Differentiation

We now get to our first computational problem: given a function, how can we approximate its derivative at a point? Before we begin, we must be clear what a "function" is. Consider three possible scenarios:

1. *Black-box function*: Consider a floating-point valued function $f^{\text{FP}} : D \to F$ where $D \subset F \equiv F_{\sigma,Q,S}$ (e.g., we are given a double precision function that takes in a `Float64` and returns another `Float64`) which we only know *pointwise*. This is the situation if we have a function that relies on a compiled C library, which composes floating point arithmetic operations. Since F is a discrete set such an f^{FP} cannot be differentiable in a rigorous way, therefore we need to assume that f^{FP} approximates a differentiable function f with controlled error in order to state anything precise.
2. *Generic function*: Consider a function that is a formula (or, equivalentally, a *piece of code*) that we can evaluate it on arbitrary types, including custom types that we create. An example is a polynomial: $p(x) = p_0 + p_1 x + \cdots + p_n x^n$ which can be evaluated for x in the reals, complexes, or any other ring. More generally, if we have a function defined in Julia that does not call any C libraries it can be evaluated on different types. For analysis we typically consider both a differentiable function $f : D \to \mathbb{R}$ for $D \subset \mathbb{R}$, which would be what one would have if we could evaluate a function exactly using real arithmetic, and $f^{\text{FP}} : D \cap F \to F$, which is what we actually compute when evaluating the function using floating point arithmetic.
3. *Graph function*: The function is built by composing different basic "kernels" with known differentiability properties. We won't consider this situation in this module, though it is the model used by Python machine learning toolbox's like PyTorch and TensorFlow.

We discuss the following techniques:

1. Finite-differences: Use the definition of a derivative that one learns in calculus to approximate its value. Unfortunately, the round-off errors of floating point arithmetic typically limit its accuracy.
2. Dual numbers (forward-mode automatic differentiation): Define a special type that when applied to a function computes its derivative. Mathematically, this uses *dual numbers*, which are analoguous to complex numbers.

Note there are other techniques for differentiation that we don't discuss:

1. Symbolic differentiation: A tree is built representing a formula which is differentiated using the product and chain rule.
2. Adjoints and back-propagation (reverse-mode automatic differentiation): This is similar to symbolic differentiation but automated, to build up a tape of operations that tracks interdependencies. It's outside the scope of this module but is computationally preferred for computing gradients of large dimensional functions which is critical in machine learning.
3. Interpolation and differentiation: We can also differentiate functions *globally*, that is, in an interval instead of only a single point, which will be discussed later in the

module.

In [ ]:
```
using ColorBitstring
```

1. Finite-differences

The definition

$$f'(x) = \lim_{h\to 0} \frac{f(x+h) - f(x)}{h}$$

tells us that

$$f'(x) \approx \frac{f(x+h) - f(x)}{h}$$

provided that h is sufficiently small.

It's important to note that approximation uses only the *black-box* notion of a function but to obtain bounds we need more.

If we know a bound on $f''(x)$ then Taylor's theorem tells us a precise bound:

Proposition The error in approximating the derivative using finite differences is

$$\left| f'(x) - \frac{f(x+h) - f(x)}{h} \right| \leq \frac{M}{2} h$$

where $M = \sup_{x \leq t \leq x+h} |f''(t)|$.

Proof Follows immediately from Taylor's theorem:

$$f(x+h) = f(x) + f'(x)h + \frac{f''(t)}{2}h^2$$

for some $x \leq t \leq x+h$.

■

There are also alternative versions of finite differences. Leftside finite-differences:

$$f'(x) \approx \frac{f(x) - f(x-h)}{h}$$

and central differences:

$$f'(x) \approx \frac{f(x+h/2) - f(x-h/2)}{h}$$

Composing these approximations is useful for higher-order derivatives as we discuss in the problem sheet.

Note this is assuming *real arithmetic*, the answer is drastically different with *floating point arithmetic*.

Does finite-differences work with floating point arithmetic?

