Cheap Newton Steps for Optimal Control Problems: Automatic Differentiation and Pantoja’s Algorithm

Bruce Christianson

Numerical Optimisation Centre, University of Hertfordshire: Hatfield, England, Europe

November 1996, revised November 1997

In this paper we discuss Pantoja’s construction of the Newton direction for discrete time optimal control problems.

We show that Automatic Differentiation techniques can be used to calculate the Newton direction accurately, without requiring extensive re-writing of user code, and at a surprisingly low computational cost: for an $N$-step problem with $p$ control variables and $q$ state variables at each step, the worst case cost is $6(p + q + 1)$ times the computational cost of a single target function evaluation, independent of $N$, together with at most $p^3/3 + p^2(q + 1) + 2p(q + 1)^2 + (q + 1)^3$, i.e., less than $(p+q+1)^3$, floating point multiply-and-add operations per timestep. These costs may be considerably reduced if there is significant structural sparsity in the problem dynamics.

The systematic use of checkpointing roughly doubles the operation counts, but reduces the total space cost to the order of $4pN$ floating point stores.

A naïve approach to finding the Newton step would require the solution of an $Np \times Np$ system of equations together with a number of function evaluations proportional to $Np$, so this approach to Pantoja’s construction is extremely attractive, especially if $q$ is very small relative to $N$.

Straightforward modifications of the AD algorithms proposed here can be used to implement other discrete time optimal control solution techniques, such as differential dynamic programming (DDP), which use state-control feedback.

The same techniques also can be used to determine with certainty, at the cost of a single Newton direction calculation, whether or not the Hessian of the target function is sufficiently positive definite at a point of interest. This allows computationally cheap post-hoc verification that a second-order minimum has been reached to a given accuracy, regardless of what method has been used to obtain it.
1. Introduction. Consider the following optimal control problem: choose $u_i \in \mathbb{R}^p$ for $0 \leq i < N$ so as to minimize
\[ z = F(x_N) \]
where $x_0$ is some fixed constant and
\[ x_{i+1} = f_i(x_i, u_i) \quad \text{for} \quad 0 \leq i < N \]
Here each $f_i$ is a smooth map from $\mathbb{R}^q \times \mathbb{R}^p \to \mathbb{R}^q$ and $F$ is a smooth map from $\mathbb{R}^q$ to $\mathbb{R}$. Note that for notational convenience we have assumed $p$ and $q$ independent of $i$, but our methods and results can be generalized if this restriction is relaxed and $p, q$ are replaced throughout by $p_i, q_i$.

The more usual formulation of a discrete time optimal control problem is:
choose $u_i$ so as to minimize
\[ z = \sum_{i=0}^{N-1} F_i(x_i, u_i) + F_N(x_N) \]
but this is equivalent to a problem in the form introduced above. To see this, adjoin to each state $x_i$ a new component $v_i \in \mathbb{R}$ defined by
\[ v_0 = 0 \quad v_{i+1} = v_i + f_i(x_i, u_i) \]
and then define $F(x_N, v_N) = v_N + F_N(x_N)$. Consequently, we lose nothing by restricting attention to target functions of the form $z = F(x_N)$.

In 1983, Pantoja described a stagewise construction of the Newton direction for discrete optimal control problems of this form [18][19]. An elementary account of Pantoja’s construction and its properties is given elsewhere [9].

In this paper, we show how Automatic Differentiation can be combined with Pantoja’s algorithm and a checkpointing technique in such a way as to allow accurate evaluation of the Newton direction at an extremely low computational cost.

In the next section we give a brief introduction to the Automatic Differentiation techniques which we shall use later. In Section 3 we introduce Pantoja’s algorithm. In Section 4 we show how forward and reverse Automatic Differentiation techniques can be combined so as to provide an efficient implementation of Pantoja’s algorithm, and give an analysis of the corresponding time and space bounds. In Section 5 we show how to incorporate checkpointing and discuss the effect of this on the time and space bounds. We summarise our conclusions in the final section.

