Efficient Symbolic Differentiation for Graphics Applications

Brian Guenter
Microsoft Research

Abstract

Functions with densely interconnected expression graphs, which arise in computer graphics applications such as dynamics, space-time optimization, and PRT, can be difficult to efficiently differentiate using existing symbolic or automatic differentiation techniques. Our new algorithm, **D***, computes efficient symbolic derivatives for these functions by symbolically executing the expression graph at compile time to eliminate common subexpressions and by exploiting the special nature of the graph that represents the derivative of a function. This graph has a sum of products form; the new algorithm computes a factorization of this derivative graph along with an efficient grouping of product terms into subexpressions. For the problems in our test suite **D*** generates symbolic derivatives which are up to 4.6×10^3 times faster than those computed by the symbolic math program Mathematica and up to 2.2×10^5 times faster than the non-symbolic automatic differentiation program CppAD. In some cases the **D*** derivatives rival the best manually derived solutions.

Keywords: Symbolic differentiation

1 Introduction

Derivatives are essential in many computer graphics applications: optimization applied to global illumination and dynamics problems, computing surface normals and curvature, etc. Derivatives can be computed manually or by a variety of automatic techniques, such as finite differencing, automatic differentiation, or symbolic differentiation. Manual differentiation is tedious and error-prone; automatic techniques are desirable for all but the simplest functions. However, functions whose expression graphs are densely interconnected, such as recursively defined functions or functions that involve sequences of matrix transformations, are difficult to efficiently differentiate using existing techniques. These types of expressions occur in a number of important graphics applications.

For example, spherical harmonics, as used in PRT, have a natural recursive form whose expression graph has many common subexpressions. Optimization of the spherical harmonics coefficients [Sloan et al. 2005] can be made more efficient if a gradient can be computed but the recursive spherical harmonics equations are difficult to differentiate directly.

Another example is the dynamics equations of articulated figures. These have sequences of matrix transformations that have to be differentiated to solve the inverse dynamics and space-time optimization problems. This has proven remarkably difficult. More than 60 research papers[1] have been published on the topic of solving the inverse dymamics problem for robot manipulators. More than a decade elapsed before the first $O(n)$ solution was found and almost two decades passed before truly efficient $O(n)$ solutions were developed [Featherstone and Orin 2000]. Computing the gradient of the function to be minimized in space-time optimization for robot manipulators is also quite difficult: efficient, though complex, solutions have only recently been published [Martin and Bobrow 1997; Lee et al. 2005].

1.1 Related work

Up to now, there have been three basic ways of computing derivatives: finite differencing, automatic differentiation, and symbolic differentiation.

The finite difference method is both inaccurate and much less efficient, in general, than other techniques so it won't be discussed further.

Forward and reverse automatic differentiation are non-symbolic techniques independently developed by several groups in the 60s and 70s respectively [Griewank 2000; Rall 1981]. In the forward method derivatives and function values are computed together in a forward sweep through the expression graph. In the reverse method function values and partial derivatives at each node are computed in a forward sweep and then the final derivative is computed in a reverse sweep. Users generally must choose which of the two techniques to use on the entire expression graph, or whether to apply forward to some subgraphs and reverse to others. Some tools such as ADIFOR [Bischof et al. 1996] and ADIC [Bischof et al. 1997] automatically apply reverse at the statement level and then forward at the global level. Forward and reverse are the most widely used of all automatic differentiation algorithms.

The forward method is efficient for $\mathbb{R}^1 \rightarrow \mathbb{R}^n$ functions but may do n times as much work as necessary for $\mathbb{R}^n \rightarrow \mathbb{R}^1$ functions. Conversely, the reverse method is efficient for $f : \mathbb{R}^n \rightarrow \mathbb{R}^1$ but may do n times as much work as necessary for $f : \mathbb{R}^1 \rightarrow \mathbb{R}^n$. For $f : \mathbb{R}^n \rightarrow \mathbb{R}^m$ both methods may do more work than necessary.

Efficient differentiation can also be cast as the problem of computing an efficient elimination order for a sparse matrix [Griewank 2000; Griewank 2003] using heuristics which minimize fill in. However, as of the time of [Griewank 2003] good elimination heuristics that worked well on a wide range of problems remained to be developed. More recently, [Tadjouddine 2007] proposed an algorithm which used the Markowitz elimination heuristic (as noted

[1] See [Balafoutis and Patel 1991] chapter 5 for an extensive bibliography.

in [Griewank 2003] this heuristic can give results worse than either forward or reverse for some graphs) followed by graph partioning to perform interface contraction through the use of vertex separators. The algorithm for partitioning the graph remained to be implemented as future work, but the authors claim that, in the form described in the paper, it is limited to computational graphs with a few hundred nodes, roughly 2 orders of magnitude smaller than the largest problems we solve in our test suite. They suggested potential alternative algorithms which might address this problem but did not implement them. No test results appear in this paper so it is difficult to say how well it would perform in comparison to our new algorithm.

An extensive list of downloadable automatic differentiation software packages can be found at `http://www.autodiff.org`.

Symbolic differentiation has traditionally been the domain of expensive, proprietary symbolic math systems such as Mathematica. These systems work well for simple expressions but computation time and space grow rapidly, often exponentially, as a function of expression size, in practice frequently exceeding available memory or acceptable computation time.

1.2 Contributions

D* combines some of the best features of current automatic and symbolic differentiation methods. Like automatic differentiation **D*** can be applied to relatively large, complex problems but instead of generating exclusively a numerical derivative, as automatic differentiation does, **D*** generates a true symbolic derivative expression; consequently any order of derivative can be easily computed by applying **D*** successively. Unlike forward and reverse techniques the user does not have to make any choices about which algorithm to apply - the symbolic derivative expression is generated completely automatically with no user intervention. **D*** exploits the special nature of the sum of products graph that represents the derivative of a function; our primary contributions are two new greedy algorithms: the first computes a factorization of the derivative graph and the second computes a grouping of common product terms into subexpressions. While not guaranteed to be optimal in practice these two algorithms together produce extremely efficient derivatives. Secondary contributions include symbolically executing the expression graph at compile time to eliminate common subexpressions and embedding the **D*** algorithm in a conventional programming language[2] which is very beneficial from a software engineering perspective.

1.3 Limitations of the current implementation

The current implementation of **D*** inlines all functions and unrolls all loops at expression analysis time. Inlining is not required for the factorization algorithm to work; it is a software engineering choice analogous to the inlining trade-offs made in conventional compilers. This approach exposes maximum opportunities for optimization, and it simplifies the embedding of **D*** in C#. A side effect of this design choice is that the compiled derivative functions may be larger than desired for some applications. It also requires loop iteration bounds to be known at compile time. For our initial set of applications this design trade-off worked quite well but future implementations may perform less inlining to allow for a broader range of application of the algorithm.

The time to compute the symbolic derivative is guaranteed to be polynomial in the size of the expression graph. For an expression graph $f : \mathbb{R}^n \rightarrow \mathbb{R}^m$ with v nodes the worst-case time to compute the symbolic derivative is $O(nmv^3)$. We have never observed this worst-case running time for any of the examples we have tested, although we have seen cubic running times. We believe that an incremental version of the algorithm currently under development will significantly improve on this worst-case time. More details are provided in sections 4.1 and 6.3. In practice, the current algorithm is fast enough to compute the symbolic derivative of expression graphs with hundreds to thousands of nodes in a few seconds and tens of thousands of nodes in an hour or less.

2 Implementation in C#

D* is embedded in C# by overloading all the standard C# arithmetic operators, and by providing special definitions for all the standard mathematical functions, such as sin, cos, etc. The language embedding and differentiation algorithm together total 2800 lines of C# code.

Every **D*** program is a function from $\mathbb{R}^n \rightarrow \mathbb{R}^m$. Executing a **D*** program creates an expression graph representing the function. The graph is made up of nodes, which are instances of classes, and edges, which connect nodes to their children. Creating the graph and its manipulation and simplification is handled by the F class. Variables are instances of the class V. The following snippet of code creates an expression graph for a function, g, which computes a∗b

```
V a = new V("a"), b = new V("b");
F g;

g = a*b;
```

Multidimensional functions are created with the F constructor

```
g = new F(a*b,F.sin(a));
```

Individual range elements can be accessed using an indexer. For the function g in the previous example g[0] is a∗b and g[1] is sin(a).

Every F node has a domain. The domain of a node, *n*, is the set of V's at the leaves of the expression graph rooted at *n*.

