Automatic Differentiation – Lecture No 1

Warwick Tucker

The CAPA group
Department of Mathematics
Uppsala University, Sweden

eScience Winter School, Geilo

UPPSALA
UNIVERSITET

When do we use derivatives?

Introduction to AD

When do we use derivatives?

- (1:st order) Solving non-linear equations: Newton's method, monotonicity. Stability.

When do we use derivatives?

- (1:st order) Solving non-linear equations: Newton's method, monotonicity. Stability.
- (2:nd order) Optimization: convexity.

When do we use derivatives?

- (1:st order) Solving non-linear equations: Newton's method, monotonicity. Stability.
- (2:nd order) Optimization: convexity.
- (n:th order) High-order approximations, quadrature, differential equations.

When do we use derivatives?

- (1:st order) Solving non-linear equations: Newton's method, monotonicity. Stability.
- (2:nd order) Optimization: convexity.
- (n:th order) High-order approximations, quadrature, differential equations.

Example (A simple calculus task)

What is the value of $f^{(n)}(x_0)$, where

$$f(x) = e^{\sin e^{\cos x + 2x^5}}$$

UPPSALA UNIVERSITET

When do we use derivatives?

- (1:st order) Solving non-linear equations: Newton's method, monotonicity. Stability.
- (2:nd order) Optimization: convexity.
- (n:th order) High-order approximations, quadrature, differential equations.

Example (A simple calculus task)

What is the value of $f^{(n)}(x_0)$, where

$$f(x) = e^{\sin e^{\cos x + 2x^5}}$$

for $x_0 = +1$ and $n = 1$? [Undergraduate maths - but tedious]

When do we use derivatives?

- (1:st order) Solving non-linear equations: Newton's method, monotonicity. Stability.
- (2:nd order) Optimization: convexity.
- (n:th order) High-order approximations, quadrature, differential equations.

Example (A simple calculus task)

What is the value of $f^{(n)}(x_0)$, where

$$f(x) = e^{\sin e^{\cos x + 2x^5}}$$

for $x_0 = +1$ and $n = 1$? [Undergraduate maths - but tedious]
for $x_0 = -2$ and $n = 100$? [Undergraduate maths - impossible?]

How do we compute derivatives in practice?

How do we compute derivatives in practice?

- (Symbolic representation) Generates exact formulas for $f, f', \ldots, f^{(n)}, \ldots$. This is very memory/time consuming. Produces enormous formulas. Actually too much information.

How do we compute derivatives in practice?

- (Symbolic representation) Generates exact formulas for $f, f', \dots, f^{(n)}, \dots$. This is very memory/time consuming. Produces enormous formulas. Actually too much information.
- (Finite differences) Generates numerical approximations of the value of a derivative, e.g. $f'(x_0) \approx \frac{f(x_0+h)-f(x_0)}{h}$, based on

$$f(x_0+h) = f(x_0) + hf'(x_0) + h^2 f''(x_0) + \mathcal{O}(h^3).$$

Various errors: roundoff, cancellation, discretization. Which h is optimal? How does the error behave? Can't really handle high-order derivatives.

Introduction to AD

How do we compute derivatives in practice?

- (Symbolic representation) Generates exact formulas for $f, f', \ldots, f^{(n)}, \ldots$. This is very memory/time consuming. Produces enormous formulas. Actually too much information.
- (Finite differences) Generates numerical approximations of the value of a derivative, e.g. $f'(x_0) \approx \frac{f(x_0+h)-f(x_0)}{h}$, based on

  $$f(x_0 + h) = f(x_0) + hf'(x_0) + h^2 f''(x_0) + \mathcal{O}(h^3).$$

  Various errors: roundoff, cancellation, discretization. Which h is optimal? How does the error behave? Can't really handle high-order derivatives.
- (Complex differentiation) A nice "trick" using complex extensions: $f'(x_0) \approx \frac{\Im(f(x_0+ih))}{h}$, where $\Im(x + iy) = y$. Avoids cancellation, and gives quadratic approximation, but requires a complex extension of the function.

Example

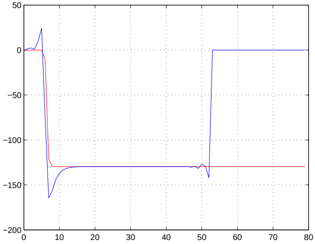
Consider our test function $f(x) = e^{\sin e^{\cos x + 2x^5}}$. Let $h = 2^{-k}$ for $k = 0, \ldots, 80$, and compute the two approximations

$$f_x(h) = \frac{f(1) - f(1+h)}{h} \quad \text{and} \quad f_z(h) = \frac{\Im(f(1+ih))}{h}.$$

Example

Consider our test function $f(x) = e^{\sin e^{\cos x + 2x^5}}$. Let $h = 2^{-k}$ for $k = 0, \ldots, 80$, and compute the two approximations

$$f_x(h) = \frac{f(1) - f(1+h)}{h} \quad \text{and} \quad f_z(h) = \frac{\Im(f(1+ih))}{h}.$$



Plot of k versus $f_z(h)$ and $f_z(h)$.