2. Automatic Differentiation. Automatic differentiation (AD) is a set of techniques for obtaining derivatives of numerical functions to the same order of accuracy as the function values themselves, but without the labour of forming
explicit symbolic expressions for the derivative functions [13][20]. Automatic differentiation works by repeated use of the chain rule, but applied to numerical values rather than to symbolic expressions. This may be achieved either by pre-processing the function code, or by using operator overloading: for convenience, we briefly describe the latter approach here.

The forward accumulation technique of AD associates with each program variable a vector \( v \), which contains numerical values for the partial derivatives of \( v \) with respect to each of the independent variables. The combined structure \( V = (v, \dot{v}) \) is called a doublet. The doublets \( U_i \) corresponding to the independent variables \( u_i \) are initialized by setting \( \dot{u}_i \) to be the \( i \)-th Cartesian unit vector. We write this (rather loosely) as

\[
[\dot{u}] = [I]
\]

The floating point arithmetic operators are overloaded so as to operate correctly on the numerical values in the vector parts: for example

\[
\sin(V) = (\sin(v), \cos(v) \ast \dot{v}) \quad V \ast W = (v \ast w, v \ast \dot{w} + \dot{v} \ast w)
\]

Re-compiling the same code which evaluates \( y = f(u) \) with all real variables re-declared to be doublets will cause it to evaluate \( Y = f(U) \), following which we have \( \dot{y} = f'(u) \dot{u} = f'(u) \).

The reverse accumulation technique of AD works by overloading the floating point operations so that they record a trace of the program evaluation for \( y = f(u) \). The trace consists of a list of elementary floating point operations in the order in which they were performed by the program, together with the address(es) of their argument(s) and the numerical values of their partial derivative(s) at the point in question. For example the operations \( v := w \ast u; w := \sin(v) \) would record the values of \( u; w \) and of \( \cos(v) \) respectively.

A floating point adjoint variable \( \bar{v} \) (initially zero) is associated with each program variable \( v \). The adjoint variable \( \bar{v} \) is updated so that it contains the numerical value of the partial derivative of the dependent variable \( y \) with respect to \( v \) at the corresponding point in the trace. These updates are calculated numerically in the reverse order to the function evaluation, whence the term ‘reverse accumulation’. Initially the adjoint \( \bar{y} \) is set to 1.0, and (for example) the adjoint operations corresponding to \( v := w \ast u; w := \sin(v) \) are the operations

\[
\bar{v} += \bar{w} \ast \cos(v), \quad \bar{w} := 0, \quad \bar{w} += u \ast \bar{v}, \quad \bar{u} += \bar{v} \ast w, \quad \bar{v} := 0.
\]

At the end of the reverse pass through the trace for the code which evaluates \( y = f(u) \) we have \([\bar{u}] = \bar{y} [f'(u)]^T = [f'(u)]^T \). For further details see [11][6].

The forward and reverse techniques can be combined to allow us to calculate Hessians. We embed doublet arithmetic into an implementation of reverse AD: each program variable value is a doublet rather than a real, and so is the
corresponding adjoint variable value. Preparing such an AD package when operator overloading has been used to implement forward and reverse separately is a simple matter of re-declaring the relevant fields in the trace type.

After setting \([\hat{u}] = [I]\) we calculate \(Y = f(U)\) giving \(\hat{y} = f'(u)\) as before. We then initialize \(\hat{Y}\) by setting \(\hat{y} = 1.0, \hat{y} = 0\) and perform the reverse pass in doublet arithmetic. For example the reverse accumulation step

\[
\hat{V} = \hat{W} \ast \cos(V)
\]

will (by operator overloading) be executed as

\[
\hat{v} = \hat{w} \ast \cos(v) \quad \hat{b} = \hat{w} \ast \cos(v) - \hat{w} \ast \sin(v) \ast \hat{v}
\]