The derivative of a function is computed in two steps. First, the overloaded D function

```
Function D(F a, int rangeIndex, params V[] vars)
Function[,] D(F[,] a, params V[] vars)
```

creates a Derivative node which contains a specification of the derivative to be computed. Derivative nodes can be used as arguments to other functions or they can be the root of an expression graph. For example if g = new F(a∗b,F.sin(a)) then D(g,0,a) specifies $\frac{\partial g[0]}{a}$ and D(g,0,a,b) specifies $\frac{\partial^2 g[0]}{\partial^2 ab}$.

After all derivatives are specified the evalDeriv function is called. The arguments to evalDeriv are the roots of the expression graph. evalDeriv creates a list, leafDerivatives, of all Derivative nodes which do not have Derivative nodes below them in the graph. The leafDerivatives list is passed to the global derivative analysis algorithm of section 4.1 and the derivatives are computed. Each Derivative node in leafDerivatives is replaced with its full functional form. This repeats until no unevaluated Derivative nodes are left in the graph.

The derivative of an expression is a new expression whose single range element is the derivative term. Composite derivative entities, such as the Jacobian and Hessian, are created by invoking D() multiple times with the appropriate range and domain terms.

[2] **D*** is currently embedded in C# but can easily be embedded in other languages, such as C++, which support operator overloading.

2.1 Expression optimization

Before an expression is created its hash code is used to see if it already exists. If it does the existing value is used, otherwise a new expression is created. Commutative operators, such as $+$ and $*$ test both orderings of their arguments.

The V arguments to the Derivative constructor are sorted by their unique identifier before computing the hash code. This eliminates redundant computation for derivatives that differ only in the order in which the derivatives are taken so that $\frac{\partial^2 g}{\partial^2 ab}$ and $\frac{\partial^2 g}{\partial^2 ba}$ will hash to the same value.

Algebraic simplification happens only at node construction time which reduces implementation complexity and increases efficiency since the graph is never rewritten. This is much less powerful then the algebraic simplification performed by a program like Mathematica but powerful enough for these important common cases:

$a*1$	$\rightarrow$	a	$a*-1$	$\rightarrow$	$-a$
$a*0$	$\rightarrow$	0	$a\pm 0$	$\rightarrow$	a
a/a	$\rightarrow$	1	$a/-1$	$\rightarrow$	$-a$
$a-a$	$\rightarrow$	0	$f(c_0)$	$\rightarrow$	Constant$(f(c_0))$
c_0*c_1	$\rightarrow$	Constant(c_0*c_1)	$c_0\pm c_1$	$\rightarrow$	Constant$(c_0\pm c_1)$
c_0/c_1	$\rightarrow$	Constant(c_0/c_1)			

Functions can be interpretively evaluated but this is very slow so **D*** has an expression compiler which generates either C# or C++ source and then compiles the source to executable code.

3 Graph structure of the chain rule

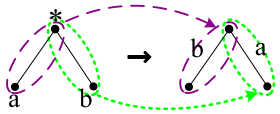
In the first stage of the new differentiation algorithm the derivative expression is factored. To understand the factorization step we must examine the special structure of the graph which results from differentiating a function, the *derivative graph*, and how this relates to the chain rule of differentiation. The conventional form of the chain rule is not very convenient for this purpose so we derive an equivalent form of the chain rule in Appendix A and show how this relates to the derivative graph. This will lead to a simple derivative algorithm which takes worst case time exponential in the number of edges, e, in the original graph. In section 4 we will present a new algorithm which reduces this to polynomial time.

Before we can begin we have to introduce some notation which will minimize clutter in the illustrations. We will use the following notation for derivatives: for $f:\mathbb{R}^n\rightarrow\mathbb{R}^m$, f^i_j is the derivative of the ith range element with respect to the jth domain element. Range and domain indices start at 0. Higher order derivatives are indicated by additional subscript indices. For example

$$f^i_{jk} = \frac{\partial^2 f^i}{\partial f_j \partial f_k} \tag{1}$$

The chain rule can be graphically expressed in a *derivative graph*. The derivative graph of an expression graph has the same structure as the expression graph but the meaning of nodes and edges is different. In a conventional expression graph nodes represent functions and edges represent function composition. In a derivative graph an edge represents the partial derivative of the parent node function with respect to the child node argument. Nodes have no operational function; they serve only to connect edges.

As a simple first example Fig.1 shows the graph representing the function $f=ab$ and its corresponding derivative graph. The edge connecting the $*$ and a symbols in the original function graph corresponds to the edge representing the partial $\frac{\partial ab}{\partial a}=b$ in the derivative graph. Similarly, the $*,b$ edge in the original graph corresponds to the edge $\frac{\partial ab}{\partial b}=a$ in the derivative graph.



Original graph Derivative graph

Figure 1: *The derivative graph of multiplication. The derivative graph has the same structure as its corresponding expression graph but the meaning of edges and nodes is different: edges represent partial derivatives and nodes have no operational function.*

The derivative graph for a more complicated function, $f=\sin(\cos(x))*\cos(\cos(x))$, is shown in Fig.2. The nodes in the original function graph have been given labels $\mathbf{v_i}$ to minimize clutter in the derivative graph:

$$\begin{aligned}\mathbf{v_0} &= \cos(x)\\ \mathbf{v_1} &= \sin(\cos(x)) = \sin(\mathbf{v_0})\\ \mathbf{v_2} &= \cos(\cos(x)) = \cos(\mathbf{v_0})\end{aligned}$$

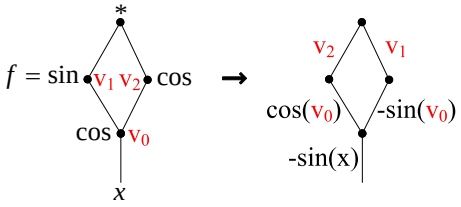


Figure 2: *The derivative graph of an expression*

Given some $f:\mathbb{R}^n\rightarrow\mathbb{R}^m$ we can use the derivative graph of f to compute the derivative f^i_j as follows. Find all paths from node i to node j. For each path compute the product of all the partial derivatives that occur along that path; f^i_j is equal to the sum of these path products. This is precisely equivalent to the alternative form of the chain rule derived in Appendix A. In the worst case, the number of paths is exponential in the number of edges in the graph so this algorithm takes exponential time, and produces an expression whose size is exponential in the number of edges in the graph.

If we apply this differentiation algorithm to compute f^0_0 we get the result shown in Fig.3. For each path from the root we compute the product of all the edge terms along the path, then sum the path products:

$$\begin{aligned}f^0_0 &= \mathbf{v_2}\cos(\mathbf{v_0})(\text{-}\sin(x)) + \mathbf{v_1}(\text{-}\sin(\mathbf{v_0}))(\text{-}\sin(x))\\ &= \cos(\cos(x))\cos(\cos(x))(\text{-}\sin(x)) + \sin(\cos(x))(\text{-}\sin(\cos(x)))(\text{-}\sin(x))\end{aligned}$$

For $f:\mathbb{R}^n\rightarrow\mathbb{R}^m$ the path product sum may have redundant computations of two forms: common factors and common product subsequences. Both will be discussed in more detail in section 4 but we can get an intuitive grasp of common factor redundancy by looking at the simple example of Fig.4. Each branching of the graph, either upward or downward, corresponds to a factorization of the expression. All product paths that pass through the node marked B will

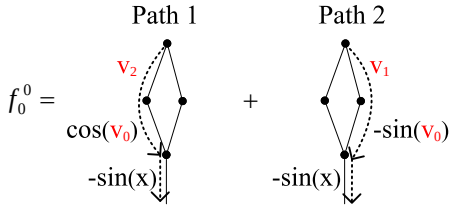


Figure 3: *The sum of all path products equals the derivative*

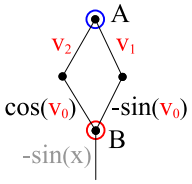


Figure 4: *Each branching in the derivative graph corresponds to a factorization of the derivative. There is a branch at node A and at node B.*

include $-\sin(x)$ as a factor. If we collapse the two product paths into a single edge that groups the terms which share $-\sin(x)$ as a factor then we get the graph of Fig.5. This is mathematically the same as summing the product paths of the graph of Fig.3 but now there is a single product path where there used to be two.

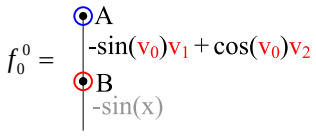


Figure 5: *Factoring out the terms which share $-sin(x)$ reduces the number of paths in the graph of Fig.3 from two to one.*

4 Factoring the derivative graph

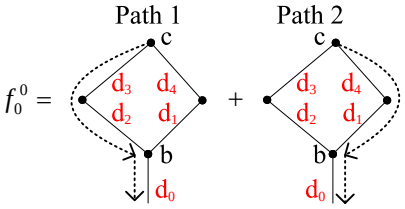
Since the derivative of $f : \mathbb{R}^n \to \mathbb{R}^m$ is just the derivative of each of its nm $\mathbb{R}^1 \to \mathbb{R}^1$ constituent functions we'll begin by developing an algorithm for factoring the derivative of $\mathbb{R}^1 \to \mathbb{R}^1$ functions, and then generalize to the more complicated case of $f : \mathbb{R}^n \to \mathbb{R}^m$ in section 4.1.