$f_x(h)$ is plotted in blue, $f_z(h)$ is plotted in red. Notice that $f_z(h)$ is not affected by cancellation due to a small h.

Automatic differentiation

Generates evaluations (and not formulas) of the derivatives. Based on a strategy similar to symbolic differentiation, but does not use placeholders for constants or variables. All intermediate expressions are evaluated as soon as possible; this saves memory, and removes the need for later simplification.

Automatic differentiation

Generates evaluations (and not formulas) of the derivatives. Based on a strategy similar to symbolic differentiation, but does not use placeholders for constants or variables. All intermediate expressions are evaluated as soon as possible; this saves memory, and removes the need for later simplification.

Bonus properties

Introduction to AD

Automatic differentiation

Generates evaluations (and not formulas) of the derivatives. Based on a strategy similar to symbolic differentiation, but does not use placeholders for constants or variables. All intermediate expressions are evaluated as soon as possible; this saves memory, and removes the need for later simplification.

Bonus properties

- No discretization errors.

Automatic differentiation

Generates evaluations (and not formulas) of the derivatives. Based on a strategy similar to symbolic differentiation, but does not use placeholders for constants or variables. All intermediate expressions are evaluated as soon as possible; this saves memory, and removes the need for later simplification.

Bonus properties

- No discretization errors.
- No huge memory consumption.

Introduction to AD

Automatic differentiation

Generates evaluations (and not formulas) of the derivatives. Based on a strategy similar to symbolic differentiation, but does not use placeholders for constants or variables. All intermediate expressions are evaluated as soon as possible; this saves memory, and removes the need for later simplification.

Bonus properties

- No discretization errors.
- No huge memory consumption.
- No complex “tricks“.

Automatic differentiation

Generates evaluations (and not formulas) of the derivatives. Based on a strategy similar to symbolic differentiation, but does not use placeholders for constants or variables. All intermediate expressions are evaluated as soon as possible; this saves memory, and removes the need for later simplification.

Bonus properties

- No discretization errors.
- No huge memory consumption.
- No complex “tricks“.
- Very easy to understand.

First-order scalar AD - definitions

An arithmetic for differentiation

We will perform all computations with ordered pairs of real numbers

$$\vec{u} = (u, u').$$

First-order scalar AD - definitions

An arithmetic for differentiation

We will perform all computations with ordered pairs of real numbers

$$\vec{u} = (u, u').$$

The first component holds the value of the function $f(x_0)$; the second component holds the value of the derivative $f'(x_0)$. In what follows, we assume that $f : \mathbb{R} \to \mathbb{R}$.

First-order scalar AD - definitions

An arithmetic for differentiation

We will perform all computations with ordered pairs of real numbers

$$\vec{u} = (u, u').$$

The first component holds the value of the function $f(x_0)$; the second component holds the value of the derivative $f'(x_0)$. In what follows, we assume that $f : \mathbb{R} \to \mathbb{R}$.

Basic arithmetic

$$\begin{aligned}
\vec{u} + \vec{v} &= (u + v, u' + v') \\
\vec{u} - \vec{v} &= (u - v, u' - v') \\
\vec{u} \times \vec{v} &= (uv, uv' + u'v) \\
\vec{u} \div \vec{v} &= (u/v, (u' - (u/v)v')/v),
\end{aligned}$$

where we demand that $v \neq 0$ when dividing.

We need to know how constants and the independent variable x are treated. Following the usual rules of differentiation, we define

$$\vec{x} = (x, 1) \qquad \text{and} \qquad \vec{c} = (c, 0).$$

First-order scalar AD - definitions

We need to know how constants and the independent variable x are treated. Following the usual rules of differentiation, we define

$$\vec{x} = (x, 1) \qquad \text{and} \qquad \vec{c} = (c, 0).$$

Example

Let $f(x) = \frac{(x+1)(x-2)}{x+3}$. We wish to compute the values of $f(3)$ and $f'(3)$. It is easy to see that $f(3) = 2/3$. The value of $f'(3)$, however, is not immediate.

UPPSALA UNIVERSITET

First-order scalar AD - definitions

We need to know how constants and the independent variable x are treated. Following the usual rules of differentiation, we define

$$\vec{x} = (x, 1) \qquad \text{and} \qquad \vec{c} = (c, 0).$$

Example

Let $f(x) = \frac{(x+1)(x-2)}{x+3}$. We wish to compute the values of $f(3)$ and $f'(3)$. It is easy to see that $f(3) = 2/3$. The value of $f'(3)$, however, is not immediate. Applying the techniques of differentiation arithmetic, we define

$$\vec{f}(\vec{x}) = \frac{(\vec{x}+\vec{1})(\vec{x}-\vec{2})}{\vec{x}+\vec{3}} = \frac{\big((x,1)+(1,0)\big) \times \big((x,1)-(2,0)\big)}{(x,1)+(3,0)}.$$

Inserting the AD-variable $\vec{x} = (3, 1)$ into $\vec{f}$ produces...