Following the reverse accumulation pass we will have

\[
[\hat{u}] = \hat{y}[f'(u)]^T = [f'(u)]^T
\]

as before, and

\[
[\hat{b}] = \hat{y}[f''(u)]\hat{u} + \hat{y}[f'(u)]^T = [f''(u)]
\]

The important points to note about this process of automatic Hessian evaluation are as follows. First, the same code that is used to calculate the function \(y\) can be used to calculate the Hessian \(H\) of \(y\), with only minor modifications: variables must be declared to have a new type, independent variables must be explicitly identified (so as to contain the correct cartesian vector) and a function call to perform the reverse pass must be inserted after the evaluation of \(f\). Secondly, the total amount of floating point arithmetic involved is remarkably small: If there are \(r\) independent variables, then the computational cost of a complete Hessian is less than \(6r + 4\) times the cost of evaluating \(y\), where the cost is measured in floating point multiply-and-add operations. For further details see [6].

3. Pantoja's Algorithm. Define the adjoint problem corresponding to our original problem as follows. Define variables \(\tilde{x}_i \in \mathbb{R}^q\) and \(\tilde{u}_i \in \mathbb{R}^q\) by setting

\[
\tilde{x}_N = F'(x_N) \quad \tilde{x}_i = [f'_{x,i}]^T \tilde{x}_{i+1} \quad \tilde{u}_i = [f'_{u,i}]^T \tilde{x}_{i+1} \quad \text{for } 0 \leq i < N
\]

where \(f'_{x,i}\) and \(f'_{u,i}\) are the Jacobians of \(f_i\) with respect to \(x_i\) and \(u_i\) respectively, evaluated at \((x_i, u_i)\). Then \(\tilde{u}_i = \partial z / \partial u_i\) by the chain rule.

The adjoint problem is extremely similar to the reverse accumulation technique introduced in the previous section, but applied to complete time steps rather than at the level of individual floating point operations.

Suppose that we linearize both the original and the adjoint problems at a starting point \(u^0\), so that the \(\tilde{u}_i\) are approximated by linear functions of the control variables \(u\). Let \(u^{\text{new}}\) be the point at which these linear functions all
vanish. Then the Newton direction is the vector \( u^{new} - u^0 \). The details of this construction are embedded in Pantoja’s algorithm which follows.

**Notation.** Let \( g \) be the block vector with \( i \)-th block given by

\[
g_i = \hat{u}_i = \left[ \frac{\partial z}{\partial u_i} \right]
\]

Let \( H \) be the block matrix with \((i,j)\)-th block given by

\[
H_{ij} = \left[ \frac{\partial^2 z}{\partial u_i \partial u_j} \right]
\]

For given values of \( t_i, b_i \in \mathbb{R}^p : 0 \leq i < N \) we write \( Ht = b \) to denote

\[
\sum_{j=0}^{N-1} H_{ij} t_j = b_i \quad \text{for } 0 \leq i < N
\]

**Algorithm 3.1.** (Pantoja) Given a starting position \( u_i \) to obtain values for \( t_i \) such that \( Ht = -g \).

Step 1. For \( i \) from 1 up to \( N \) calculate \( x_i \) by

\[
x_{i+1} = f_i(x_i, u_i)
\]

where \( x_0 \) is a fixed constant.

Step 2. Define \( \bar{x}_N, a_N \in \mathbb{R}^q, D_N \in \mathbb{R}^{q \times q} \) by

\[
\bar{x}_N = a_N = F'(x_N) \quad D_N = F''(x_N)
\]

