D* Tutorial

August 14, 2009

1 Introduction

D* is a language for conveniently expressing and computing efficient symbolic derivatives. There are many applications which require computing derivatives, and future chapters will describe several in great detail. This chapter will teach you how to write **D*** programs.

D* is implemented using a technique called language embedding. When you write a **D*** program you are actually programming in C#. Each type in the **D*** language has a corresponding C# class. **D*** mathematical operations are implemented by overloading the standard C# arithmetic operators and by providing special definitions for all the standard mathematical functions, such as sine and cosine.

D* code and C# code can be freely intermingled, with a few caveats. The most important is that the function Function.NewContext must be called before beginning the definition of any **D*** program. This sets up global data structures to keep track of all **D*** variable and function definitions.

2 D* Functions

Every **D*** program is a function from $\mathbb{R}^n \to \mathbb{R}^m$. For example, this program creates the $\mathbb{R}^2 \to \mathbb{R}^1$ function $f = ab$:

```
Function.NewContext(); //must be called before defining any D*
    variables or functions

Variable a = new Variable(), b = new Variable();
Function f = a*b;
f.print();
```

The print function displays the symbolic form of the **D*** program. The console output looks like this:

```
v0*v1
```

Expressions are automatically named by the system if you do not provide a name. Variable names can be assigned in the variable constructor and functions can be named using the `lhsName` property:

```
Function.NewContext(); //must be called before defining any D*
    variables or functions

Variable a = new Variable("a"), b = new Variable("b");
Function f = a*b;
f.lhsName = "f";
f.print();
```

This gives the more readable printout:

```
a*b
```

In general $\mathbb{R}^n \to \mathbb{R}^1$ functions are defined by a statement of the form

```
f = expression
```

where the expression can contain any combination of arithmetic operators and function composition.

$\mathbb{R}^n \to \mathbb{R}^m$functions are defined differently, by using the Function constructor. The following code creates the $\mathbb{R}^2 \to \mathbb{R}^2$ function $\mathbf{g} = (ab, \sin(b))$:

```
Variable a = new Variable("a"), b = new Variable("b");
Function g = new Function(a*b,Function.sin(b));
g.lhsName = "g"
g.print();
```

which prints out like this:

```
g[0]
a*b
g[1]
SINb
```

Individual range elements of a function are accessed with an indexer:

```
Function h = g[0]*g[1]; //(a*b)*sin(b)
```

3 Differentiation

The most powerful feature of **D*** is the ability to easily specify and compute symbolic derivatives which can be evaluated very efficiently. You specify derivatives of arbitrary order with the Function.D function:

```
Variable a = new Variable(), b = new Variable();
Function f = a*b;
Function dfa = Function.D(f,a); // Df/Da

//equality of mixed partials wrt variables
Function dfab = Function.D(f,a,b); // D(Df/Da)/Db
Function dfba = Function.D(f,b,a); // D(Df/Db)/Da

// does D(Df/Db)/Da = D(Df/Da)/Db?
Console.WriteLine((dfab == dfba));
```

which prints out:

```
true
```

Notice that **D*** automatically detects that the two mixed partials are equal and only computes one of them. This will work regardless of the order or number of terms in the mixed partials.

For $\mathbb{R}^n \to \mathbb{R}^m$ functions you must specify the index of the range element you want to take the derivative of unless you are computing a parametric partial:

```
Variable a = new Variable("a"), b = new Variable("b");
Function g = new Function(a*b,Function.sin(b));

Function dg0 = Function.D(g[0],a); // D(a*b)/Da
Function dg1 = Function.D(g[1],b); // D(sin(b))/Db

//take derivative of all range elements
Function dgda = Function.D(g,a);
//dgda[0] = dg[0]/da
//dgda[1] = dg[1]/da
```

Derivatives can be used as arguments to other functions:

```
Function df = Function.D(f,a);
Function sindf = Function.sin(df));
Function ddf = Function.D(df,a);
Function derivExpression = ddf*sindf/df;
```