First-order scalar AD - definitions

Example

$$\begin{aligned}\vec{f}(3,1) &= \frac{\big((3,1)+(1,0)\big)\times\big((3,1)-(2,0)\big)}{(3,1)+(3,0)} \\ &= \frac{(4,1)\times(1,1)}{(6,1)} = \frac{(4,5)}{(6,1)} = (\frac{2}{3},\frac{13}{18}).\end{aligned}$$

UPPSALA UNIVERSITET

Example

$$\begin{aligned}\vec{f}(3,1) &= \frac{\big((3,1)+(1,0)\big)\times\big((3,1)-(2,0)\big)}{(3,1)+(3,0)} \\ &= \frac{(4,1)\times(1,1)}{(6,1)} = \frac{(4,5)}{(6,1)} = (\frac{2}{3},\frac{13}{18}).\end{aligned}$$

From this calculation it follows that $f(3) = 2/3$ (which we already knew) and $f'(3) = 13/18$. Note that we never used the expression for f'.

UPPSALA UNIVERSITET

First-order scalar AD - definitions

Example

$$\vec{f}(3,1) = \frac{\big((3,1)+(1,0)\big)\times\big((3,1)-(2,0)\big)}{(3,1)+(3,0)}$$
$$= \frac{(4,1)\times(1,1)}{(6,1)} = \frac{(4,5)}{(6,1)} = \left(\frac{2}{3}, \frac{13}{18}\right).$$

From this calculation it follows that $f(3) = 2/3$ (which we already knew) and $f'(3) = 13/18$. Note that we never used the expression for f'.

If we use the different (but equivalent) representation $f(x) = x - \frac{4x+2}{x+3}$, we arrive at the same result by a completely different route. Try it!

UPPSALA UNIVERSITET

First-order scalar AD - definitions

AD for standard functions

We can extend the ideas to standard functions using the chain rule:

$$\vec{g}(\vec{u}) = \vec{g}(u, u') = (g(u), u' g'(u)).$$

First-order scalar AD - definitions

AD for standard functions

We can extend the ideas to standard functions using the chain rule:

$$\vec{g}(\vec{u}) = \vec{g}(u, u') = (g(u), u' g'(u)).$$

Applying this to some common functions yields:

$$
\begin{array}{lllll}
\sin \vec{u} & = & \sin (u, u') & = & (\sin u, u' \cos u) & \\
\cos \vec{u} & = & \cos (u, u') & = & (\cos u, -u' \sin u) & \\
e^{\vec{u}} & = & e^{(u,u')} & = & (e^{u}, u' e^{u}) & \\
\log \vec{u} & = & \log (u, u') & = & (\log u, u'/u) & (u > 0) \\
|\vec{u}| & = & |(u, u')| & = & (|u|, u' \mathsf{sign}(u)) & (u \neq 0) \\
\vec{u}^{\alpha} & = & (u, u')^{\alpha} & = & (u^{\alpha}, u' \alpha u^{\alpha - 1}) & (\text{sometimes}).
\end{array}
$$

Feel free to add your own favourites!

UPPSALA UNIVERSITET

First-order scalar AD - definitions

Example

Let $f(x) = (1 + x + e^x)\sin x$, and compute $f'(0)$.

UPPSALA UNIVERSITET

First-order scalar AD - definitions

Example

Let $f(x) = (1 + x + e^x)\sin x$, and compute $f'(0)$. Set

$$\vec{f}(\vec{x}) = (\vec{1} + \vec{x} + e^{\vec{x}})\sin \vec{x},$$

and evaluate it at $\vec{x} = (0, 1)$.

First-order scalar AD - definitions

Example

Let $f(x) = (1 + x + e^x)\sin x$, and compute $f'(0)$. Set

$$\vec{f}(\vec{x}) = (\vec{1} + \vec{x} + e^{\vec{x}})\sin\vec{x},$$

and evaluate it at $\vec{x} = (0, 1)$. This gives

$$\begin{aligned}
\vec{f}(0,1) &= \left((1,0) + (0,1) + e^{(0,1)}\right)\sin(0,1) \\
&= \left((1,1) + (e^0, e^0)\right)(\sin 0, \cos 0) = (2,2)(0,1) = (0,2).
\end{aligned}$$

From this calculation, it follows that $f(0) = 0$ and $f'(0) = 2$.

UPPSALA UNIVERSITET

First-order scalar AD - definitions

Example

Let $f(x) = (1 + x + e^x) \sin x$, and compute $f'(0)$. Set

$$\vec{f}(\vec{x}) = (\vec{1} + \vec{x} + e^{\vec{x}}) \sin \vec{x},$$

and evaluate it at $\vec{x} = (0, 1)$. This gives

$$\begin{aligned} \vec{f}(0,1) &= \left((1,0) + (0,1) + e^{(0,1)}\right) \sin (0,1) \\ &= \left((1,1) + (e^0, e^0)\right)(\sin 0, \cos 0) = (2,2)(0,1) = (0,2). \end{aligned}$$

From this calculation, it follows that $f(0) = 0$ and $f'(0) = 2$.

