Matrix functions and automatic differentiation

Intermezzo: some words on automatic differentiation: a trick popular now in machine learning that allows one to compute derivatives of arbitrary functions on a computer.

Problem

Given code `function y = f(x)` to compute a function $f : \mathbb{R} \to \mathbb{R}$, how does one compute (or approximate) $f'(x)$ in a given point?

```
function y = f(x)
z = x * x;
w = x + 5;
y = z * w;
```

Numerical differentiation

First attempt: numerical differentiation: compute $g = \frac{f(x+h)-f(x)}{h}$, with a fixed $h > 0$.

Problem: Two sources of error:

- $g - f'(x) = \frac{1}{2}f''(\xi)h$ for a nearby point ξ (Taylor expansion).
- because of machine arithmetic, even with perfect code we can compute only $f(x)(1+\delta_1)$ and $f(x+h)(1+\delta_2)$ with $|\delta_i| < \mathbf{u}$. So the computed value $\tilde{g}$ of $g = \frac{f(x+h)-f(x)}{h}$ is affected by an error that we can bound with $\mathbf{u}\frac{|f(x)|+|f(x+h)|}{h}$

So the total error can only be bounded by

$$|\tilde{g} - f'(x)| \leq |\tfrac{1}{2}f''(\xi)|h + \mathbf{u}\frac{|f(x)| + |f(x+h)|}{h}.$$

Assuming $|\frac{1}{2}f''(\xi)|, |f(x)|, |f(x+h)| = \mathcal{O}(1)$, this error bound is minimized when $h \approx \mathbf{u}^{1/2}$ and is $\mathcal{O}(\mathbf{u}^{1/2})$.

Numerical derivatives can only be computed with accuracy $\mathcal{O}(\mathbf{u}^{1/2})$.

Example

```
>> x = 5
x =
     5
>> h = 1e-4; g = (f(x+h) - f(x)) / h
g =
     1.250020000097152e+02
% error ≈ 10^-4
>> h = 1e-8; g = (f(x+h) - f(x)) / h
g =
     1.250000025265763e+02
% error ≈ 10^-8
>> h = 1e-12; g = (f(x+h) - f(x)) / h
g =
     1.249986780749168e+02
% error ≈ 10^-4
```

Complex step differentiation

A similar trick: if f is holomorphic, and your code to compute it works also for complex inputs, then for $x \in \mathbb{R}$

$$f(x+ih) = f(x) + f'(x)ih - \frac{f''(x)}{2}h^2 + O(h^3),$$

so $g = \frac{\operatorname{Im} f(x+ih)}{h}$ is an approximation of the derivative $f'(x)$ with error $g - f'(x) = O(h^2)$.

This time, the total error is bounded by

$$|\tilde{g} - f'(x)| \leq \tfrac{1}{6}|f'''(\xi)|h^2 + |\operatorname{Im} f(x+ih)|\frac{\mathbf{u}}{h}$$

Error $\mathcal{O}(\mathbf{u})$ with $h = \mathcal{O}(h^{1/2})$, when the error on $\operatorname{Im} f(x+ih)$ is $\sim |\operatorname{Im} f(x+ih)|\mathbf{u}$ (but if real/imaginary parts are 'mixed' in computation, it is $\sim |f(x+ih)|\mathbf{u}$).

Key idea

We exploited the fact that our code runs also for a more general type (complex numbers vs. reals) to obtain a better bound.

Example

```
>> x = 5
x =
     5
>> h = 1e-4; g = imag(f(x+1i*h) - f(x)) / h
g =
     1.249999999900000e+02
>> h = 1e-8; g = imag(f(x+1i*h) - f(x)) / h
g =
     1.250000000000000e+02
>> h = 1e-12; g = imag(f(x+1i*h) - f(x)) / h
g =
     1.250000000000000e+02
```

Automatic differentiation via matrix functions

Suppose our code works also for matrix arguments x (which we can achieve with some changes):

```
function y = f(x)
n = size(x, 1)
z = x * x;
w = x + 5*eye(n);
y = z * w;
```

Then,

$$f\left(\begin{bmatrix} \lambda & 1 & \\ & \lambda & 1 \\ & & \lambda \end{bmatrix}\right) = \begin{bmatrix} f(\lambda) & f'(\lambda) & \frac{f''(\lambda)}{2} \\ & f(\lambda) & f'(\lambda) \\ & & f(\lambda) \end{bmatrix}.$$

No "small h" and subtractions are needed this time $\implies$ the derivative can be computed with error $\mathcal{O}(\mathbf{u})$.

Automatic differentiation

This trick (known as automatic differentiation) computes derivatives up to machine precision error $O(\mathbf{u})$.

It is something fundamentally different from numerical differentiation; it is more similar to symbolic differentiation with a computer algebra system, but easier to do algorithmically.

This works in much greater generality, for instance with loops, conditionals, and more complicated functions:

```
function y = somefunction(x)
a = x*x + 1;
z = 2 / a;
while z < 5
  z = z^2;
end
y = exp(z);
```

This function is not continuous at "decision points" (when $z = 5$ at some iteration of the `while`).

```
function y = somefunction(x)
n = size(x, 1);
a = x*x + eye(n);
z = 2 * inv(a);
while z(1,1) < 5
  z = z^2;
end
y = expm(z);
```

What is going on

Actually, we do not need matrices here: all operations are on triangular Toeplitz matrices, so we can just store the first row.