You can create functions of a variable without specifying what the function is, and you can compute derivatives of the unspecified function:

```
Variable a = new Variable("a");
UnspecifiedFunction q = UnspecifiedFunction.functionOf("q",a);

Function h = Function.D(Function.sin(q),a);
```

You can also take derivatives with respect to functions. Given a function $f(q(t), \dot{q}(t))$ you can specify the derivatives $\frac{\partial f}{\partial q}$and $\frac{\partial f}{\partial \dot{q}}$:

```
Variable a = new Variable("t");
UnspecifiedFunction q = UnspecifiedFunction.functionOf("q",t);

Function dq = Function.D(Function.sin(q),q); // D(sin(q))/Dq

Function qdot = Function.D(q,t); // Dq/Dt
Function L = Function.sin(qdot);
Function dL_dqdot = Function.D(L,qdot); // DL/Dqdot
Function dL_dqdot_dt = Function.D(L,qdot,t); // D(DL/Dqdot)/Dt

Function et = Function.exp(t);
Function dsin_det = Function.D(Function.sin(et),et); // D(sin(e^t))/D(e
    ^t)
```

This type of derivative pops up occasionally, perhaps most importantly in the Euler-Lagrange equations of the calculus of variations. These equations are central to classical mechanics, covered in Chapter **??**.

Equality of mixed partials taken in different order no longer holds when you take derivatives with respect to functions. This is because the things you are differentiating with respect to are not independent. For a function $f(p(t))$

$\frac{\partial f}{\partial p \partial t} = \frac{\partial}{\partial p}\left(\frac{\partial f}{\partial p}\frac{\partial p}{\partial t}\right) = \frac{\partial^2 f}{\partial^2 p}\frac{\partial p}{\partial t} + \frac{\partial f}{\partial p}\left\{\frac{\partial}{\partial p}\left(\frac{\partial p}{\partial t}\right)\right\}$

but

$\frac{\partial f}{\partial t \partial p} = \frac{\partial}{\partial t}\left(\frac{\partial f}{\partial p}\right) = \frac{\partial}{\partial p}\left(\frac{\partial f}{\partial p}\right)\frac{\partial p}{\partial t} = \frac{\partial^2 f}{\partial^2 p}\frac{\partial p}{\partial t}$

The derivatives will be the same if the term $\frac{\partial}{\partial p}\left(\frac{\partial p}{\partial t}\right)$ is zero. An example where this is not true is the function $\cos(e^t)$ with $f = \cos()$ and $p = e^t$

$\frac{\partial(\cos(e^t)}{\partial t \partial p} = \frac{\partial}{\partial t}\left(-\sin(e^t)\right) = -\cos(e^t)e^t$

but

$\frac{\partial(\cos(e^t)}{\partial p \partial t} = \frac{\partial}{\partial p}\left(-\sin(e^t)e^t\right) = -\cos(e^t)e^t - \sin(e^t)$

4 More Complex Functions

So far all of the **D*** functions we have written have been simple expressions. What if your function is too complicated to fit on a single line or if there are complex conditional expressions that have to be evaluated during its creation?

Because **D*** is embedded in C# it is easy to write C# functions which return **D*** functions. Let's write a program, revSurf, that will take as input a two-dimensional profile curve, represented as an $\mathbb{R}^1 \to \mathbb{R}^2$ **D*** function

$$f(t) := [x(t), y(t)]^T$$

and return a new $\mathbb{R}^2 \to \mathbb{R}^3$ **D*** function

$$g(\theta, t) = [x(\theta, t), y(\theta, t), z(\theta, t)]^T$$

which represents a surface of revolution along the y axis:

```
Function revSurf(Variable theta, Variable t, Function f){
  const int x = 0, y = 1;
  Function cosTheta = Function.cos(theta),
         sinTheta = Function.sin(theta);
  Function df = Function.D(f, t);
  Function denominator = Function.sqrt(df[x]*df[x] + df[y]*df[y]);
  Function d = 1 / denominator;
  return new Function(cosTheta * f[x], f[y], sinTheta * f[x]);
}
```