Note that the differentiation arithmetic is well-suited for implementations using operator overloading (`C++`, `MATLAB` `Java`).

UPPSALA UNIVERSITET

First-order scalar AD - implementations

Implementing the class constructor is straight-forward in MATLAB.

```
function ad = autodiff(val, der)
% A naive autodiff constructor.
ad.val = val;
if nargin == 1
    der = 0.0;
end
if strcmp(der,'variable')
    der = 1.0;
end
ad.der = der;
ad = class(ad, 'autodiff');
```

First-order scalar AD - implementations

Implementing the class constructor is straight-forward in MATLAB.

```
01 function ad = autodiff(val, der)
02 % A naive autodiff constructor.
03 ad.val = val;
04 if nargin == 1
05     der = 0.0;
06 end
07 if strcmp(der,'variable')
08     der = 1.0;
09 end
10 ad.der = der;
11 ad = class(ad, 'autodiff');
```

Lines 04-06 automatically cast a real number c into an AD-type constant $\vec{c} = (c, 0)$.

UPPSALA
UNIVERSITET

First-order scalar AD - implementations

Implementing the class constructor is straight-forward in MATLAB.

```
01 function ad = autodiff(val, der)
02 % A naive autodiff constructor.
03 ad.val = val;
04 if nargin == 1
05     der = 0.0;
06 end
07 if strcmp(der,'variable')
08     der = 1.0;
09 end
10 ad.der = der;
11 ad = class(ad, 'autodiff');
```

Lines 04-06 automatically cast a real number c into an AD-type constant $\vec{c} = (c, 0)$.
Lines 07-09 manually cast a real number x into a AD-type variable $\vec{x} = (x, 1)$.

First-order scalar AD - implementations

The display of `autodiff` objects is handled via `display.m`:

```
function display(ad)
% A simple output formatter for the autodiff class.
disp([inputname(1), ' = ']);
fprintf('  (%17.17f, %17.17f)\n', ad.val, ad.der);
```

First-order scalar AD - implementations

The display of `autodiff` objects is handled via `display.m`:

```
function display(ad)
% A simple output formatter for the autodiff class.
disp([inputname(1), ' = ']);
fprintf('  (%17.17f, %17.17f)\n', ad.val, ad.der);
```

We can now input/output `autodiff` objects within the `MATLAB` environment:

```
>> a = autodiff(3), b = autodiff(2, 'variable')
a =
  (3.00000000000000000, 0.00000000000000000)
b =
  (2.00000000000000000, 1.00000000000000000)
```

First-order scalar AD - implementations

Arithmetic is easy to implement. Here is (matrix) multiplication:

```
function result = mtimes(a, b)
% Overloading the '*' operator.
[a, b] = cast(a, b);
val = a.val*b.val;
der = a.val*b.der + a.der*b.val;
result = autodiff(val, der);
```

First-order scalar AD - implementations

Arithmetic is easy to implement. Here is (matrix) multiplication:

```
function result = mtimes(a, b)
% Overloading the '*' operator.
[a, b] = cast(a, b);
val = a.val*b.val;
der = a.val*b.der + a.der*b.val;
result = autodiff(val, der);
```

And here is the logarithm:

```
function result = log(a)
% Overloading the 'log' operator.
if (a.val <= 0.0 )
    error('log undefined for non-positive arguments.');
end
val = log(a.val);
der = a.der/a.val;
result = autodiff(val, der);
```

First-order scalar AD - implementations

Here is a simple function that returns the derivative of a general function f at a given point x_0:

```
function dx = computeDerivative(fcnName, x0)
f  = inline(fcnName);
x  = autodiff(x0, 'variable');
dx = getDer(f(x));
```

First-order scalar AD - implementations

Here is a simple function that returns the derivative of a general function f at a given point x_0:

```
function dx = computeDerivative(fcnName, x0)
f  = inline(fcnName);
x  = autodiff(x0, 'variable');
dx = getDer(f(x));
```

A typical usage is

```
>> dfx = computeDerivative('(1 + x + exp(x))*sin(x)', 0)
dfx =
    2
>> dfx = computeDerivative('exp(sin(exp(cos(x) + 2*power(x,5))))', 1)
dfx =
    129.6681309181679
```

Great for checking your calculus homework

UPPSALA UNIVERSITET

First-order scalar AD - implementations

A more practical application is solving non-linear equations.

```
function y = newtonSearch(fcnName, x, tol)
f = inline(fcnName);
y = newtonStep(f, x);
while ( abs(x-y) > tol )
    x = y;
    y = newtonStep(f, x);
end
end

function Nx = newtonStep(f, x)
xx = autodiff(x, 'variable');
fx = f(xx);
Nx = x - getVal(fx)/getDer(fx);
end
```

First-order scalar AD - implementations

A more practical application is solving non-linear equations.