The derivative graph, f_0^0, of an $\mathbb{R}^1 \to \mathbb{R}^1$ function has one root and one leaf; there is a potential factorizaton of f_0^0 when two or more paths must pass through the same node on the way to the root or to the leaf. As an example, in Fig. 6 all paths from c to the leaf must pass through node b and therefore must include the common factor d_0.

Factoring is closely related to a graph property called *dominance*. If a node b is on every path from node c to the root then b *dominates* c (b **dom** c). If b is on every path from c to the leaf then b *postdominates* c (b **pdom** c). Looking again at Fig. 6 we can see that node b postdominates node c and so all paths from c to the leaf must include the common term d_0, which can be factored out.

A slightly more complicated example is shown in Fig.7. Here node 0 postdominates nodes 1, 2 and 3 (0 **pdom** $\{1,2,3\}$) but node 2 does not dominate node 0. Node 3 dominates nodes 0,1, and 2 (3 **dom** $\{0,1,2\}$).

An efficient, simple algorithm (roughly 40 lines of code) for finding



$$f_0^0 = d_0d_2d_3 + d_0d_1d_4 = d_0(d_2d_3 + d_1d_4)$$

Figure 6: *Relationship between factoring and dominance. Node b postdominates node c so all paths from c to the leaf must include the common factor d_0.*

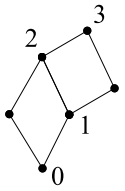


Figure 7: *Dominance and post dominance relationships. Node 0 postdominates nodes 1, 2 and 3 (0* **pdom** $\{1,2,3\}$*) but node 2 does not dominate 0 or 1. Node 3 dominates nodes 0,1, and 2, (3* **dom** $\{0,1,2\}$*) but node 1 does not postdominate 3 or 2.*

the dominators or postdominators of a graph is described in [Cooper et al. 2001]. For a DAG (directed acyclic graph) this takes worst case time $O(n^2)$ where n is the number of nodes in the graph. Linear time algorithms exist but in practice these algorithms are slower until n becomes quite large.

Factorable subgraphs are defined by a dominator or postdominator node at a branch in the graph. If a dominator node b has more than one child, or if a post-dominator node b has more than one parent, then b is a *factor* node. If c is dominated by a factor node b and has more than one parent, or c is postdominated by b and has more than one child, then c is a *factor base* of b. A factor subgraph, $[b,c]$ consists of a factor node b, a factor base c of b, and those nodes on any path from b to c.

For example, the factor nodes in Fig.7 are 0 and 3. The factor subgraphs of node 3, highlighted in blue in Fig.8, are $[3,1],[3,0]$. Node 2 is not a factor node because the sole node dominated by 2 has only one parent and no node is postdominated by 2. Node 1 is not a factor node because no nodes are dominated or postdominated by 1.

The factor subgraphs of node 0 are $[0,2],[0,3]$. Notice that $[3,0]$ and $[0,3]$ are the same graph. This is true in general, i.e., $[a,b] = [b,a]$ if both exist. However, you can see that $[1,3]$ is not a factor subgraph even though $[3,1]$ is. This is because 1 does not postdominate 3 in the graph of Fig.7. In general, the existence of $[a,b]$ does not imply the existence of $[b,a]$.

We can factor the graph by using the factor subgraphs and the dominance relations for the graph. We'll assume that the graph has been DFS numbered from the root, so the parents of node b will always have a higher number than b. Each edge, e, in the graph has nodes $e.1, e.2$ with the node number of $e.2$ greater than the node number of $e.1$, i.e., $e.2$ will always be higher in the graph. The following algorithm computes which edges to delete from the original graph and which edges to add to a new factored edge:

given: a list L of factor subgraphs [X,Y]
and a graph G

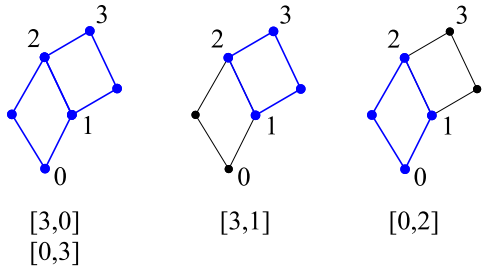


Figure 8: *Factor subgraphs of the graph of Fig.7.*

```
S = empty subgraph edge
for each factor subgraph [A,B] in L{
  E = all edges which lie on a path from B to A

  for each edge e in E{
    if(isDominator(A){ //dominatorTest
      if(B pdom e.1){
      delete e from G
      }
    }
    else{ //postDominatorTest
      if(B dom e.2){
        delete e from G
      }
    }
    add e to S
  }
  add subgraph edge S to G, connecting node A to node B
  if any [X,Y] in L no longer exists delete [X,Y] from L
}
```

The *subgraph* edges that are added to the original graph are edges which themselves contain subgraphs. The subgraphs contained in subgraph edges are completely isolated from the rest of the original graph, and from the point of view of further edge processing behave as though they were a single edge[3].

The correctness of this algorithm is easily verified for a **dom** b; the proof for b **pdom** a is similar. There are two classes of paths: those which pass through both a and b: $root\cdots a\cdots b\cdots leaf$ and those which pass through a but not b: $root\cdots a\cdots leaf$[4]. Starting with the first class: if we remove all edges $e\in[a,b]$ from the original graph and replace them by a single edge whose value is the sum of all path products from a to b, then the value of the sum of all path products over the paths $root\cdots a\cdots b\cdots leaf$ will be unchanged. Computation will be reduced because of the factorization but algebraically the two sums will be identical. For example, in Fig.9 the factor subgraph $[3,1]$ has been replaced by a single edge from node 3 to node 1 and the paths which precede a and follow b in $root\cdots a\cdots b\cdots leaf$ have been factored out.

Edges $e\in[a,b]$ which belong to the second class of paths, $root\cdots a\cdots leaf$, cannot be deleted because this would change the sum of products over all paths. In Fig.9 if edge d_1 is removed then the product $d_0d_1d_5d_6$ will be destroyed. All such e have the property that b **pdom** $e.1$ is not true, i.e., there is a path through e to $leaf$ which does not pass through b. In Fig.9 b **pdom** $d_3.1$ so edge d_3 can be removed from the graph but b **pdom** $d_1.1$ is not true so edge d_1 cannot be removed from the graph.

Factorization does not change the value of the sum of products expression which represents the derivative so factor subgraphs can be factored in any order [5].

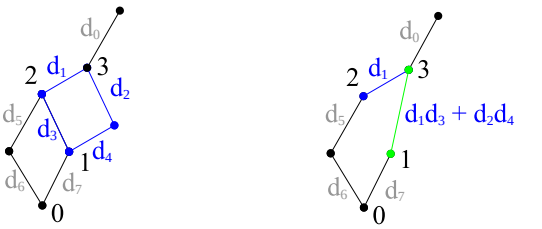


Figure 9: *The factorization rule does not change the value of the sum of products over all paths.*

In Fig. 10 the factoring algorithm is applied to a postdominator case. Factor node 0 is a postdominator node; the red edge labeled **d_4** does not satisfy the postDominatorTest so it is not deleted from the original graph. The three blue edges labeled **d_3**,**d_5**,**d_6** satisfy the test so they are deleted. Since factor subgraph $[3,1]$ no longer exists in the graph, it is deleted from the list of factor subgraphs and not considered further.

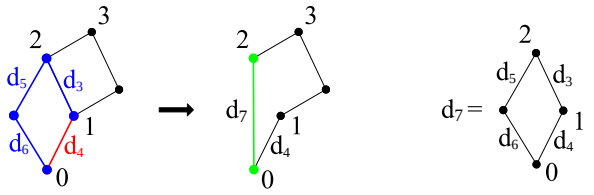


Figure 10: *Factor subgraph* $[0,2]$*, highlighted in blue and red in the leftmost graph, is factored out of the graph and replaced with an equivalent subgraph edge* d_7.

The factor subgraphs for the new graph, shown in Fig.11 on the left hand side, are $[3,0],[0,3]$. We choose $[0,3]$ arbitrarily. All edges satisfy the postDominatorTest so the final graph, in Fig.11 on the far right hand side, has the single subgraph edge d_8.

