Matrix functions and automatic differentiation

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

Problem

How does one compute derivatives of an arbitrary (computable) function f on a computer?

First attempt: numerical differentiation $f'(x) \approx \frac{f(x+h)-f(x)}{h}$.
Problem: It is an approximated method. Even for "tame" functiones, h too small $\implies$ cancellation in the subtraction.

Error $O(\mathbf{u}/h)$ from the computation of the numerator, so the best we can do is error $O(\mathbf{u}^{1/2})$ with $h = \mathbf{u}^{1/2}$.

Matrix functions and automatic differentiation

Idea

- Take a function, e.g. $f(x) = \frac{x^2+5}{1+\exp(x)}$.
- Write "matrix-friendly" code to compute it:

```
n = size(X, 1);
Y = inv(eye(n) + expm(X)) * (X*X + 5*eye(n));
```

  (`inv` used here for clarity; normally \ is better.)
  (Note that the matrices $I + \exp(X)$ and $X^2 + 5I$ commute, so the order in the product does not matter as long as my expression contains only functions of a single matrix X.)
- Then, one can read derivatives of f off functions of Jordan blocks.

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

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.

We can achieve it just by rewriting code to be matrix-friendly. (See next example)

```
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);
```

Demistifying automatic differentiation

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: at each step we store (e.g. with $n = 3$) Taylor expansions in x for each quantity appearing in the code:

$$\mathrm{a} : \begin{bmatrix} a(x) & a'(x) & a''(x) \end{bmatrix}, \quad \mathrm{b} : \begin{bmatrix} b(x) & b'(x) & b''(x) \end{bmatrix},$$

and we update them according to various operations: for instance, `a * b` becomes

$$\begin{bmatrix} a & a' & a'' \end{bmatrix} * \begin{bmatrix} b & b' & b'' \end{bmatrix} =$$

$$\begin{bmatrix} ab & a'b + ab' & a''b + 2a'b' + ab'' \end{bmatrix}.$$

This could be implemented with a special 'Taylor' class and operator overriding.

Special case: dual numbers

A different formalism for $n = 2$ (first derivative): 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 $\mathbb{R}[\varepsilon]/(\varepsilon^2)$.
- $\varepsilon = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix}$.

Complex step differentiation

Another cheap trick: if you have a real holomorphic function $f : \mathbb{R} \to \mathbb{R}$, and code to compute it 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

$$f'(x) = \frac{\mathrm{Im}(f(x + ih))}{h} + O(h^2).$$

Avoids the subtraction, and achieves one more order of accuracy.

If one sets $h = \mathbf{u}^{1/3}$, error of the order of $\mathbf{u}^{2/3}$.

In practice, the actual accuracy obtained depends on how exactly complex arithmetic is used in computing $f(x)$.

What machine learning does

This is called forward mode of automatic differentiation. There is also a reverse mode which is more popular in some field (it is called back-propagation in machine learning).

General idea: After having computed $f(x)$, "roll back" the code and (starting from the last line) determine iteratively the contribution of each intermediate variable to $f'(x)$.

Requires more complicated transformations to the code to be implemented. We will not see details.

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