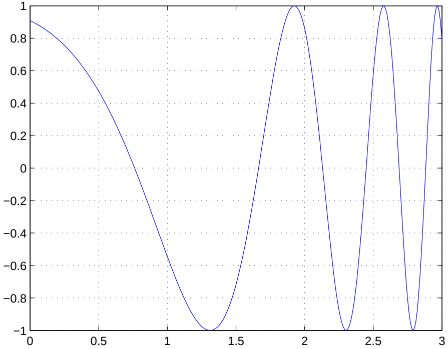
```
function y = newtonSearch(fcnName, x, tol)
f = inline(fcnName);
y = newtonStep(f, x);
while ( abs(x-y) > tol )
    x = y;
    y = newtonStep(f, x);
end
end

function Nx = newtonStep(f, x)
xx = autodiff(x, 'variable');
fx = f(xx);
Nx = x - getVal(fx)/getDer(fx);
end
```

Note that this function "hides" the AD from the user: all input/output is scalar.

First-order scalar AD - implementations

Some sample outputs:

```
>> x = newtonSearch('sin(exp(x) + 1)', 1, 1e-10)
x =
  0.761549782880894
>> x = newtonSearch('sin(exp(x) + 1)', 0, 1e-10)
x =
  2.131177121086310
```



UPPSALA UNIVERSITET

Second-order scalar AD - definitions

An arithmetic for differentiation

Extend the ideas to computations with ordered tripples of real numbers

$$\vec{u} = (u, u', u'').$$

Second-order scalar AD - definitions

An arithmetic for differentiation

Extend the ideas to computations with ordered tripples of real numbers

$$\vec{u} = (u, u', u'').$$

The third component holds the value of the second derivative $f''(x_0)$. As before, we assume that $f\colon \mathbb{R} \to \mathbb{R}$.

Second-order scalar AD - definitions

An arithmetic for differentiation

Extend the ideas to computations with ordered tripples of real numbers

$$\vec{u} = (u, u', u'').$$

The third component holds the value of the second derivative $f''(x_0)$. As before, we assume that $f\colon \mathbb{R} \to \mathbb{R}$.

Basic arithmetic

$$\begin{aligned}
\vec{u} + \vec{v} &= (u + v, u' + v', u'' + v'') \\
\vec{u} - \vec{v} &= (u - v, u' - v', u'' - v'') \\
\vec{u} \times \vec{v} &= (uv, uv' + u'v, uv'' + 2u'v' + u''v) \\
\vec{u} \div \vec{v} &= (u/v, (u' - (u/v)v')/v, (u'' - 2(u/v)'v' - (u/v)v'')/v),
\end{aligned}$$

where we demand that $v \neq 0$ when dividing.

Second-order scalar AD - definitions

Constants and the independent variable x are treated as before. Following the usual rules of differentiation, we define

$$\vec{x} = (x, 1, 0) \qquad \text{and} \qquad \vec{c} = (c, 0, 0).$$

UPPSALA UNIVERSITET

Second-order scalar AD - definitions

Constants and the independent variable x are treated as before. Following the usual rules of differentiation, we define

$$\vec{x} = (x, 1, 0) \qquad \text{and} \qquad \vec{c} = (c, 0, 0).$$

AD for standard functions

Similarly, standard functions are implemented via the chain rule:

$$\vec{g}(\vec{u}) = \vec{g}(u, u', u'') = (g(u), u'g'(u), u''g'(u) + (u')^2 g''(u)).$$

Second-order scalar AD - definitions

Constants and the independent variable x are treated as before. Following the usual rules of differentiation, we define

$$\vec{x} = (x, 1, 0) \qquad \text{and} \qquad \vec{c} = (c, 0, 0).$$

AD for standard functions

Similarly, standard functions are implemented via the chain rule:

$$\vec{g}(\vec{u}) = \vec{g}(u, u', u'') = (g(u), u'g'(u), u''g'(u) + (u')^2 g''(u)).$$

Applying this to some useful functions yields:

$$\begin{array}{lclcl} \sin \vec{u} & = & \sin(u, u', u'') & = & (\sin u, u' \cos u, u'' \cos u - (u')^2 \sin u) \\ e^{\vec{u}} & = & e^{(u, u', u'')} & = & (e^u, u'e^u, u''e^u + (u')^2 e^u) \end{array}$$

Straight-forward, but tedious!!!

Taylor series AD - definitions

A more effective (and perhaps less error-prone) approach to high-order automatic differentiation is obtained through the calculus of Taylor series:

$$f(x) = f_0 + f_1(x - x_0) + \cdots + f_k(x - x_0)^k + \ldots,$$

Here we use the notation $f_k = f_k(x_0) = f^{(k)}(x_0)/k!$

UPPSALA UNIVERSITET

Taylor series AD - definitions

A more effective (and perhaps less error-prone) approach to high-order automatic differentiation is obtained through the calculus of Taylor series:

$$f(x) = f_0 + f_1(x - x_0) + \cdots + f_k(x - x_0)^k + \ldots,$$

Here we use the notation $f_k = f_k(x_0) = f^{(k)}(x_0)/k!$

Basic arithmetic

$$\begin{aligned}
(f + g)_k &= f_k + g_k \\
(f - g)_k &= f_k - g_k \\
(f \times g)_k &= \sum_{i=0}^{k} f_i g_{k-i} \\
(f \div g)_k &= \frac{1}{g_0}\left(f_k - \sum_{i=0}^{k-1} (f \div g)_i g_{k-i}\right).
\end{aligned}$$

Taylor series AD - definitions

Proof: (formula for division).