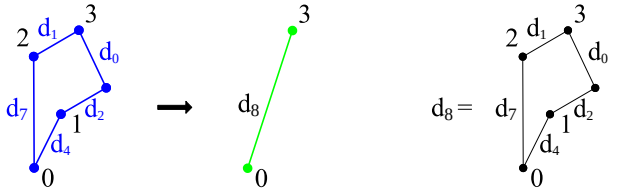


Figure 11: *Factor subgraph* $[0,3]$*, highlighted in blue in the leftmost graph, is factored out of the graph and replaced with an equivalent subgraph edge* d_8.

Alternatively we could have factored $[3,1]$ first as shown in Fig.12. Factor node 3 is a dominator node; the red edge labeled **d_1** does

[3] Except for the final evaluation step when the edge subgraphs are recursively visited to find the value of the factored derivative graph.

[4] Paths which pass through b but not a cannot occur because a **dom** b.

[5] However, for $f:\mathbb{R}^n\to\mathbb{R}^m$ different orders may lead to solutions with very different computational efficiency.

not satisfy the dominatorTest so it is not deleted from the original graph. The three blue edges labeled $\mathbf{d_0}$,$\mathbf{d_2}$,$\mathbf{d_3}$ satisfy the test so they are deleted. Since factor subgraph $[0,2]$ no longer exists in the graph, it is deleted from the list of factor subgraphs and not considered further.

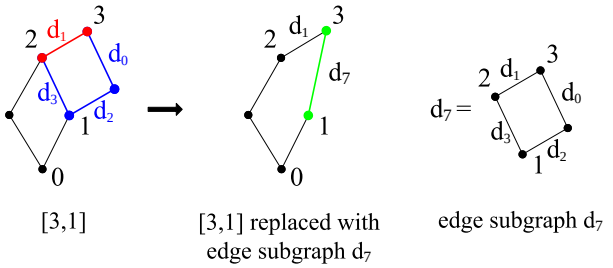


Figure 12: *Factor subgraph $[3,1]$ is factored out of the graph and replaced with an equivalent subgraph edge d_7.*

The factor subgraphs for this new graph are $[3,0],[0,3]$. We choose $[3,0]$ arbitrarily. All edges satisfy the dominatorTest so we get the result of Fig.13.

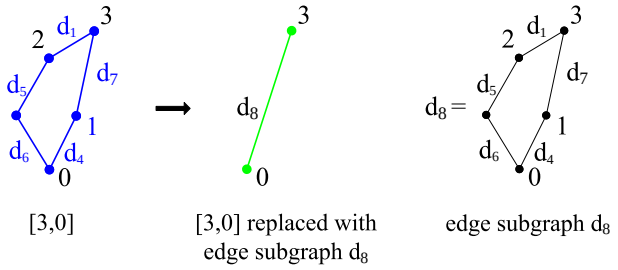


Figure 13: *Factor subgraph $[3,0]$, highlighted in blue in the left-most graph, is factored out of the graph and replaced with an equivalent subgraph edge d_8.*

To evaluate the factored derivative we compute the sum of products along all product paths, recursively substituting in subgraphs when necessary. For the factorization of Figs.10, 11 we get

$$\begin{aligned} f_0^0 &= d_8 && (2)\\ &= d_1d_7 + d_0d_2d_4 && (3)\\ &= d_1(d_5d_6 + d_3d_4) + d_0d_2d_4 && (4)\end{aligned}$$

and for the factorization of Figs.12, 13 we get

$$\begin{aligned} f_0^0 &= d_8 && (5)\\ &= d_1d_5d_6 + d_7d_4 && (6)\\ &= d_1d_5d_6 + (d_1d_3 + d_0d_2)d_4 && (7)\end{aligned}$$

The two factorizations of eq.4 and eq.7 are trivially different; they have the same operations count.

For $f:\mathbb{R}^1 \to \mathbb{R}^1$ this algorithm is all we need. For $f:\mathbb{R}^n \to \mathbb{R}^m$ we will need the more sophisticated algorithm of section 4.1.

4.1 Factoring $\mathbb{R}^n \to \mathbb{R}^m$ functions

Two complications arise in factoring $f:\mathbb{R}^n \to \mathbb{R}^m$ which did not arise in the $f:\mathbb{R}^1 \to \mathbb{R}^1$ case. The first is that the order in which the factor subgraphs are factored can make an enormous difference in computational efficiency. The second is that after factorization different derivatives may share partial product subsequences so it is desirable to find product subsequences that are most widely shared. The order of factorization will be dealt with in this section and the product subsequence issue will be dealt with in the next.

The derivative of f is just the derivative of each of its nm $\mathbb{R}^1 \to \mathbb{R}^1$ constituent functions. These nm $\mathbb{R}^1 \to \mathbb{R}^1$ derivative graphs will, in general, have a non empty intersection which represents redundant computation. An example of this is shown in Fig.14. Here the

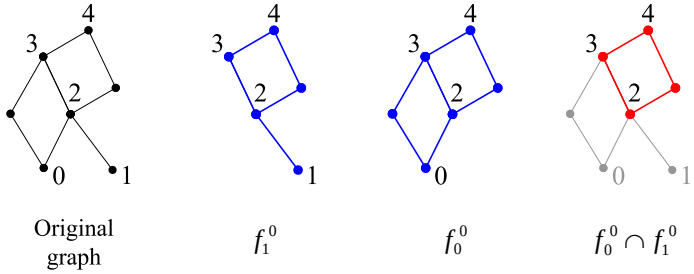


Figure 14: *The derivatives f_0^0 and f_1^0 intersect only in the red highlighted subgraph.*

derivatives f_0^0 and f_1^0 intersect in the red highlighted region, which is a common factor subgraph of f_0^0 and f_1^0. If we choose to factor $[0,3]$ from f_0^0 then we get Fig.15 where $f_0^0 \cap f_1^0$ does not contain a factor subgraph of either derivative. If instead we factor $[4,2]$ from

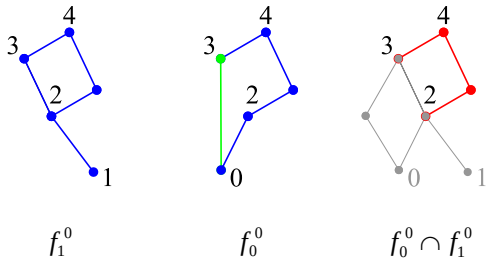


Figure 15: *Factor $[0,3]$ from f_0^0. $f_0^0 \cap f_1^0$, highlighted in red, does not contain a factor subgraph of either derivative.*

both f_0^0 and f_1^0 then we get Fig.16. $f_0^0 \cap f_1^0$ contains the common subgraph edge 4,2.

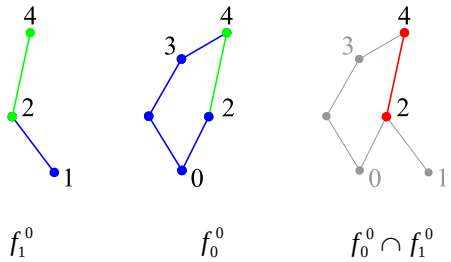


Figure 16: *Factor $[4,2]$ from both f_0^0 and f_1^0. $f_0^0 \cap f_1^0$ contains the common subgraph edge 4,2.*

The computation required for f_0^0 is independent of whether $[0,3]$ or $[4,2]$ is factored first. But the computation required to compute both f_0^0 and f_1^0 is significantly less if $[4,2]$ is factored first because we

can reuse the $[4,2]$ factor subgraph expression in the factorization of f_1^0.

The solution to the problem of common factor subgraphs is to count the number of times each factor subgraph $[i,j]$ appears in the nm derivative graphs. The factor subgraph which appears most often is factored first. If factor subgraph $[k,l]$ disappears in some derivative graphs as a result of factorization then the count of $[k,l]$ is decremented. To determine if factorization has eliminated $[k,l]$ from some derivative graph f_j^i it is only necessary to count the children of a dominator node or the parents of a postdominator node. If either is one the factor subgraph no longer exists. The counts of the $[k,l]$ are efficiently updated during factorization by observing if either node of a deleted edge is either a factor or factor base node. Ranking of the $[k,l]$ can be done efficiently with a priority queue. The complete factorization algorithm is:

```
factorSubgraphs(function F){
  hash table Counts: counts of [k,l]
  list Altered: [k,l] whose counts have changed due
        to factorization
  priority queue Largest: sorted by factor subgraph count

  foreach(derivative graph Fij in F){
    compute factor subgraphs of Fij;

    foreach(factor subgraph [k,l] in Fij){
      if(Counts[[k,l]] == null){
        Counts[[k,l]] = [k,l];
        }
      else{Counts[[k,l]].count += 1;
    }
    foreach([k,l] in Counts){Largest.insert([k,l]);}
  }

  while(Largest not empty){
    maxSubgraph = Largest.max
    foreach(Fij in which maxSubgraph occurs){
      Altered.Add(Fij.factor(maxSubgraph))
      compute factor subgraphs of Fij;
    }
    foreach([k,l] in Altered){Largest.delete([k,l])}
    foreach([k,l] in Altered){Largest.insert([k,l])}
  }
}
```

For $f:\mathbb{R}^n \rightarrow \mathbb{R}^m$ with v nodes there are nm F_{ij} each of which can have at most v factor subgraphs. At most v iterations will be required to factor all of these subgraphs. Re-computing the factor subgraphs takes worst-case time $O(v^2)$; this is done at each iteration. Multiplying these terms together gives a worst-case time of $O(nmv^3)$. We have never observed this worst-case running time for any of the examples we have tested and in practice the algorithm is fast enough to differentiate expression graphs with tens of thousands of nodes.