Step 3. For \( i \) from \( N-1 \) down to 0 calculate \( \bar{x}_i, a_i \in \mathbb{R}^q, \bar{u}_i, c_i \in \mathbb{R}^p, A_i, D_i \in \mathbb{R}^{q \times q}, B_i \in \mathbb{R}^{p \times q}, C_i \in \mathbb{R}^{p \times p} \) by

\[
\bar{x}_i = [f'_{x,i}]^T \bar{x}_{i+1}
\]

\[
\bar{u}_i = [f'_{u,i}]^T \bar{x}_{i+1}
\]

\[
A_i = [f'_{u,i}]^T D_{i+1} \left[ f'_{x,i} \right] + \left( \bar{x}_{i+1} \right)^T \left[ f''_{x,x,i} \right]
\]

\[
B_i = [f'_{u,i}]^T D_{i+1} \left[ f'_{u,i} \right] + \left( \bar{x}_{i+1} \right)^T \left[ f''_{u,x,i} \right]
\]

\[
C_i = [f'_{u,i}]^T D_{i+1} \left[ f'_{u,i} \right] + \left( \bar{x}_{i+1} \right)^T \left[ f''_{u,u,i} \right]
\]

where \([\cdot] \) denotes evaluation at \((x_i, u_i)\), and we write (for example)

\[
\left( [f'_{u,i}]^T D_{i+1} \left[ f'_{x,i} \right] \right)_{j,k} \quad \text{for } \sum_{l=1}^{q} \sum_{m=1}^{q} \frac{\partial (x_{i+1})_l}{\partial (u_i)_j} \left( D_{i+1} \right)_{l,m} \frac{\partial (x_{i+1})_m}{\partial (u_i)_k} \text{ etc.}
\]
If $C_i$ is singular then the algorithm fails, otherwise set
\[
D_i = A_i - B_i^T C_i^{-1} B_i \\
c_i = [f'_{u,i}]^T a_{i+1} \\
a_i = [f'_{x,i}]^T a_{i+1} - B_i^T C_i^{-1} c_i
\]

Step 4. For $i$ from 0 up to $N - 1$ calculate $t_i \in \mathbb{R}^p$, $s_{i+1} \in \mathbb{R}^q$ by
\[
t_i = -C_i^{-1}(B_i s_i + c_i) \\
s_{i+1} = [f'_{x,i}]^T s_i + [f'_{u,i}]^T t_i
\]
where $s_0$ is the fixed constant 0.

STOP

Proposition 3.2. Either Algorithm 3.1 fails because some $C_i$ is singular, or else at the end the $t_i$ satisfy $Ht = -g$. If all the $C_i$ defined in Step 3 of Algorithm P are invertible, then so is $H$.

If all the $C_i$ are positive definite, then so is $H$. Conversely, if $H$ is positive definite then all the $C_i$ are positive definite (and hence are invertible).

Proof: See [9].

4. Implementation using AD. In this section we describe how to use Automatic Differentiation to implement Algorithm 3.1, which is Algorithm 2.1 of [9]. Minor modifications can be made to implement the other algorithms described in [9]. The basic idea of the implementation is to note that the recurrence relations for $D_i$ correspond to a first order expansion of the forward and adjoint state equations.

Algorithm 4.1 (Pantoja with AD)

We start with stored values for $u_i : 0 \leq i < N$. Recall that $x_0$ is a fixed constant.

Step 1. For $i$ from 1 up to $N$ calculate $x_i$ using ordinary floating point arithmetic. Store $x_i$.

Step 2. Define doublets $X_N$ with scalar parts $x_N$ respectively, and vector parts (of length $q$) given by $\hat{x}_N = [I_q]$. Evaluate $Z = F(X_N)$ in doublets. We now have $\hat{z} = [F'(x_N)]$. Define $\hat{Z}$ by setting $\hat{z} = 1.0, \hat{Z} = 0_q$. Reverse through the trace for $Z$ to obtain the doublets $\hat{X}_N$. We have $\hat{x}_N = [F'(x_N)]^T, \hat{x}_N = [F''(x_N)]$. Set $a_N = \hat{x}_N, D_N = \hat{x}_N$. Delete the trace for $Z$.

Step 3. For $i$ from $N - 1$ down to 0 we calculate $\hat{x}_i, a_i, D_i$ as follows. We assume at each stage that the corresponding quantities are available for $i + 1$. 