If we are given some **D*** function xyfunction as our profile curve input then we can create the **D*** function representing the surface of revolution like this:

```
//returns a D* function from R1->R2
Function xyfunction(Variable var){...}

//make the surface of revolution
Variable theta, t;
Function surface = revSurf(theta,t,xyFunction(t));
```

5 Recursive Functions

It is easy to define recursive C# functions which return **D*** functions. We will do a simple example here, Chebyshev polynomials, and a more complicated example later in Section **??** . The recursion equation for the Chebyshev polynomial, T_n, of order n is

$$\begin{aligned} T_0(x) &= 1 \\ T_1(x) &= x \\ T_{n+1}(x) &= 2xT_n(x) - T_{n-1}(x) \quad n \geq 1 \end{aligned}$$

The C# function which implements this recursion looks almost exactly like the mathematical equations:

```
//returns n degree Chebyshev polynomial
Function T(Function x, int n){
  if(n == 0){return 1;}
  if(n == 1){return x;}
  return 2*x*T(x,n-1) - T(x,n-2);
}
```

Let's print out the symbolic form of the Chebyshev polynomial of degree 4 and compare that to the number of operations in the **D*** expression graph. The printOperatorCounts method will print out the total number of operations of each kind that are present in the graph.

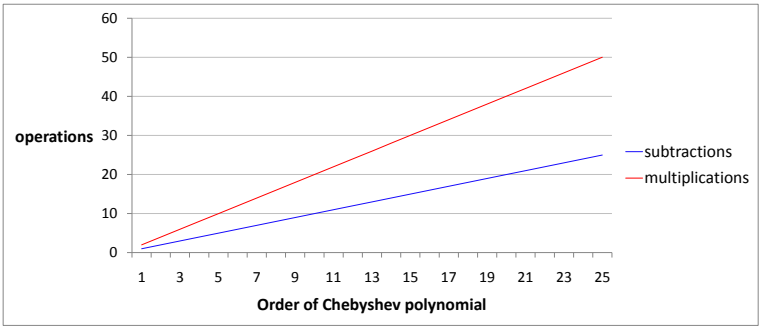

Figure 1: Arithmetic operations in **D*** Chebyshev polynomial function

```
Variable x = new Variable("x");
Function res = T(x,4);

res.printOperatorCounts();
res.print();
```

which prints out :

```
-:4 *:8

(x*(x*(x*(x*x*2 - 1)*2 - x)*2 - (x*x*2 - 1))*2 - (x*(x*x*2 - 1)*2 - x))
```

There are 4 subtractions and 8 multiplications in the **D*** graph but there are 14 multiplications and 7 subtractions in the symbolic printout. This is because there are many common subexpressions in the graph; the print function recursively expands the graph into a tree which, in the worst case, will lead to a symbolic printout that will be exponentially larger than the expression graph.

In Figure 1 you can see that the number of arithmetic operations in the **D*** function increases linearly as the order of the Chebyshev polynomial increases; **D*** has automatic common subexpression elimination, which is detecting and eliminating the many common terms that result from the recursion.

The *time* it takes to compute these polynomials is a different matter, however. Looking at the curve labeled not-memoized in Figure 2 you can see that computation time is increasing exponentially as a function of polynomial order. **D*** is eliminating the common subexpressions as it finds them but they are still being created, which is taking exponential time.

Because recursive definitions of mathematical functions are common, and we don't want to waste time computing them, **D*** has a feature called memoization. Memoization caches the values of recursive function calls so that they do not have to be recomputed. We can redefine our Chebyshev function as a lambda expression and then apply the Memoize method extension to memoize the function:

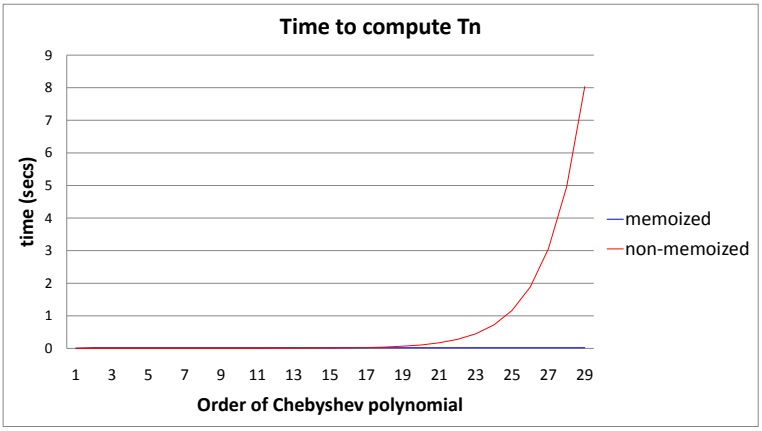

Figure 2: Memoized versus non-memoized recursive function execution time

```
Func<Function, int, Function> T = null;
T = (Function x, int n) => {
  if (n == 0) { return 1; }
  if (n == 1) { return x; }
  return 2 * x * T(x, n - 1) - T(x, n - 2);
};

//memoize the function
T = T.Memoize();

//create Chebyshev polynomial of order 10
Variable y = new Variable("y");
Function cheb10 = T(y,10);
```

There is a big difference in the execution time of the memoized function vs. non-memoized function, as shown in Figure 2. The memoized function has essentially constant execution time as a function of polynomial order[1]; for $n = 29$ the memoised function takes approximately .02 seconds while the non-memoized function takes 8 seconds, roughly 400 times longer.

6 Piecewise Functions

Some functions are most easily represented in piecewise fashion; piecewise polynomial splines are a widely used example which we will see much more of in Chapter ??. In a piecewise function the symbolic function definition itself is a

[1] For $n < 17$ the overhead of setting up the memoization is greater than the cost of evaluating the Chebyshev recursion. This is why you shouldn't blindly use memoization all the time; for small functions it is faster not to memoize.

function of some other function or variable.

The FArray class along with the Function.floor, Function.ceiling, Function.max, Function.min operators provides this functionality. To create an FArray you use Function.array. Each element in an FArray is an independent function:

```
Variable r = new Variable("r");
Variable s = new Variable("s");
FArray arr = Function.array(1.0, s, r * s);

arr.lhsName = "a";
arr.get(0).print();
arr.get(1).print();
arr.get(2).print();
```

which prints out:

```
a[0] // references 1.0
a[1] // references s
a[2] // references r*s
```

Note that the get function doesn't return the contents of the array element being indexed. Instead it returns an instance of type Reference which is a function which references the array element[2].

We can use arrays to create a C# class which will make **D*** functions that represent cubic B-splines:

Listing 1: Cubic B-spline Function

```
public class BSpline{
  FArray P; //controlPoints
  public BSpline(params Function[] controlPoints){
    P = Function.array(controlPoints);
  }

  public Function curveValue(Function t){
    IntegerValue index = Function.floor(t + 3);
  t = t - Function.floor(t);
    Function B3 = ((1 - t)^3);
    Function B2 = 3 * (t^3) - 6 * (t^2) + 4;
    Function B1 = 3 * (-(t^3) + (t^2) + t) + 1;
    Function B0 = (t^3);
    return (1.0 / 6.0) * (B0 * P.get(index) +
                    B1 * P.get(index - 1) +
                    B2 * P.get(index - 2) +
                    B3 * P.get(index - 3));
  }
```

[2]Why? This is one of those technical details to be explained in Section 3.

```
  public Function tangent(Function t){
  fTangent = Function.D(fCurveValue, t);
  fTangent = Function.derivative(fTangent);
  return fTangent;
  }

   public Function acceleration(Function t){
  if (fTangent == null){
   fTangent = tangent(t);
  }
  fAcceleration = Function.D(fTangent, t);
  fAcceleration = Function.derivative(fAcceleration);
  return fAcceleration;
  }
}
```

There are several new features in this code: the IntegerValue class, the Function.floor function, and the exponentiation operator ^.

The argument to the get function for Farray must be of type IntegerValue. Function.floor returns a function of type IntegerValue which computes the floor of its argument. The ^ operator performs exponentiation by integer powers. Because the ^ operator has the lowest precedence of the C# operators you must enclose your exponentiation expression in parentheses to avoid unexpected results.