In the current algorithm any time a factor subgraph is factored all of the factor subgraphs of the F_{ij} are recomputed which requires running the dominator algorithm on all the nodes in F_{ij}. This is very wasteful since the vast majority of the dominance relationships will remain unchanged after factoring any given factor subgraph. It would be far more efficient to incrementally update just those dominance relations which might have been changed by the factorization step - we are currently implementing such an algorithm. We discuss the consequences of this inefficiency further in section 6.3.

4.2 Computing Common Subproducts

After the graph has been completely factored there is no branching, i.e., for each F_{ij} there is a single path from node i to node j. Fig.17 shows the derivative graph of an $\mathbb{R}^3 \rightarrow \mathbb{R}^2$ function. Each of the nm derivative functions is completely factored giving six path products:

$$f_0^0 = d_1d_2d_4,\ f_1^0 = d_1d_2d_5,\ f_2^0 = d_1d_2d_3$$
$$f_0^1 = d_0d_2d_4,\ f_1^1 = d_0d_2d_5,\ f_2^1 = d_0d_2d_3$$

The subproducts d_1d_2 and d_0d_2 can each be used in 3 path products, whereas the subproducts d_2d_4, d_2d_5, and d_2d_3 can each only be used in 2 path products. If we compute and reuse the subroducts d_1d_2 and d_0d_2 we can compute all six path products with only $2+2*3=8$ multiplies. If we compute and reuse the products d_2d_4, d_2d_5, and d_2d_3 it will take $3+3*2=9$ multiplies. In this simple example it's easy to determine the best choice but it becomes quite difficult for more complex graphs.

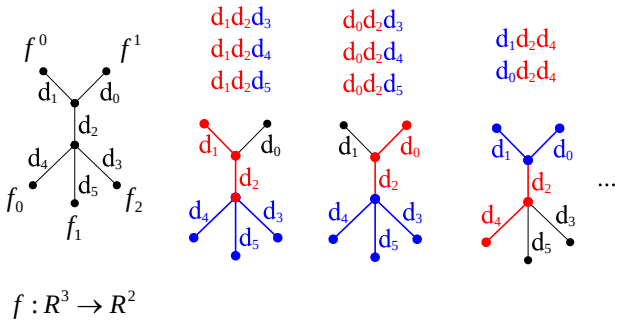


Figure 17: *The $\mathbb{R}^3 \rightarrow \mathbb{R}^2$ function on the left has 6 derivatives; all of the derivatives are completely factored. What is the best way to form the path products? The subproducts d_1d_2 and d_0d_2 can each be used in 3 path products. The subproducts d_2d_4,d_2d_5, and d_2d_3 can each only be used in 2 path products.*

The solution to the problem of common subproducts is to compute the number of product paths that pass through each subproduct and then form the subproduct with the highest path count. This is performed in two stages. First the path counts of pairs of edges which occur in sequence along the path are computed. Then the highest count pair is merged into an EdgePair which is inserted into all paths of all f_j^i derivative graphs which have the pair. The counts of existing edge pairs are updated. This takes time $O(1)$ per edge pair that is updated. This process is continued until all paths in all f_j^i are one edge long. Each edge pair may itself contain an edge pair and edges may contain subgraphs so the final evaluation of the derivative requires recursively expanding each of these data types as it is encountered.

The following pseudocode assumes that each f_j^i is stored as a linked list of edges and that a hashtable or similar data structure is employed so that any edge can be found in $O(1)$ time. To simplify the presentation all the (many) tests for special cases such as null values, no previous or next edges, etc. have been eliminated. When the program terminates every f_j^i will consist of a set of paths each of which will be a sequence which will contain one, and only one, of the following types: edges, edge subgraphs, and edge pairs.

```
optimumSubproducts(graph G){
  //count of paths edge e occurs on
  hash table Counts
  priority queue Largest: sorted by edge path count
  foreach(derivative graph Fij in G){
    foreach(edge eSub in Fij){
      if(eSub.isEdgeSubgraph){
        foreach(edge e, e.next in eSub){
          temp = new EdgePair(ei, e.next)
```

```
        Counts[temp].pathCount += 1
      }
    }
    else{
      temp = new EdgePair(ei, e.next)
      Counts[temp].pathCount += 1
    }
  }
 }

 foreach(EdgePair e in Counts){Largest.insert(e)}

 while(Largest not empty){
   maxProduct = Largest.max
   foreach(Fij which has maxProduct){
     ei = Fij.find(maxProduct.edge1)
     eiNext = ei.next
     eiPrev = ei.previous
     eiNext2 = eiNext.next
     Fij.delete(ei)
     Fij.delete(eiNext)
     oldPair = new EdgePair(ei,eiNext)
     eiPrev.insertNext(oldPair)
     prevPair = new EdgePair(eiPrev,ei)
     nextPair = new EdgePair(eiNext,eiNext2)
     updateCounts(oldPair, prevPair,nextPair)
   }
 }
}

updateCounts(oldPair, prevPair, nextPair){
  Counts.delete(oldPair)
  Largest.delete(oldPair)
  Counts[prevPair] −= 1
  Counts[nextPair] −= 1
  Largest.delete(prevPair)
  Largest.delete(nextPair)
  Largest.insert(prevPair)
  Largest.insert(nextPair)
}
```

5 Examples

For our test set we have chosen problems which arise in many areas of graphics. For two of the examples, inverse dynamics and space time optimization, it has taken years to find efficient derivatives.

The functions and their representation in **D*** are described in this section. In section 6 the speed and operation count of the solutions generated by **D*** are compared with those from the automatic differentiation program CppAD and the symbolic math program Mathematica, respectively.

5.1 Spherical harmonics

Spherical harmonics are used in many algorithms to approximate global illumination functions. For example, in the PRT algorithm [Sloan et al. 2005] the smallest possible set of basis functions is sought which approximates a given illumination function. A gradient based optimization routine is used to minimize the number of spherical harmonic coefficients. Computing the gradient is complicated by the fact that the spherical harmonics are most easily defined in a recursive fashion and it is not obvious how to differentiate these recursive equations directly.

The spherical harmonic functions are defined by the following set of 4 recursive equations: Legendre polynomials, P, divided by $\sqrt{(1-z^2)^m}, 0 \le l < n, m \le l$, functions of z

$$\begin{aligned} P(0,0) &= 1 \\ P(m,m) &= (1-2m)P(m-1,m-1) \\ P(m+1,m) &= (2m+1)zP(m,m) \\ P(l,m) &= \frac{(2l-1)zP(l-1,m)-(l+m-1)P(l-2,m)}{(l-m)} \end{aligned}$$

sin/cos, written S,C, multiplied by $\sqrt{(1-z^2)^m}, 0 \le m < n$, functions of x,y

$$\begin{aligned} S(0) &= 0 \\ C(0) &= 1 \\ S(m) &= xC(m-1)-yS(m-1) \\ C(m) &= xS(m-1)+yC(m-1) \end{aligned}$$

constants, N, $0 \le l < n, m \le l$

$$\begin{aligned} N(l,m) &= \sqrt{(2l+1)/(4\pi)}m = 0 \\ N(l,m) &= \sqrt{\tfrac{(2l+1)}{(2\pi)}\tfrac{(l-|m|)!}{(l+|m|)!}}m > 0 \end{aligned}$$

and the spherical harmonic basis functions, Y, $0 \le l < n, |m| \le l$

$$\begin{aligned} Y(l,m) &= N(l,|m|)P(l,|m|)S(|m|)m < 0 \\ Y(l,m) &= N(l,|m|)P(l,|m|)C(|m|)m \ge 0 \end{aligned}$$

The order of the spherical harmonic function is specified by the first argument, l, to the function $Y(l,m)$. At each order n there are $2n+1$ basis functions. The total number of basis functions up to order n is

$$\sum_{i=1}^{n} 2i+1 = n^2 \tag{8}$$

$Y(l,m)$ is an $\mathbb{R}^3 \to \mathbb{R}^{n^2}$ function so there are $3n^2$ derivative terms in the gradient of Y.