In essence, this is propagating Taylor expansions: instead of the input x, we start from $x + \varepsilon$, and whenever we compute a variable we compute the first n coefficients of its Taylor expansion alongside it; for instance given code

```
function y = f(x) % input: x=5
z = x * x; % z is 25
w = x + 5; % w is 10
y = z * w; % y is 250
```

we can compute two derivatives ($n = 3$) alongside it:

```
function y = f(x) % input: x = 5 + ε = 5 + 1ε + 0ε^2 + O(ε^3)
z = x * x; % z is (5 + 1ε + 0ε^2 + O(ε^3))(5 + 1ε + 0ε^2 + O(ε^3))
% z = 25 + 10ε + 1ε^2 + O(ε^3)
w = x + 5; % w is (5 + 1ε + 0ε^2 + O(ε^3)) + 5
% w = 10 + 1ε + 0ε^2 + O(ε^3)
y = z * w; % y is (25 + 10ε + 1ε^2 + O(ε^3))(10 + 1ε + 0ε^2 + O(ε^3))
% y = 250 + 125ε + 20ε^2 + O(ε^3)
```

From this Taylor expansion we can read off the first two derivatives of $y = f(x)$ in $x = 5$.

How do we get the computer to do all this automatically without writing ad-hoc code? By defining a class Taylor that contains a length-3 vector as its data member.

```
function y = f(x) % input: x = Taylor[5 1 0]
z = x * x; % z = Taylor[5 1 0] * Taylor[5 1 0]
% z = Taylor[25 10 1]
w = x + 5; % w = Taylor[5 1 0] + Taylor[5 0 0]
% w = Taylor[10 1 0]
y = z * w; % y = Taylor[25 10 1] * Taylor[10 1 0]
% y = Taylor[250 125 20]
```

Rules for operations:

- A real a is converted to `Taylor[a 0 0]`
- `Taylor[a0, a1, a2] + Taylor[b0, b1, b2]`
  `= Taylor[a0+b0, a1+b1, a2+b2]`
- `Taylor[a0, a1, a2] * Taylor[b0, b1, b2]`
  `= Taylor[a0*b0, a1*b0+a0*b1, a2*b0+a1*b1+a0*b2]`

Matlab is not the best language in the world, but it can do OOP, too.

In Matlab

```
classdef Taylor
   properties
      coeffs %length-3 vector
   end
   methods
      function obj = Taylor(v)
         obj.coeffs = v;
      end
      function c = plus(a, b)
         if isa(b, 'double'), b = Taylor([b 0 0]); end
         c = Taylor(a.coeffs + b.coeffs);
      end
      function c = mtimes(a, b)
         c = Taylor([a.coeffs(1)*b.coeffs(1), a.coeffs(1)*b
      end
   end
end
```

Automatic differentiation, generically

For any elementary operation $z = f(a, b, \ldots)$, we can update derivatives alongside according to composite-function differentiation rules:

$$
\begin{aligned}
z' &= \frac{\partial f}{\partial a} a' + \frac{\partial f}{\partial b} b' + \ldots \\
z'' &= \frac{\partial^2 f}{\partial a^2} (a')^2 + \frac{\partial f}{\partial a} a'' + \frac{\partial^2 f}{\partial b^2} (b')^2 + \frac{\partial f}{\partial b} b'' + \ldots \\
\ldots & \quad \ldots
\end{aligned}
$$

(The formulas get lengthy if we want higher derivatives.)

As long as we can do this for each operation appearing in our code $(ab, a/b, \exp(a), \ldots)$, by overloading these functions for our type `Taylor`, we can effectively compute derivatives algorithmically.

Again, the key is code that supports different types and operator overloading.

Special case: dual numbers

The most common case is when one only needs one derivative.

A convenient formalism for this case: dual numbers.

- Replace each quantity a with $a + \varepsilon a'$.
- Operations are performed with usual algebraic rules plus $\varepsilon^2 = 0$.
- `a * b` becomes $(a + \varepsilon a')(b + \varepsilon b') = ab + (a'b + ab')\varepsilon$.
- The input variable x becomes $x + \varepsilon 1$.

Various ways to think about it:

- ε is "infinitesimal".
- Operations in $\mathbb{R}[\varepsilon]/(\varepsilon^2)$.
- $\varepsilon = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix}$.

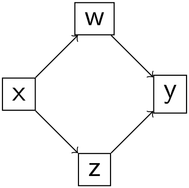
...but in the end they are implemented as length-2 vectors `[a a']`.

What machine learning does

This is called forward mode of automatic differentiation. There is also a reverse mode which is more popular in some contexts; most notably machine learning, where it is known as back-propagation).

General idea: After having computed $y = f(x)$, revisit your code backwards line-by-line and for each intermediate variable a determine iteratively $\frac{\partial y}{\partial a}$ (and higher derivatives if needed).

Reverse-mode: example




```
function y = f(x) % input: x=5
z = x * x; % ∂z/∂x = 2x = 10
w = x + 5; % ∂w/∂x = 1
y = z * w; % ∂y/∂w = z = 25, ∂y/∂z = w = 10
```

We can work our way upwards and compute starting from the end

$$\frac{\partial y}{\partial w} = z = 25, \quad \frac{\partial y}{\partial z} = w = 10,$$

$$\frac{\partial y}{\partial x} = \frac{\partial y}{\partial w}\frac{\partial w}{\partial x} + \frac{\partial y}{\partial z}\frac{\partial z}{\partial x} = 25 \cdot 1 + 10 \cdot 10 = 125.$$

Comments

This manipulation requires more complicated transformations to the code than forward-mode: one must build a dependency graph, and 're-interpret' code backwards. Operator overloading is not sufficient.

While they are equivalent for scalar functions, they behave differently if one tries to extend them to $f : \mathbb{R}^n \to \mathbb{R}^m$.

Quick result: for a function $f : \mathbb{R}^n \to \mathbb{R}^m$, computing J_f (all derivatives) is faster with forward mode if $n \ll m$ (many outputs), and with reverse mode if $n \gg m$ (many inputs).