Using the Bspline class you can make a Bspline with constant coefficients:

```
BSpline c = new BSpline(1, 2, 3, 4, 5);
Variable t = new Variable("t");
Function ct = c.curveValue(t);
```

or with variable coefficients:

```
Variable a0 = new Variable("a0"),
       a1 = new Variable("a1"),
       a2 = new Variable("a2"),
       a3 = new Variable("a3"),
       a4 = new Variable("a4");

BSpline c = new BSpline(a0,a1,a2,a3,a4);
Variable t = new Variable("t");
Function ct = c.curveValue(t);
```

You can also have coefficients that are mixtures of functions, variables, and constants:

```
Variable r = new Variable("t"),
       s = new Variable("s");

BSpline c = new BSpline(Function.sin(r),(r^2),s*r,3,r);
Variable t = new Variable("t");
Function ct = c.curveValue(t);
```

You can compute derivatives of piecewise functions but not with respect to a variable or function that is in one of the Farray elements that make up the piecewise function.

```
Variable r = new Variable("r"), t = new Variable "t";
FArray arr = Function.array(1.0, r, (r^2));
Function ct = t*arr.get(Function.floor(t));

ct1 = Function.D(ct, t); // okay: t is not an element of the array

ct2 = Function.D(ct, r); // not okay: r is an element of the array
```

7 Evaluating D* Functions

You may have noticed that none of the previous examples had a printout of a **D*** function which contained a derivative. Let's make an example that does this right now:

```
Variable a = new Variable("a"), b = new Variable("b");
Function f = new Function(a * b);
Function g = Function.D(f, b); // D(a*b)/Db = a
g.lhsName = "g";
g.print();
```

This prints out:

```
(a*b derivative b)
```

This is surprising; instead of what you would expect, $\frac{\mathrm{d}(a*b)}{\mathrm{d}b} = a$, you get this funny (a*b derivative b) thing. What is going on here? When Function.D executes it doesn't immediately compute a derivative; it creates a specification of a derivative, a placeholder in the expression graph for the actual derivative. This is because the **D*** derivative analysis algorithm[3] needs definitions of *all* the derivatives in your function so that it can globally analyze the entire graph to determine the most efficient way to compute all the derivatives at once.

To actually compute the symbolic derivative you use Function.derivative.

[3] Explained in Chapter **??**.

This function invokes the global derivative analysis algorithm and computes a new symbolic expression graph which has actual symbolic derivatives rather than placeholders. If we apply this to our example function we get:

```
g = Function.derivative(g);
g.print();
```

which prints out:

```
a
```

7.1 Compiling D* Functions

Interpretive evaluation of the function graph would be very slow so evaluation of D* functions is done by transforming the D* expression graph to an intermediate high level language and then compiling to an executable. The D* code generator has two back ends: C# and C++. C# code can be dynamically compiled and executed immediately in the calling function. C++ code must be written to a file, compiled off-line, and then manually combined with the user code that calls the D* function.

For functions with less than 64 local variables the C# and C++ code have eqivalent performance. However, for code with more than 64 local variables the C# .NET jit compiler does not do register allocation[4], which results in code that can be 5 to 10 times slower than offline compiled C++ code[5]. If you have large functions and need maximum performance you should use the C++ backend, invoked by the function:

```
compileCCodeToFile(string filename)
```

To see how this compilation process works let's define a simple $\mathbb{R}^2 \rightarrow \mathbb{R}^2$ function $\mathbf{g} = (ab, \sin(b))$:

```
Function.newContext();
Variable a = new Variable("a"), b = new Variable("b");
Function f = new Function(a * b, Function.sin(b));
Function g = Function.D(f, b);
Function e = Function.derivative(g);

e.lhsName = "e";
e.print();
```

which prints:

[4] As of the middle of 2009.