6
Define doublets $X_i$ and $U_i$ with scalar parts $x_i, u_i$ respectively, and vector parts (of length $q + p$) given by

\[
\begin{bmatrix}
\hat{x}_i \\
\hat{u}_i
\end{bmatrix} = \begin{bmatrix}
I_0 & O \\
O & I_p
\end{bmatrix}
\]

Evaluate $X_{i+1} = f_i(X_i, U_i)$ in doublets. Now we have

\[
\begin{bmatrix}
\hat{x}_{i+1} \\
\hat{u}_{i+1}
\end{bmatrix} = \begin{bmatrix}
f_{x,i} & f'_{u,i}
\end{bmatrix}
\]

Define $\tilde{X}_{i+1}$ by setting $\tilde{x}_{i+1}$ to the supplied value and setting

\[
\begin{bmatrix}
\tilde{x}_{i+1} \\
\tilde{u}_{i+1}
\end{bmatrix} = \begin{bmatrix}
D_{i+1}f_{x,i} & D_{i+1}f'_{u,i}
\end{bmatrix}
\]

Reverse through the trace for $X_{i+1}$ to obtain the doublets $\tilde{X}_i, \tilde{U}_i$. Then

\[
\begin{bmatrix}
\tilde{x}_i \\
\tilde{u}_i
\end{bmatrix} = \begin{bmatrix}
A_i & B_i^T \\
B_i & C_i
\end{bmatrix}
\]

Calculate the column vectors $[f_{x,i}^T a_{i+1}, [f_{u,i}^T a_{i+1}]$. Adjoin these to form:

\[
\begin{bmatrix}
A_i & B_i^T \\
B_i & C_i
\end{bmatrix} \begin{bmatrix}
[f_{x,i}^T a_{i+1} \\
[f_{u,i}^T a_{i+1}]
\end{bmatrix}
\]

Row reduce this to obtain

\[
\begin{bmatrix}
A_i - B_i^T C_i^{-1} B_i & O \\
C_i^{-1} B_i & I
\end{bmatrix} = \begin{bmatrix}
D_i \\
C_i^{-1} B_i
\end{bmatrix} \begin{bmatrix}
a_i \\
C_i^{-1} c_i
\end{bmatrix}
\]

Now $\tilde{x}_i, a_i, D_i$ are available for the next iteration. Store the values $\tilde{x}_{i+1}, \tilde{u}_i$, $C_i^{-1} B_i, C_i^{-1} c_i$. Delete the trace for $X_{i+1}$.

Step 4. In ordinary arithmetic, for $i$ from 0 up to $N - 1$ calculate $t_i \in \mathbb{R}^p, s_{i+1} \in \mathbb{R}^q$ by

\[
t_i = -(C_i^{-1} B_i) s_i - (C_i^{-1} c_i)
\]

and

\[
s_{i+1} = [f_{x,i}^T] s_i + [f_{u,i}^T] t_i
\]

where $s_0$ is the fixed constant 0.

STOP

We now give some time and space bounds for Algorithm 4.1.
Proposition 4.2 The total computational cost of Algorithm 4.1 is less than
the cost of \(6(p + q + 1)\) evaluations of \(z\), regardless of \(N\), together with at most
\(p^3/3 + p^2(q + 1) + 2p(q + 1)^2 + (q + 1)^3\) floating point multiply-and-add operations
per time step.

The total storage requirement is bounded by \(2(p + q + 2)W\), where \(W\) is the
maximum number of floating point operations in any single \(f_i\) or \(F\), together
with at most \((q + 1)(2p + q)N\) floating point stores.

Proof: The floating point cost of the doublet calculation in Step 3 is at most
\(6(p + q) + 4\) times the corresponding scalar arithmetic cost in Step 1. The total
cost of Step 2 is at most \(6q + 4\) times the cost of an ordinary evaluation of \(F\). Step 1 itself adds one function evaluation. The computational cost of the
matrix multiplications in Step 3 of Algorithm 4.1 are \(q^3(p + q)\) multiply-and-add operations
to form \(\hat{X}_{i+1}\) and a further \(q(p + q)\) to form the products with \(a_{i+1}\).