The **D*** functions are essentially identical to the recursive mathematical equations and require only 28 lines of code including the code necessary to specify the derivatives to be computed.

```
F SHDerivatives(int maxL, double x, double y, double z){
  List harmonics = new List();

  for (int l = 0; l < maxL; l++) {
    for (int m = −l; m <= l; m++) { harmonics.Add(Y(l,m,x,y,z)); }
  }
  F[] dY = new F[harmonics.Count ∗ 3];
  for (int i = 0; i < dY.GetLength(0) / 3; i++) {
    dY[i ∗ 3] = D((F)harmonics[i],0,x);
    dY[i ∗ 3 + 1] = D((F)harmonics[i],0,y);
    dY[i ∗ 3 + 2] = D((F)harmonics[i],0,z);
  }

  return evalDeriv(dY);
}

F P(int l, int m, Var z){
  if(l==0 && m==0){return 1.0;}
  if(l==m){return (1−2∗m)∗P(m−1,m−1,z);}
  if(l==m+1){return (2∗m + 1)∗z∗P(m,m,z);}
  return(((2∗l −1)/(l−m))∗z∗P(l−1,m,z) − ((l+m−1)/(l−m))∗P(l−2,m,z));
}

F S(int m, Var x, Var y){
  if(m==0){return 0;}
  else{return x∗C(m−1,x,y) − y∗S(m−1,x,y);}
}

F C(int m, Var x, Var y){
  if(m==0){return 1;}
```

```
  else{return x*S(m-1,x,y) + y*C(m-1,x,y);}
}

F N(int l, int m){
  int absM = Math.Abs(m);
  if(m==0){return Math.Sqrt((2*l+1)/(4*Math.PI));}
  else{return Math.Sqrt((2*l+1)/(2*Math.PI)*(factorial(l-absM)/factorial(l+absM)))
       ;}
}

F Y(int l, int m, Var x, Var y, Var z){
  int absM = Math.Abs(m);
  if(m<0){return N(l,absM)*P(l,absM,z)*S(absM,x,y);}
  else{return N(l,absM)*P(l,absM,z)*C(absM,x,y);}
}
```

5.2 Inverse Dynamics

Inverse dynamics is the problem of solving for the forces and torques needed to move a mechanism in a specified way. The inverse dynamics problem itself is not of much interest for graphics applications but it is an integral part of many space-time optimization algorithms, which will be discussed in section 5.3.

This example is restricted to the class of mechanisms with a single linear sequence of connected links with rotary actuators. The extension to tree structured manipulators with both rotary and linear actuators is not difficult.

Each link, l_i, in the manipulator has an associated 4x4 homogenous transformation matrix, $\mathbf{A}_i$, which is a function of the joint angle, q_i. $\mathbf{A}_i$ relates the l_i coordinate frame to the coordinate frame preceding it in the chain. The transformation from l_i coordinates to the global coordinate frame is $\mathbf{W}_i$:

$$\mathbf{W}_i = \mathbf{A}_0\mathbf{A}_1\ldots\mathbf{A}_i = \mathbf{W}_{i-1}\mathbf{A}_i \tag{9}$$

In the Langrangian formulation the torque for a given joint, τ_i, is given by

$$\tau_i = \frac{d}{dt}\frac{\partial \mathbf{L}}{\partial \dot{q}_i} - \frac{\partial \mathbf{L}}{\partial q_i} \tag{10}$$

where $\mathbf{L}$ is the Langrangian

$$\mathbf{L} = \Phi - P \tag{11}$$

and Φ is the kinetic and $\mathbf{P}$ the potential energy of the system. Φ is

$$\Phi = \frac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{n}\sum_{k=1}^{n} tr\left[\frac{\partial \mathbf{W}_i}{\partial q_j}\mathbf{J}_i\frac{\partial \mathbf{W}_i^T}{\partial q_k}\dot{q}_j\dot{q}_k\right] \tag{12}$$

where tr is the matrix trace operator (the sum of the diagonal elements), and $\mathbf{J}_i$ is the 4x4 Euler inertia tensor for link i. The potential energy, $\mathbf{P}$, is

$$\mathbf{P} = \text{constant} - \sum_{j=1}^{n} m_j\mathbf{g}^T\mathbf{W}_j\mathbf{r}_j \tag{13}$$

where m_j is the mass of link j, $\mathbf{g}^T$ is the gravity vector, and $\mathbf{r}_j$ is the vector from the origin of the jth coordinate frame to the center of mass of link j.

Waters [Waters 1979] noticed that eq.10 could be written in the following form:

$$\tau_i = \sum_{j=i}^{n}\left[tr\left(\frac{\partial \mathbf{W}_j}{\partial q_i}\mathbf{J}_j\ddot{\mathbf{W}}_j^T\right) - m_j\mathbf{g}^T\frac{\partial \mathbf{W}_j}{\partial q_i}\mathbf{r}_j\right], i = 1,\cdots,n \tag{14}$$

We can rewrite the triple matrix product of eq.14 as

$$\sum_{j=i}^{n} tr\left(\frac{\partial \mathbf{W}_j}{\partial q_i}\mathbf{J}_j\ddot{\mathbf{W}}_j^T\right) = \sum_{j=i}^{n} tr\left(\mathbf{W}_{i-1}\frac{\partial \mathbf{A}_i}{\partial q_i}\mathbf{A}_{i+1}\mathbf{A}_{i+2} + \ldots\mathbf{A}_j\mathbf{J}_j\ddot{\mathbf{W}}_j^T\right) \tag{15}$$

$$= tr\left[\mathbf{W}_{i-1}\frac{\partial \mathbf{A}_i}{\partial q_i}\left(\mathbf{J}_i\ddot{\mathbf{W}}_i^T + \mathbf{A}_{i+1}(\mathbf{J}_{i+1}\ddot{\mathbf{W}}_{i+1}^T + \mathbf{A}_{i+2}(\mathbf{J}_{i+2}\ddot{\mathbf{W}}_{i+2}^T\ldots\mathbf{A}_n\mathbf{J}_n\ddot{\mathbf{W}}_n^T))\right)\right] \tag{16}$$

Note that this does not change the number of operations required.

The **D*** program for solving the inverse dynamics problem is shown below[6].

```
F computeTorque(F[][,] A, F[][] r, F[] q){
  F[] torque = new F[numLinks];
  F[][,] W = new F[numLinks][4,4];

  for (int i = 0; i < numLinks; i++) {
    DAi_qi[i] = D(A[i],q[i]);
    Wdd[i] = J*transpose(D(W[i],t,t));
  }

  for (int i = 1; i < numLinks; i++) {W[i] = W[i - 1]*A[i];}

  torque[0] = trace(DAi_qi[0]*sumProd(0,A,Wdd))
        - m * dot(g,DAi_qi[0]*sumProd(0,A,r));
  for (int i = 1; i <numLinks; i++) {
    torque[i] = trace(W[i - 1]*DAi_qi[i]*sumProd(i,A,Wdd))
          - m * dot(g,W[i - 1]*DAi_qi[i]*sumProd(i,A,r));
  }
  return evalDeriv(torque);
}

F[,] sumProd(int start,F[][,] A,F[][,] b) {
  F[,] sum = b[b.GetLength(0) - 1]; //set sum to last entry in b

  for (int i = A.GetLength(0) - 1; i >= start + 1; i--) {
  sum = b[i - 1] + A[i]*sum;
  }
  return sum;
}
```

5.3 Space Time Optimization

Space time optimization is a sophisticated technique for automatically generating animations of complex articulated figures. For a representative, but by no means complete, sampling of these types of algorithms see [van de Panne et al. 2000; Witkin and Kass 1988; Liu et al. 1994; Fang and Pollard 2003; Liu et al. 2005].

In space-time optimization the animation problem is cast as an optimization problem with an objective function

$$F = \int_0^{t_f} f(q_0(t), q_1(t), \ldots, q_n(t)) \tag{17}$$

to be minimized, where the q_i are the generalized coordinates of the system, and a set of constraints

$$c(q_0(t), q_1(t), \ldots, q_n(t)) = 0 \tag{18}$$

to be satisfied. The $q_i(t)$ are defined in terms of basis functions which have enough degrees of freedom to contain the desired motion but which allow for tractable numerical solutions. Piecewise polynomial splines and multiresolution splines, among others, have been used with considerable success.

[6] For technical reasons which don't concern us here it is not possible in C# to overload the + and × operators for array multiplication; however, we have used this notation to make the code more readable.