[5] The more local variables the slower the code.

```
e[0]
a
e[1]
COSb
```

and compile this function into an executable using the compile method, which by default dynamically compiles C# intermediate code. You can see the intermediate C# code that the system creates by setting the Function.printCompilerSource to true.

```
Function.printCompilerSource = true;
RuntimeFunction erun = e.compile();
```

This prints the automatically generated source code:

```
using System;
using System.Collections;
using System.IO;
namespace DifferentiableFunction {
   public class newClass0:DifferentiableFunction.RuntimeFunction{
      public int rangeDimension{get{return 2;}}
      public int domainDimension{get{return 2;}}
      protected double Square(double a){return a*a;}

      public void eval(double[] result, params double[] vars){

// DOMAIN variables
// a = index:0
// b = index:1

// RANGE variables
// a = index:0
// Dv3_Db = index:1

         double a,b;
         double Dv3_Db;

         a= vars[0];
         b= vars[1];
//**** a

//**** Dv3_Db
         Dv3_Db = Math.Cos(b);
         result[0] = a;
         result[1] = Dv3_Db;
       }
    }
}
```

To compute the value of the derivative at a particular point use the eval method:

```
double[] result = new double[2], vars = {1,Math.PI};
erun.eval(result,vars);
Console.WriteLine("e[0]:" + result[0] + "e[1]:" + result[1]);
```

this prints out :

```
e[0]:
1
e[1]:
-1
```

In this case we knew that vars[0] corresponded to variable a and vars[1] corresponded to variable b because we printed out the automatically generated source code. It is possible, though, that if the e function was defined in a different context that the correspondence might be different; vars[1] might correspond to variable a and vars[0] to b. There is no way to know exactly what order **D*** will put variables in since the algebraic simplification rewrite rules can change the order in which they occur in an expression. Things become even more complicated if the expression has UnspecifiedFunction elements. For example, given the code to compute $\frac{\mathrm{d}[ab,\sin(q_0(t)]^T}{\mathrm{d}b} = [a, \cos(q(t))\dot{q}_0(t)]^T$:

```
Variable a = new Variable("a"), t = new Variable("t");
UnspecifiedFunction q0 = UnspecifiedFunction.functionOf("q0", t);

Function f = new Function(a * t, Function.sin(q0));
Function g = Function.D(f, t);
Function e = Function.derivative(g);
e.lhsName = "e";
e.print();
```

```
e[0]
a
e[1]
COSq0*q0_d_t
```

the printout for the function e[1] has a new UnspecifiedFunction, q0_d_t, corresponding to $\dot{q}_0(t)$, which was not explicitly declared in the code. The system has created this new UnspecifiedFunction term automatically.

You specify the correspondence between variables and indices in the vars argument of the eval function by using the Function.orderVariablesInDomain function. Once you order the variables in a function they are guaranteed to stay in that order. Variables are always ordered first followed by unspecified function terms:

```
Function dq0db = Function.D(q0,b);

e.order(new[]{a, b}, new[]{q0,dq0db});

Function.printCompilerSource = true;
RuntimeFunction grun = e.compile();
```

If we look at just that portion of the automatically generated C# source

code which relates to the mapping between variables and indices, you see that the mapping is the way we specified it:

```
public void eval(double[] result, params double[] vars){// DOMAIN
    variables
// a = index:0
// b = index:1
;
// UNSPECIFIED function variables
// q0 = index:2
// q0_d_b = index:3

// RANGE variables
// a = index:0
// Dv6_Db = index:1

double a,b;
double Dv6_Db,v13,v5,q0,v15,v14,q0_d_b;

a= vars[0];
b= vars[1];

q0= vars[2];
q0_d_b= vars[3];
```

Function expressions with many operations[6] can take a long time to differentiate. If the symbolic derivative is not changing between invocations then it can be much faster to save the evaluation function to disk and then read it in again when you need it. To save the code for the evaluation function to disk use the function compileToFile. Use the function Function.compileFromFile to compile the evaluation function stored in a file. Here is a code snippet showing the use of these functions:

```
Variable a = new Variable("a"), b = new Variable("b"),
       c = new Variable("c"), d = new Variable("d");

Function f = (a + b) * (c + d);

f.compileToFile(filename);
RuntimeFunction r = Function.compileFromFile(filename);
double[] result = new double[1], vars = { 1, 1, 1, 1 };
r.eval(result, vars);
```

[6] Thousands of operations or more.