The computational cost of the row-elimination operations in Step 3 is at
most \(p(q + 1)(p + q) + p^3/3\) multiply-and-add operations per time step.

The matrix multiplications in Step 4 add another \(pq\) multiply-and-add operations
to form \(t_i\) and \(q(p + q)\) to form \(s_{i+1}\).

The total number of floating point multiply-and-add operations per time step is thus less than

\[
p^3/3 + p^2q + 2pq^2 + q^3 + p^2 + 4pq + 2q^2 \leq p^3/3 + p^2(q + 1) + 2p(q + 1)^2 + (q + 1)^3
\]

We need enough space to store a single trace for (the largest) single \(f_i\), with
\(q + p\) vector doublets, ie \(2(p + q) + 4\) per floating point operation in \(f_i\). We also
need to store \(\hat{x}_{i+1}, \hat{u}_i, C_i^{-1}B_i, C_i^{-1}c_i\) for each of \(N\) values of \(i\), with storage costs
respectively \(q(p + q), p, pq, p\). The total is of order \(q(2p + q) + 2p\) per time step,
which is less than \((q + 1)(2p + q)N\) in total.

QED

Note that the number of additional floating point multiply-and-add operations
is bounded by \((3p/4 + q + 1)^3\).

If the computational complexity of evaluating \(f_i\) is significantly less than the
order of \(q(p + q)\) floating point operations, then there is likely to be redundancy
(eg sparsity or rank deficiency) in the structure of the Jacobians \(f'_{u,i}, f'_{v,i}\). If
there is such redundancy, then this can be exploited in the row reductions to
reduce the cost of Step 3. In either case, the total cost of the matrix operations
is likely to be reducible to the cost of about \(2(p + q + 1)\) additional function evaluations,
regardless of \(N\), assuming that \(p\) is not larger than \(q\). The operation count for our Algorithm 4.1 should be compared with that given by Coleman
and Liao [10, §2.1] viz \(p^3/3 + 2p^2q + 7pq^2/2 + 2q^3\).

Usually the requirement to be able to store the graph of the largest \(f_i\) (or \(F\)) is trivial relative to the other storage: provided the number of floating point
operations in any \(f_i\) is small relative to \(N\) the store required will be bounded
by \(2(p + q + 1)\) floating point stores per time step. If this requirement is not met, it is possible to perform the doublet calculation several times with shorter vector components, thus extracting the required matrices a block at a time [5]. Alternatively, it may be possible to split the function evaluation into two or more stages.

However the total space requirements appear infeasible if \(q\) is large. We discuss a strategy to address this problem in the next section.

5. Checkpointing. In Algorithm 4.1 we store a large amount of data long before we need it; for example values for all the \(C_i^{-1}B_i\) are stored during Step 3, ready to be used in Step 4. If \(q\) is large, this storage overhead may well be unacceptable. In this case, it would make more sense to store sufficient information to allow values to be re-computed when they are actually needed. For example, suppose that \(N\) is a million. If we store values for \(x_i, \tilde{x}_i, D_i\) just when \(i\) is a multiple of a thousand, then we can re-compute the values of \(C_i^{-1}B_i\) when we need them, in groups of a thousand at a time. This doubles the computational effort required (although much of this could be computed in parallel [1]) but reduces the storage requirement from a million records to a thousand records, plus a thousand checkpoints. Further development of this line of argument leads to the following algorithm.

Algorithm 5.1 (Pantoja with AD and checkpointing)

We start with stored values for \(u_i : 0 \leq i < N\). Recall that \(x_0\) is a fixed constant. For convenience, we assume that \(N = n^2\).