By definition, we have

$$\sum_{k=0}^{\infty} f_k(x-x_0)^k \Big/ \sum_{k=0}^{\infty} g_k(x-x_0)^k = \sum_{k=0}^{\infty} (f \div g)_k (x-x_0)^k.$$

Taylor series AD - definitions

Proof: (formula for division).

By definition, we have

$$\sum_{k=0}^{\infty} f_k(x-x_0)^k \Big/ \sum_{k=0}^{\infty} g_k(x-x_0)^k = \sum_{k=0}^{\infty} (f \div g)_k (x-x_0)^k.$$

Multiplying both sides with the Taylor series for g produces

$$\sum_{k=0}^{\infty} f_k(x-x_0)^k = \sum_{k=0}^{\infty} (f \div g)_k (x-x_0)^k \sum_{k=0}^{\infty} g_k(x-x_0)^k,$$

Taylor series AD - definitions

Proof: (formula for division).

By definition, we have

$$\sum_{k=0}^{\infty} f_k(x-x_0)^k / \sum_{k=0}^{\infty} g_k(x-x_0)^k = \sum_{k=0}^{\infty} (f \div g)_k (x-x_0)^k.$$

Multiplying both sides with the Taylor series for g produces

$$\sum_{k=0}^{\infty} f_k(x-x_0)^k = \sum_{k=0}^{\infty} (f \div g)_k (x-x_0)^k \sum_{k=0}^{\infty} g_k(x-x_0)^k,$$

and, by the rule for multiplication, we have

$$f_k = \sum_{i=0}^{k} (f \div g)_i g_{k-i} = \sum_{i=0}^{k-1} (f \div g)_i g_{k-i} + (f \div g)_k g_0.$$

Taylor series AD - definitions

Proof: (formula for division).

By definition, we have

$$\sum_{k=0}^{\infty} f_k(x-x_0)^k \Big/ \sum_{k=0}^{\infty} g_k(x-x_0)^k = \sum_{k=0}^{\infty} (f \div g)_k (x-x_0)^k.$$

Multiplying both sides with the Taylor series for g produces

$$\sum_{k=0}^{\infty} f_k(x-x_0)^k = \sum_{k=0}^{\infty} (f \div g)_k (x-x_0)^k \sum_{k=0}^{\infty} g_k(x-x_0)^k,$$

and, by the rule for multiplication, we have

$$f_k = \sum_{i=0}^{k} (f \div g)_i g_{k-i} = \sum_{i=0}^{k-1} (f \div g)_i g_{k-i} + (f \div g)_k g_0.$$

Solving for $(f \div g)_k$ produces the desired result. $\square$

Taylor series AD - definitions

Constants and the independent variable x are treated as expected: seen as functions, these have particularly simple Taylor expansions:

$$
\begin{aligned}
x &= x_0 + 1 \cdot (x - x_0) + 0 \cdot (x - x_0)^2 + \cdots + 0 \cdot (x - x_0)^k + \ldots, \\
c &= c + 0 \cdot (x - x_0) + 0 \cdot (x - x_0)^2 + \cdots + 0 \cdot (x - x_0)^k + \ldots.
\end{aligned}
$$

Taylor series AD - definitions

Constants and the independent variable x are treated as expected: seen as functions, these have particularly simple Taylor expansions:

$$\begin{aligned} x &= x_0 + 1 \cdot (x - x_0) + 0 \cdot (x - x_0)^2 + \cdots + 0 \cdot (x - x_0)^k + \ldots, \\ c &= c + 0 \cdot (x - x_0) + 0 \cdot (x - x_0)^2 + \cdots + 0 \cdot (x - x_0)^k + \ldots. \end{aligned}$$

We now represent a function as a, possibly infinite, string of its Taylor coefficients:

$$f(x_0) \sim (f_0, f_1, \ldots, f_k, \ldots) \qquad f_k = f^{(k)}(x_0)/k.$$

Taylor series AD - definitions

Constants and the independent variable x are treated as expected: seen as functions, these have particularly simple Taylor expansions:

$$\begin{aligned} x &= x_0 + 1 \cdot (x - x_0) + 0 \cdot (x - x_0)^2 + \cdots + 0 \cdot (x - x_0)^k + \ldots, \\ c &= c + 0 \cdot (x - x_0) + 0 \cdot (x - x_0)^2 + \cdots + 0 \cdot (x - x_0)^k + \ldots. \end{aligned}$$

We now represent a function as a, possibly infinite, string of its Taylor coefficients:

$$f(x_0) \sim (f_0, f_1, \ldots, f_k, \ldots) \qquad f_k = f^{(k)}(x_0)/k.$$

Exercise

Write down the formal expression for $f \times f$ using the rule for multiplication. Using the appearing symmetry, find a more efficient formula for computing the square f^2 of a function f.