8 Expression Optimization

D* performs two kinds of expression optimization: common subexpression elimination, and algebraic simplification. 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. Similarly, the variable arguments to `Function.D` are sorted by their unique identifier before computing the hash code. This ensures that $\frac{\partial^2 g}{\partial^2 ab}$ and $\frac{\partial^2 g}{\partial^2 ba}$ will hash to the same value.

Algebraic simplification is performed by creating special constructors for each operator. For example, this constructor for the $*$ operator

```
public static Function operator *(Function a,Function b)
  Function alreadyExists = commutativeOperators(typeof(Times),a,b);
  if (alreadyExists != null) return alreadyExists;
  //do various simple constant optimizations
  Constant ca = a as Constant,cb = b as Constant;
  if (ca != null && cb != null) return ca * cb;
  //will use Constant * operator overloading
  if (ca != null){
    if (ca.leafValue == 0) return 0;
    if (ca.leafValue == 1) return b;
    if (ca.leafValue == -1) return -b;
    if (ca.leafValue < 0) return -(b * (-ca.leafValue));
    return (new Times()).compose(b,a);
  }
  if (cb != null){
    if (cb.leafValue == 0) return 0;
    if (cb.leafValue == 1) return a;
    if (cb.leafValue == -1) return -a;
    if (cb.leafValue < 0) return -(a * (-cb.leafValue));
    return (new Times()).compose(a,b);
  }
  return (new Times()).compose(a,b);
```

performs the following symbolic algebraic simplifications:

$$\begin{array}{lcllcl} a*1 & \rightarrow & a & a*-1 & \rightarrow & -a \\ a*0 & \rightarrow & 0 & c_0*c_1 & \rightarrow & \text{Constant}(c_0*c_1) \end{array}$$

where a is a variable argument to the $*$ operator, c_0, c_1 are constant arguments to the $*$ operator, and `Constant()` is the constructor for the `Constant` class which creates a new node that has a constant value.

Similar algebraic simplification rules can be incorporated in the constructors for other arithmetic and functional operations. 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	$f(c_0)$	$\rightarrow$	$\text{Constant}(f(c_0))$
$c_0 * c_1$	$\rightarrow$	$\text{Constant}(c_0 * c_1)$	$c_0 \pm c_1$	$\rightarrow$	$\text{Constant}(c_0 \pm c_1)$
c_0/c_1	$\rightarrow$	$\text{Constant}(c_0/c_1)$			

9 Related Work

There are several commonly used methods of computing derivatives: first order finite differencing, Richardson's extrapolation to the limit (a high order form of finite differencing), automatic differentiation, and symbolic differentiation.

The first order finite difference method is both inaccurate and much less efficient, in general, than other techniques so it won't be discussed further. Richardson's extrapolation to the limit [?] can yield very accurate derivatives but it requires many evaluations of the function to be differentiated, making it extremely inefficient. For $f : \mathbb{R}^n \rightarrow \mathbb{R}^m$ nk function evaluations and $O(mk^2)$ arithmetic operations are required, where k is typically 5 t0 10. In addition the user must specify h, the initial step size. A principled selection of h requires knowledge of the second derivative, which is normally unavailable.

Forward and reverse automatic differentiation are non-symbolic techniques independently developed by several groups in the 60s and 70s respectively[7] [?, ?]. 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 [?] and ADIC [?] automatically apply one form at the statement level and a different one 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 [?, ?] using heuristics which minimize fill in. However, as of the time of [?] good elimination heuristics that worked well on a wide range of problems remained to be developed.

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

[7] See Section ?? for symbolic versions of these algorithms

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.

10 Advantages of the D* Algorithm

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 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 using 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. **D*** also symbolically executes the expression graph at compile time to eliminate common subexpressions and perform simple algebraic simplification. Because **D*** is embedded in a conventional programming language[8] **D*** programs can be seamlessly interleaved with other code, which is very beneficial from a software engineering perspective.

11 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 $O(nmv^3)$. More

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

details are provided in Sections ?? and ??. 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.