Step 1. For \(i\) from 1 up to \(N\) calculate \(x_i\) using ordinary floating point arithmetic. If \(i\) is a multiple of \(n\) then store \(x_i\).

Step 2. Calculate \(a_N = F'(x_N), D_N = F''(x_N)\) as in Algorithm 4.1.

Step 3. For \(j\) from \(n - 1\) down to 0 calculate \(\tilde{x}_{jn}, a_{jn}, D_{jn}\) as follows. We assume at each stage that the corresponding quantities are available for \(j + 1\). Recall that \(x_{jn}\) is available from Step 1.

For \(i\) from \(jn\) up to \((j + 1)n - 1\) recalculate \(x_i\) using ordinary floating point arithmetic. Store \(x_i\).

For \(i\) from \((j + 1)n - 1\) down to \(jn\) define doublets \(X_i\) and \(U_i\) and calculate \(\tilde{x}_i, a_i, D_i\) for the next iteration just as in Step 3 of Algorithm 4.1 Delete the trace for \(X_i\). Store \(\tilde{x}_{nj}, a_{nj}, D_{nj}\).

Step 4. For \(j\) from 0 up to \(n - 1\) proceed as follows. Recall that \(x_{nj}\) is available from Step 1, and that \(\tilde{x}_{(n+1)j}, a_{(n+1)j}, D_{(n+1)j}\) are available from Step 3.

For \(i\) from \(jn\) up to \((j + 1)n - 1\) recalculate \(x_i\) using ordinary floating point arithmetic. Store \(x_i\).

For \(i\) from \((j + 1)n - 1\) down to \(jn\) define doublets \(X_i\) and \(U_i\) and recalculate \(\tilde{a}_i, \tilde{x}_i, a_i, D_i\) just as in Step 3. Delete the trace for \(X_i\). Store \(\tilde{a}_i, C_i^{-1}B_i, C_i^{-1}c_i\).
For \( i \) from \( jn \) up to \((j + 1)n - 1\) calculate \( t_i = -(C_i^{-1} B_i) s_i - (C_i^{-1} c_i)\).

Recall that \( x_i \) is available from the first part of this step. Assume that \( s_i \) is available from the previous iteration. Define doublets (with vector parts of length 1) \( U_i, X_i \) by setting \( \bar{u}_i = t_i, \bar{s}_i = s_i \) and calculate \( X_{i+1} = f_i(X_i, U_i) \). Set \( s_{i+1} = \bar{x}_{i+1} \). Delete \( x_i, s_i \).

STOP

**Proposition 5.2** The total computational cost of Algorithm 5.1 is less than the cost of \( 12(p + q) + 15 \) evaluations of \( z \), regardless of \( N \), together with at most \( 2p^3/3 + 2p^2(q + 1) + 4p(q + 1)^2 + 2(q + 1)^3 \) floating point multiply-and-add operations per time step.

A total of \( 3pN \) floating point stores are required in order to maintain the values of \( u, \bar{u} \) and \( t \). The total storage required in addition to this is bounded by \( 2(p + q + 2)W \), where \( W \) is the maximum number of floating point operations in any single \( f_i \) or \( F \), together with

\[
(p + q + 2)(q + 1)\sqrt{N}
\]

floating point stores.

**Proof:** Step 1 represents the cost of a single function evaluation. The total cost of Step 2 is at most \( 6q + 4 \) times the cost of an ordinary evaluation of \( F \). The floating point cost of the doublet calculation in Step 3 is at most \( 6(p + q) + 4 \) times the corresponding scalar arithmetic cost in Step 1. The recalculation of the \( x_i \) adds one further function evaluation. The computational cost of the matrix multiplications in Step 3 of Algorithm 4.1 are (as before) \( q^2(p + q) \) multiply-and-add operations to form \( \bar{X}_{i+1} \) and a further \( q(p + q) \) to form the products with \( a_{i+1} \).

The computational cost of the row-elimination operations in Step 3 is the same as in Algorithm 4.1, but these operations are repeated in Step 4.