To avoid unnecessary complexity in the example we will assume that our physical system is an n degree of freedom manipulator with rotary joints, that each q_i is defined by a single cubic polynomial

$$q_i(t) = a_{i,0}t^3 + a_{i,1}t^2 + a_{i,2}t + a_{i,3} \tag{19}$$

and that we wish to minimize the sum of the joint torques:

$$f(t) = \sum_i \tau_i^2(t) \tag{20}$$

The integral of eq.17 is typically evaluated with numerical quadrature

$$\tilde{F} = \sum_i w_i f(t_i) \approx \int_0^{t_f} f(q_0(t), q_1(t), \ldots, q_n(t)) \tag{21}$$

Gradient based optimization is frequently used to compute a local minimum of $\tilde{F}$: this requires computing the derivative of f with respect to the free parameters of the basis functions

$$D(f) = \left(\frac{\partial f}{\partial a_{0,0}}, \frac{\partial f}{\partial a_{0,1}}, \frac{\partial f}{\partial a_{0,2}}, \frac{\partial f}{\partial a_{0,3}}, \ldots, \frac{\partial f}{\partial a_{n,0}}, \frac{\partial f}{\partial a_{n,1}}, \frac{\partial f}{\partial a_{n,2}}, \frac{\partial f}{\partial a_{n,3}} \right) \tag{22}$$

Computing the gradient of the space-time objective function is straightforward using **D***. Assuming that the array tau[] contains the torques, τ_i, computed with eqs.(14,16), and that indVars is an array containing the variables we wish to differentiate with respect to, we can compute the gradient with the following code:

```
F f = 0;
for(int i=0;i<tau.rangeDim;i++) {f += tau[i]*tau[i]}
F Df = gradient(f,indVars);
```

where the gradient function is:

```
F gradient(F f, F[] indVars){
  int n = indVars.GetLength(0);
  F[] derivs = new F[n];

  for (int i = 0; i < n; i++) {derivs[i] = D(f,0,indVars[i]);}
  return evalDeriv(derivs);
}
```

6 Results

D* was tested against Mathematica and the automatic differentiation program CppAD. Mathematica was chosen because it is the most widely used symbolic math program and its performance can be expected to be competitive with that of similar programs. CppAD was chosen because it has a relatively straightforward API, it supports C++, the language most graphics applications are written in, and CppAD benchmark results are competitive with other C++ automatic differentiation programs .

For comparison with Mathematica we compute the ratio of the number of operations in the Mathematica derivative expression to the number of operations in the **D*** derivative. To simplify the Mathematica expressions we used the most effective of FullSimplify or Simplify if this took less than one hour to complete; otherwise we used no simplification.

For comparison with CppAD we compute the ratio of the number of **D*** derivative evaluations per second to that of CppAD. Whether comparing to Mathematica or CppAD ratios greater than one indicate that the **D*** derivative is faster. All timings were performed on a 3.4 GHz Pentium 4 processor with 3 GB of RAM. The **D*** and CppAD C++ derivative evaluation programs were compiled with Microsoft Visual C++ 2005 with compiler flag /O2.

Reasonable, but not extraordinary, efforts were made to manually optimize both the Mathematica and CppAD results. For Mathematica we applied substitution rules for easily spotted common subexpressions. For CppAD we chose the forward or reverse method, whichever was the most efficient, and we wrote a special implementation of the trace operator which evaluated only the diagonal terms of the triple matrix product of eq.16 which reduced the number of operations by a factor of four[7]. No effort was made to optimize the **D*** programs or the results of the **D*** symbolic differentiation.

6.1 Spherical harmonics

Derivatives of spherical harmonics (sec.5.1) with values of L from 5 to 20 were computed. Table 1 shows the results for **D*** versus CppAD. For the smallest problem size, $L = 5$, **D*** is 243 times faster while for $L = 20$ **D*** is more than 222,000 times faster. This enormous difference in efficiency results from the fact that CppAD continually reevaluates redundant parts of the recursive expression, which is unavoidable using automatic differentiation techniques.

order, L	**5**	**10**	**15**	**20**
D*	**6,622,516**	**1,117,318**	**468,384**	**222,024**
CppAD	27,239	886	29	1
ratio	**243**	**1261**	**16,151**	**222,024**
D* symbolic time (secs.)	**.02**	**.55**	**4.9**	**53**

Table 1: *Spherical harmonics: CppAD vs. **D***. Number of derivative evaluations per second. Ratio is the number of **D*** evaluations per second divided by the number of CppAD evaluations per second. The last line in the table shows the amount of time **D*** took to compute the symbolic derivative.*

Table 2 summarizes the results for **D*** versus Mathematica. We can see that even for the smallest problem size **D*** generates derivatives which are more efficient; the Mathematica derivative size is clearly growing nonlinearly and becomes relatively larger as L increases. For $L = 20$ Mathematica ran out of memory and failed to compute a solution.

order, L	**5**	**15**	**19**	**20**	**5**	**15**	**19**	**20**
operation	$\pm$	$\pm$	$\pm$	$\pm$	$\times$	$\times$	$\times$	$\times$
D*	**57**	**412**	**642**	**707**	**139**	**1714**	**2820**	**3139**
Mathematica	60	2730	11,576	*	179	5351	21,694	*
ratio	**1.3**	**5.6**	**18**	NA	**1.29**	**3.12**	**7.7**	NA

Table 2: *Spherical harmonics: Mathematica vs. **D***. Ratio is the Mathematica operation count divided by the corresponding **D*** value. For L=20 Mathematica ran out of memory and failed to compute the derivative.*

6.2 Inverse Dynamics

For the inverse dynamics problem (sec.5.2) the CppAD derivative evaluation function took approximately 80 times as long as the **D*** derivative evaluation function (table 3).

For this problem the Mathematica FullSimplify function was unreasonably slow, even for $n = 2$, so the Simplify function was used instead for $n = 2, 3, 4$. For $n = 5$ Simplify failed to complete within one hour so no simplification was used. Mathematica could not compute the derivative for $n = 6$ because it ran out of memory. As shown in table(5) Mathematica symbolic computation time is increasing extremely rapidly.

[7] The **D*** implementation, by contrast, computes all n^2 entries in the matrix product and then applies the trace operator to the result.

D*	CppAD	ratio
1,158,748	14,410	**80**

Table 3: *Inverse dynamics, n = 6: CppAD vs. **D***. Number of derivative evaluations per second. Ratio is the number of **D*** evaluations per second divided by the number of CppAD evaluations per second.*

number of links, n	2	3	4	5	6
D*	147	302	479	664	849
Mathematica	241	3127	46,496	797,134	*
ratio	1.6	10	97	1200	NA
D* symbolic time (secs.)	.19	.25	.35	.55	.82

Table 4: *Inverse dynamics: Mathematica vs. **D***. Number of multiply adds in derivative expression. For $n = 2,3,4$ the Mathematica Simplify function was used. For $n = 5$ Simplify did not finish within one hour so no simplification was used. For $n = 6$ Mathematica ran out of memory and could not compute the derivative. The last line in the table is the amount of time **D*** took to compute the symbolic derivative.*

number of links, n	2	3	4
D*	.19	.25	.35
Mathematica	2.7	51	2309
ratio	14	206	6597

Table 5: *Time, in secs., to compute the symbolic derivative of inverse dynamic function, **D*** vs. Mathematica.*

operation	$\pm$	$\times$	Sin	Cos
D* parallel axes	416	433	6	6
Balafoutis parallel axes	386	450	6	6
ratio	.93	1.04	1	1
D* perpendicular axes	369	362	6	6
Balafoutis perpendicular axes	386	450	6	6
ratio	1.05	1.25	1	1

Table 6: *Recursive inverse dynamics vs. **D***, n = 6: For all joint axes parallel **D*** has essentially the same operation count as the $O(n)$ Balafoutis algorithm (see text), which is among the most efficient manually derived recursive inverse dynamics algorithms. For all axes perpendicular **D*** is 14% faster.*

Comparing the **D*** derivative to one of the best published linear recursive inverse dynamics algorithms [Balafoutis and Patel 1991] (table 6) we see the **D*** derivative has only 1.5% more computation when all joint axes are parallel. For all axes perpendicular **D*** is 14% faster[8]. Real robot manipulators will be somewhere between these two extremes.

This is a surprisingly good result because directly evaluating eq.16 has computational complexity $O(n^4)$, where n is the number of links in the manipulator. For a 6 degree of freedom manipulator this formulation requires roughly 120,000 multiply/adds[Balafoutis and Patel 1991]. **D*** has eliminated virtually all of the redundant computation in spite of the fact that the **D*** program uses 4x4 homogeneous transformations which is an inefficient, but simple, way to represent rotation. By contrast, the best inverse dynamics algorithms use more complex specialized representations for transformation matrices[9].

