrix}
$$

As such, assuming that the points where one observes the state are chosen to guarantee that Equation 15 has a unique solution, one can solve Equation 15 for a unique symmetric $P_{i+1}$. Moreover, since the general theory implies that Equation 15 has a positive definite solution, the unique symmetric solution of Equation 15 must, in fact, be positive definite. As such, one can implement the Approximate Dynamic Programming Algorithm for a linear system by solving Equation 15 for $P_{i+1}$, instead of Equation 11.

Figure 1: a) NASA X-43 (HyperX) and b) its Glide Path

4. X-43 AUTOLANDER

To illustrate the implementation of the Approximate Dynamic Programming Algorithm in the linear case, we developed an autolander for the NASA X-43 (or HyperX)\textsuperscript{14}. The X-43, shown in Figure 1a, is an experimental testbed for an advanced scramjet engine operating in the Mach 7-10 range. In its present configuration, the X-43 is an expendable test vehicle, which will be launched from a Pegasus missile, perform a flight test program using its scramjet engine, after which it will crash into the ocean. The purpose of the simulation described here was to evaluate the feasibility of landing a follow-on series of X-43s. To this end, we developed an autolander for the X-43 designed to follow the glide path illustrated in Figure 1b, using the Approximate Dynamic Programming Algorithm, and simulated its performance using a 6 degree-of-freedom linearized model of the X-43.

This model has eleven states and four inputs. To stress the adaptive controller, the simulation used an extremely steep glide path angle. Indeed, so steep that the drag of the aircraft was initially insufficient to cause the aircraft to fall fast enough, requiring negative thrust. Of course, in practice one would never use such a steep glide slope, alleviating the requirement for thrust reversers in the aircraft. To illustrate the adaptivity of the controller, no a priori knowledge of either the $A$ or $B$ matrices for the X-43 model was provided to the controller. A "trim routine" is used to calculate the steady state settings of the aircraft control surfaces required to achieve the desired flight conditions, with the state variables and controlled inputs for the flight control system taken to be the deviations from the trim point. In the present example the trim control was calculated to maintain the aircraft on the specified glide slope. The performance of the X-43 autolander is summarized in Figure 2a where the altitude and lateral errors from the glide path and the vertical component of the aircraft velocity (sink rate) along the glide path are plotted. After correcting for the initial deviation from trim, the autolander brings the aircraft to, and maintains it on, the glide path. The control values employed by the autolander...
to achieve this level of performance are well within the dynamic range of the X-43's controls.

To evaluate the adaption rate of the autolander, the "cost-to-go" from the initial state is plotted as a function of time as the controller adapts in Figure 2b. As expected, the cost-to-go jumps from the low initial value associated with the initial guess, \( P_i \), to a relatively high value, and then decays monotonically to the optimal value as the controller adapts. Although the theory predicts that the cost-to-go jump should occur in a single iteration, a filter was used to smooth the adaptive process in our implementation, which spreads the initial cost-to-go jump over several iterations.

Our goal in the preceding has been to develop an implementation of Approximate Dynamic Programming algorithm requiring only local exploration of the state space at each iteration, illustrating the resultant algorithm via the X-43 autolander. Two additional implementations of the algorithm for the nonlinear case, also requiring only local exploration of the state space, are described in reference 9, together with a detailed proof of the Approximate Dynamic Programming Theorem in the nonlinear case.

5. REFERENCES