In Step 4 the re-evaluation of \( x_i \) and the doublet calculation of Step 3 are repeated, which adds another \( 6(p + q) + 5 \) function evaluations. The calculation of \( t_i \) requires \( pq \) multiply-and-add operations per time step, and the doublet arithmetic to compute \( s_{i+1} \) adds the cost of another 4 function evaluations.

The total number of function evaluations is thus \( 12(p + q) + 15 \) and the additional number of floating point multiply-and-add operations per time step is less than twice that required by Algorithm 4.1.

The storage requirement for the trace is the same as for Algorithm 4.1. We need to store \( C_i^{-1} B_i, C_i^{-1} c_i \) for each of \( n \) values of \( i \), with storage costs respectively \( pq, p \). We also need to store \( n \) checkpoints \( x_{nj}, x_{nj}, a_{nj}, D_{nj} \) with storage costs respectively \( q, q, q, q^2 \). This totals

\[
[p + q]q + (p + 3q)\sqrt{N}
\]

which is less than \((p + q + 2)(q + 1)/\sqrt{N}\) per time step.
Note that the number of additional floating point multiply-and-add operations is bounded by $2(3p/4 + q + 1)^3$.

Assume that $N > [(p + q + 2)(q + 1)]^2$ and that the number of floating point operations in any $f_i$ or $F$ is small compared with $N/(p + q)$. Then the cost of the working store can be reduced to below $N$ floating point stores. This is less than the cost of storing $u$ or $t$.

More sophisticated approaches are possible. For example, suppose that $N$ is a million. Algorithm 4.1 requires a million records of storage. Algorithm 5.1, with one level of checkpointing, requires storage for one thousand records and one thousand checkpoints, but also requires double the run time. If we employ two levels of checkpoint, we need only one hundred records, and two hundred checkpoints (one hundred at each level), but will require treble the run time. An algorithm with six levels of checkpoint requires ten records, sixty checkpoints, and seven times the run time. In general, we can reduce the storage from $N$ to order $\log N$ at the cost of a $\log N$ fold increase in run time. For further details, see for example [21] [14].

6. Conclusions. We have discussed the application of AD to Pantoja’s algorithm, and shown that the Newton step can be calculated for a discrete time optimal control problem for a very low computational cost. The pleasing feature of using AD is that existing code to evaluate the numerical value of the target function $z$ can be used, without extensive re-writing, to compute truncation-free values for the first and second derivatives required.

Similar implementations of other algorithms involving state-control feedback are also possible: for example the traditional differential dynamic programming (DDP) algorithm [16] simply substitutes $a_{i+1}$ for $x_{i+1}$ in the initialization of $X_{i+1}$ in Step 3 of Algorithm 4.1. Other algorithms given in [9] can also be implemented in this way: for example to determine a diagonal matrix $\Lambda$ such that $H + \Lambda$ is positive definite, to find a descent direction $t$ such that $(H + \Lambda)t = -g$, or to solve more general equations such as $(H + \Lambda)t = b$ with an arbitrary right hand side, as required for example by [10] to implement trust regions.

However Pantoja’s algorithm provides a useful tool even when the Newton direction is not being used to solve the optimization problem: it allows an inexpensive determination of whether the Hessian of the target function is positive definite at any point. This information is of utility to many optimization algorithms (particularly in the context of global optimization) and in particular allows post-hoc verification that a second order minimum has been reached. Similarly use of the algorithm to examine $H - \lambda I$ allows verification that the Hessian does not contain eigenvalues below a posited positive threshold. Such sensitivity analyses can in turn be used to verify the accuracy of the adjoint problem solution.
Further refinement of the approach described here using the techniques of [12][15][7][8] allows the dynamics of the problem to be expressed in terms of implicit equations $\phi(x_{i+1}, x_i, u_i)$ rather than explicitly. This in turn opens the prospect of applying similar techniques to problems arising from differential equations.

References