6.3 Space-Time Optimization

number of links, n	D*	CppAD	ratio	D* symbolic time (secs.)
6	332,225	7,613	44	7
12	172,830	2945	59	53
18	109,589	1561	70	199
40	41,000	380	108	3660

Table 7: *Space-time optimization: CppAD vs. **D***. Number of derivative evaluations per second. CppAD is relatively slower as n increases. The last column in the table shows the amount of time **D*** took to compute the symbolic derivative.*

D* derivatives are 44 times faster than CppAD for $n = 6$ (table 7). Derivative evaluations per second for the compiled **D*** derivative is changing almost perfectly linearly as a function of n, while CppAD is not. D* appears to be taking time cubic in the number of links to compute the symbolic derivative. As mentioned in section 4.1 this is because we run the dominator algorithm on the entire graph after every subgraph factorization. The algorithm can be made much more efficient by incrementally updating just those dominance relations which can possibly change. We are implementing this incremental algorithm, which should significantly improve the asymptotic running time, and will report on the results in a future paper. However, even in its current form the algorithm is fast enough to be used for space-time optimization of relatively complex articulated figures such as human beings.

The Mathematica Simplify function did not finish within one hour even on the smallest problem size, $n = 2$, so no simplification was used (table 8). For $n = 4$ Mathematica ran out of memory.

number of links, n	2	3	4	40
D*	419	881	1409	22739
Mathematica	145,950	4,076,910	*	*
ratio	358	4630	NA	NA
D* symbolic time (secs.)	.3	.84	1.9	3660

Table 8: *Space-time optimization: Mathematica vs. **D***. Number of multiply adds in derivative expression. For $n = 2,3$ the Mathematica function Simplify did not finish within one hour so no simplification was used. For $n = 4$ Mathematica ran out of memory and could not compute the derivative. The final line in the table shows the amount of time **D*** required to compute the symbolic derivative.*

It is not possible to precisely compare **D*** to the best manually derived recursive formulas because no operation counts are given in [Lee et al. 2005], and important details are left to the reader to fill in. For example, the authors of [Lee et al. 2005] state

> "It should be noted that many of the computations embedded in the forward and backward recursions above need only be evaluated once, thereby reducing the computational burden."

without giving details as to precisely which computations are redundant; as a result it would be difficult, if not impossible, to precisely reproduce their implementation.

However, we can compare the **D*** derivative to the best results that might be achieved. The lower bound for the gradient is certainly

[8] If we assume multiplication and addition take the same number of clock cycles; on most architectures multiplication is slightly slower than addition which would bias the results more heavily in favor of **D***.

[9] The Balafoutis algorithm, for example, uses a Cartesian tensor representation for which the author provides 80 pages of background mathematics.

no less than the amount of computation required for the space-time objective function, f. For $n = 6$ f has 501 $\pm$, 537 $\times$, 6 cosine and 6 sin operations. The **D*** derivative of f has 1226 $\pm$, 1419 $\times$, 6 cosine and 6 sin operations, which is less than 2.8 times the operations of f. Clearly the **D*** derivative is close to the lower bound.

7 Conclusion and future work

The **D*** symbolic differentiation algorithm generates extremely efficient symbolic derivatives. In some cases the **D*** derivatives rival the best manually derived solutions. For the problems in our test set **D*** outperformed Mathematica by up to 4.6×10^3, and CppAD by up to 2.2×10^5 times. **D*** computed efficient symbolic derivatives for all the problems while Mathematica failed to compute derivatives for three problems because it ran out of memory.

There is considerable scope for reducing the amount of time required to compute the symbolic derivatives by incrementally updating the dominator information for the function subgraphs. This will be the focus of future work.

Acknowledgements

I would like to thank Bradley Bell, the author of CppAD, and Nirupama Chandrasekaran for helping me implement the CppAD examples.

References

BALAFOUTIS, C. A., AND PATEL, R. V. 1991. *Dynamic Analysis of Robot Manipulators: A Cartesion Tensor Approach*. Kluwer Academic Publishers.

BAUER, F. L. 1974. Computational graphs and rounding error. *SIAM J. Numer. Anal. 11*, 1, 87–96.

BISCHOF, C. H., CARLE, A., KHADEMI, P., AND MAUER, A. 1996. ADIFOR 2.0: Automatic differentiation of Fortran 77 programs. *IEEE Computational Science & Engineering 3*, 3, 18–32.

BISCHOF, C. H., ROH, L., AND MAUER, A. 1997. ADIC — An extensible automatic differentiation tool for ANSI-C. *Software–Practice and Experience 27*, 12, 1427–1456.

COOPER, K., HARVEY, T., AND KENNEDY, K. 2001. A simple, fast dominance algorithm. *Software Practice and Experience*.

FANG, A. C., AND POLLARD, N. S. 2003. Efficient synthesis of physically valid human motion. *ACM Transactions on Graphics 22, 3 (July),*, 417–426.

FEATHERSTONE, R., AND ORIN, D. 2000. Robot dynamics: Equations and algorithms. *Proc. IEEE Int. Conf. Robotics & Automation*.

GRIEWANK, A. 2000. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*. Lecture Notes in Computer Science 120.

GRIEWANK, A. 2003. A mathematical view of automatic differentiation. In *Acta Numerica*, vol. 12. Cambridge University Press, 321–398.

LEE, S.-H., KIM, J., PARK, F., KIM, M., AND BOBROW, J. E. 2005. Newton-type algorithms for dynamics-based robot movement optimization. *IEEE Trans. on robotics 21*, 4, 657–667.

LIU, Z., GORTLER, S. J., AND COHEN, M. F. 1994. Hierarchical spacetime control. *Computer Graphics (SIGGRAPH 94 Proceedings)*, 35–42.

LIU, C. K., HERTZMANN, A., AND POPOVIC, Z. 2005. Learning physics-based motion style with inverse optimization. *ACM Transactions on Graphics (SIGGRAPH 2005)*.

MARTIN, B., AND BOBROW, J. 1997. Minimum effort motions for open chain manipulators with task dependent end-effector constraints.

RALL, L. B. 1981. *Automatic Differentiation: Techniques and Applications*. Springer Verlag.

SLOAN, P.-P., LUNA, B., AND SNYDER, J. 2005. Local, deformable precomputed radiance transfer. *Computer Graphics Proceedings, Annual Conference Series*, 1216–1224.

TADJOUDDINE, E. M. 2007. Vertex ordering algorithms for jacobian computations using automatic differentiation. *Submitted to the Computer Journal, Oxford University Press, March 2007*.

VAN DE PANNE, M., LAZLO, J., AND FIUME, E. L. 2000. Interactive control for physically-based animation. *Computer Graphics (SIGGRAPH 2000 Proceedings)*, 201–208.

WATERS, R. C. 1979. Mechanical arm control. *MIT Artificial Intelligence Lab Memo 549*.

WITKIN, A., AND KASS, M. 1988. Spacetime constraints. *Computer Graphics (SIGGRAPH 88 Proceedings) 22*, 159–168.

A Alternative form of the chain rule

The conventional recursive form of the chain rule is:

$$D(f(g_1(h_1,\ldots,h_m),\ldots,g_n(\ldots)) = \sum_{i=1}^{n} \frac{\partial f}{\partial g_i} D(g_i) \tag{23}$$

where the $g_i(\ldots)$ are themselves functions of some h_j and so on. Expanding one level of this recursion for g_1 we get:

$$\frac{\partial f}{\partial g_1} D(g_1) = \frac{\partial f}{\partial g_1} \sum_{j=1}^{m} \frac{\partial g_1}{\partial h_j} D(h_j) \tag{24}$$

$$= \frac{\partial f}{\partial g_1}\frac{\partial g_1}{\partial h_1} D(h_1) + \frac{\partial f}{\partial g_1}\frac{\partial g_1}{\partial h_2} D(h_2) + \ldots \frac{\partial f}{\partial g_1}\frac{\partial g_1}{\partial h_m} D(h_m) \tag{25}$$

If we expand all levels of the recursion this way we see that the derivative, $D(f)$, is simply a sum of products. This form of the chain rule undoes the factorization implicit in eq.(23). Undoubtedly this has been noted many times; the earliest instance we are aware of is [Bauer 1974].

We can write a simple recursive function to evaluate the derivative in this way. The function takes two arguments: the first argument is the product of the partials up to this level of recursion and the second argument is a list which contains all of the partial products in the derivative sum.

```
expD(double product, List sum){
  if(this.isLeaf){sum.Append(product);}
  else{
    foreach(child ci){
      ci.D(product*partialWRTChild(ci), sum)
    }
  }
}
```

The expD function is initially called on the root node with product set to one and with sum set to the empty list. The function partialWRTChild(ci) returns the partial of the current node with respect to child ci. After execution is finished each entry in sum corresponds to the product of all the partial derivatives on one path from the root to the leaf. Summing all the entries in sum gives the derivative.