Taylor series AD - definitions

Taylor series AD for standard functions

Given a function g whose Taylor series is known, how do we compute the Taylor series for, say, e^g?

Taylor series AD - definitions

Taylor series AD for standard functions

Given a function g whose Taylor series is known, how do we compute the Taylor series for, say, e^g?

Let us formally write

$$g(x) = \sum_{k=0}^{\infty} g_k (x - x_0)^k \quad \text{and} \quad e^{g(x)} = \sum_{k=0}^{\infty} (e^g)_k (x - x_0)^k,$$

and use the fact that

$$\frac{d}{dx} e^{g(x)} = g'(x) e^{g(x)}. \tag{1}$$

Taylor series AD for standard functions

Given a function g whose Taylor series is known, how do we compute the Taylor series for, say, e^g?

Let us formally write

$$g(x) = \sum_{k=0}^{\infty} g_k (x - x_0)^k \quad \text{and} \quad e^{g(x)} = \sum_{k=0}^{\infty} (e^g)_k (x - x_0)^k,$$

and use the fact that

$$\frac{d}{dx} e^{g(x)} = g'(x) e^{g(x)}. \tag{1}$$

Plugging the formal expressions for $g'(x)$ and $e^{g(x)}$ into (1) produces

$$\sum_{k=1}^{\infty} k (e^g)_k (x - x_0)^{k-1} = \sum_{k=1}^{\infty} k g_k (x - x_0)^{k-1} \sum_{k=0}^{\infty} (e^g)_k (x - x_0)^k,$$

which, after multiplying both sides with $(x - x_0)$, becomes

Taylor series AD - definitions

$$\sum_{k=1}^{\infty} k(e^g)_k(x-x_0)^k = \sum_{k=1}^{\infty} k g_k (x-x_0)^k \sum_{k=0}^{\infty} (e^g)_k (x-x_0)^k.$$

$$\sum_{k=1}^{\infty} k(e^g)_k (x - x_0)^k = \sum_{k=1}^{\infty} k g_k (x - x_0)^k \sum_{k=0}^{\infty} (e^g)_k (x - x_0)^k.$$

Using the rule for multiplication then yields

$$k(e^g)_k = \sum_{i=1}^{k} i g_i (e^g)_{k-i} \qquad (k > 0).$$

Taylor series AD - definitions

$$\sum_{k=1}^{\infty} k(e^g)_k(x-x_0)^k = \sum_{k=1}^{\infty} k g_k (x-x_0)^k \sum_{k=0}^{\infty} (e^g)_k (x-x_0)^k.$$

Using the rule for multiplication then yields

$$k(e^g)_k = \sum_{i=1}^{k} i g_i (e^g)_{k-i} \qquad (k>0).$$

Since we know that the constant term is given by $(e^g)_0 = e^{g_0}$, we arrive at:

$$(e^g)_k = \begin{cases} e^{g_0} & \text{if } k=0, \\ \frac{1}{k}\sum_{i=1}^{k} i g_i (e^g)_{k-i} & \text{if } k>0. \end{cases}$$

Taylor series AD - definitions

More standard functions ($k > 0$)

$$
\begin{aligned}
(\ln g)_k &= \frac{1}{g_0}\left(g_k - \frac{1}{k}\sum_{i=1}^{k-1} i(\ln g)_i g_{k-i}\right) \\
(g^a)_k &= \frac{1}{g_0}\sum_{i=1}^{k}\Big(\frac{(a+1)i}{k} - 1\Big) g_i (g^a)_{k-i} \\
(\sin g)_k &= \frac{1}{k}\sum_{i=1}^{k} i g_i (\cos g)_{k-i} \\
(\cos g)_k &= -\frac{1}{k}\sum_{i=1}^{k} i g_i (\sin g)_{k-i}.
\end{aligned}
$$

UPPSALA UNIVERSITET

Taylor series AD - definitions

More standard functions $(k > 0)$

$$
\begin{aligned}
(\ln g)_k &= \frac{1}{g_0}\left(g_k - \frac{1}{k}\sum_{i=1}^{k-1} i(\ln g)_i g_{k-i}\right) \\
(g^a)_k &= \frac{1}{g_0}\sum_{i=1}^{k}\Big(\frac{(a+1)i}{k} - 1\Big) g_i (g^a)_{k-i} \\
(\sin g)_k &= \frac{1}{k}\sum_{i=1}^{k} i g_i (\cos g)_{k-i} \\
(\cos g)_k &= -\frac{1}{k}\sum_{i=1}^{k} i g_i (\sin g)_{k-i}.
\end{aligned}
$$

Remember that we always have $(f \circ g)_0 = f(g(x_0))$.

Taylor series AD - implementations

We begin by implementing a `taylor` class constructor in `MATLAB`.

```
function ts = taylor(a, N, str)
% A naive taylor constructor.
if nargin == 1
    if isa(a,'taylor')
      ts = a;
    else
      ts.coeff = a;
    end
elseif nargin == 3
    ts.coeff = zeros(1,N);
    if strcmp(str,'variable')
      ts.coeff(1) = a;  ts.coeff(2) = 1;
    elseif strcmp(str,'constant');
      ts.coeff(1) = a;  ts.coeff(2) = 0;
    end
end
 ts = class(ts, 'taylor');
```

Taylor series AD - implementations

Next, we implement the way to display the class objects:

```
function display(ts)
% A simple output formatter for the taylor class.
disp([inputname(1), ' = ']);
fprintf('[')
for i=1:length(ts.coeff)-1
  fprintf('%17.17f, ', ts.coeff(i));
end
fprintf('%17.17f]\n', ts.coeff(end));
```

Taylor series AD - implementations

Next, we implement the way to display the class objects:

```
function display(ts)
% A simple output formatter for the taylor class.
disp([inputname(1), ' = ']);
fprintf('[')
for i=1:length(ts.coeff)-1
  fprintf('%17.17f, ', ts.coeff(i));
end
fprintf('%17.17f]\n', ts.coeff(end));
```

We can now input/output `taylor` objects within the `MATLAB` environment:

```
>> x = taylor(1.5, 3, 'variable'), c = taylor(pi, 2, 'constant')
x =
  [1.50000000000000000, 1.00000000000000000, 0.00000000000000000]
c =
  [3.14159265358979312, 0.00000000000000000]
```

UPPSALA
UNIVERSITET

Taylor series AD - implementations

Here is an implementation for division

```
function result = mrdivide(a, b)
% Overloading the '/' operator.
[a, b] = cast(a, b);
if ( (b.coeff(1) == 0.0) )
    error('Denominator is zero.');
else
    N = length(a.coeff);
    coeff = zeros(1,N);
    coeff(1) = a.coeff(1)/b.coeff(1);
    for k=1:N-1
        sum = a.coeff(k+1);
        for i=0:k-1
            sum = sum - coeff(i+1)*b.coeff(k-i+1);
        end
        coeff(k+1) = sum/b.coeff(1);
    end
    result = taylor(coeff);
end
```

Taylor series AD - implementations

A very clean implementation for arbitrary order differentiation:

```
function dx = computeDerivative(fcnName, x0, order)
f  = inline(fcnName);
x  = taylor(x0, order+1, 'variable');
dx = getDer(f(x), order);
```

Taylor series AD - implementations

A very clean implementation for arbitrary order differentiation:

```
function dx = computeDerivative(fcnName, x0, order)
f  = inline(fcnName);
x  = taylor(x0, order+1, 'variable');
dx = getDer(f(x), order);
```

Here, `getDer` converts a Taylor coefficient into a derivative by multiplying it by the proper factorial.

Taylor series AD - implementations

A very clean implementation for arbitrary order differentiation:

```
function dx = computeDerivative(fcnName, x0, order)
f  = inline(fcnName);
x  = taylor(x0, order+1, 'variable');
dx = getDer(f(x), order);
```

Here, `getDer` converts a Taylor coefficient into a derivative by multiplying it by the proper factorial.

```
>> df100 = computeDerivative('exp(sin(exp(cos(x) + 2*x^5)))', -2, 100)
df100 =
        1.3783e+177
```

Taylor series AD - special implementations

```
function result = mrdivide(a, b)
% Overloading the ’/’ operator for l’Hopital’s rule.
[a, b] = cast(a, b);
a_ind = find(a.coeff,1,’first’); b_ind = find(b.coeff,1,’first’);
if (a_ind < b_ind)
    error(’Denominator is zero.’);
else
    a = taylor(a.coeff(b_ind:end)); b = taylor(b.coeff(b_ind:end));
    N = length(a.coeff);
    coeff = zeros(1,N);
    coeff(1) = a.coeff(1)/b.coeff(1);
    for k=1:N-1
        sum = a.coeff(k+1);
        for i=0:k-1
            sum = sum - coeff(i+1)*b.coeff(k-i+1);
        end
        coeff(k+1) = sum/b.coeff(1);
    end
    result = taylor(coeff);
end
```

Taylor series AD - special implementations

We can now handle removable singularities too:

Taylor series AD - special implementations

We can now handle removable singularities too:

```
>> x = taylor(0, 4, 'variable')
x =
    [0.0000000, 1.0000000, 0.0000000, 0.0000000]
>> y = sin(x)
y =
    [0.0000000, 1.0000000, 0.0000000,-0.1666666]
>> z = sin(x)/x
z =
    [1.0000000, 0.0000000,-0.1666666]
w = (exp(x)-1)/x
w =
    [1.0000000, 0.5000000, 0.1666666]
```

Taylor series AD - special implementations

We can now handle removable singularities too:

```
>> x = taylor(0, 4, 'variable')
x =
    [0.0000000, 1.0000000, 0.0000000, 0.0000000]
>> y = sin(x)
y =
    [0.0000000, 1.0000000, 0.0000000,-0.1666666]
>> z = sin(x)/x
z =
    [1.0000000, 0.0000000,-0.1666666]
w = (exp(x)-1)/x
w =
    [1.0000000, 0.5000000, 0.1666666]
```

Note that the resulting Taylor series are shortened accordingly.

UPPSALA UNIVERSITET