Let's try differentiating two simple polynomials $f(x) = 1 + x + x^2$ and $g(x) = 1 + x/3 + x^2$ by applying the finite-difference approximation to their floating point implementations f^{FP} and g^{FP}:

In [ ]:
```
f = x -> 1 + x + x^2      # we treat f and g as black-boxs
g = x -> 1 + x/3 + x^2
h = 0.000001
(f(h)-f(0))/h, (g(h)-g(0))/h
```

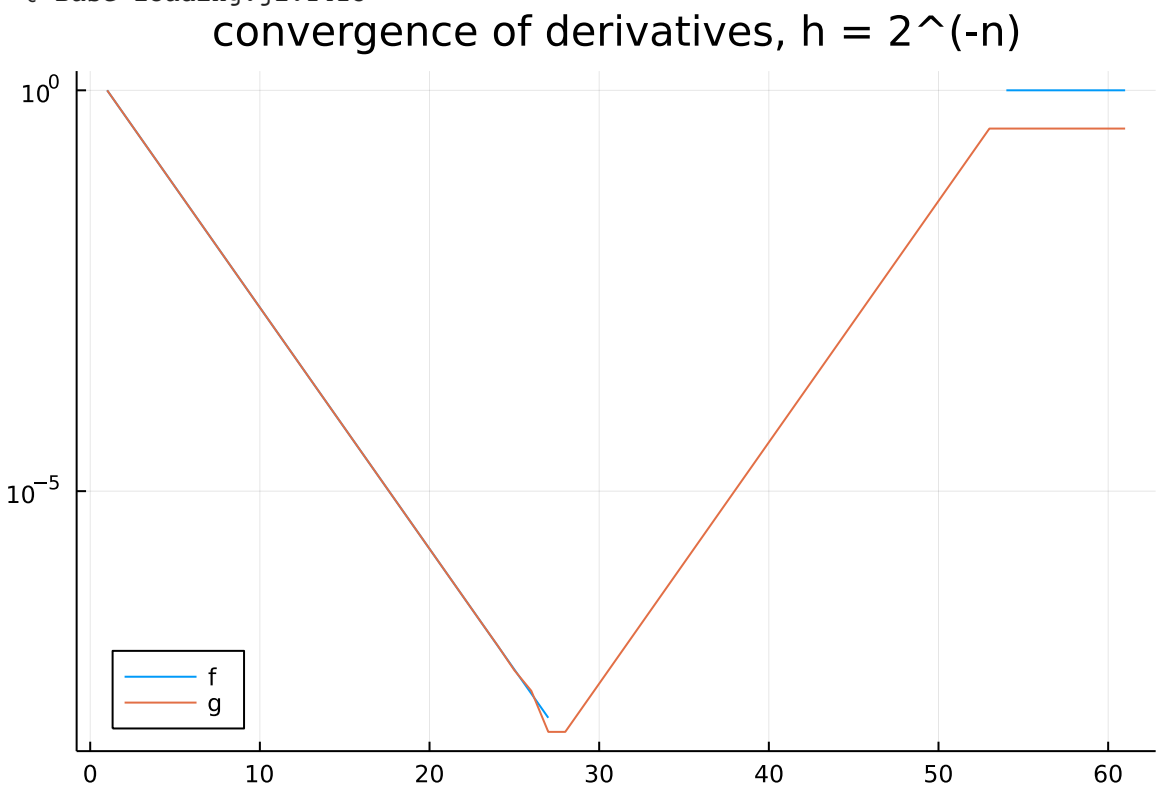
Out[ ]:
```
(1.000001000006634, 0.33333433346882657)
```

Both seem to roughly approximate the true derivatives (1 and $1/3$). We can do a plot to see how fast the error goes down as we let h become small.

In [ ]:
```
using Plots
h = 2.0 .^ (0:-1:-60)  # [1,1/2,1/4,…]
nanabs = x -> iszero(x) ? NaN : abs(x) # avoid 0's in log scale plot
plot(nanabs.((f.(h) .- f(0)) ./ h .- 1); yscale=:log10, title="convergence
plot!(abs.((g.(h) .- g(0)) ./ h .- 1/3); yscale=:log10, label = "g")
```

```
[ Info: Precompiling Plots [91a5bcdd-55d7-5caf-9e0b-520d859cae80]
[ @ Base loading.jl:1423
```

Out[ ]:



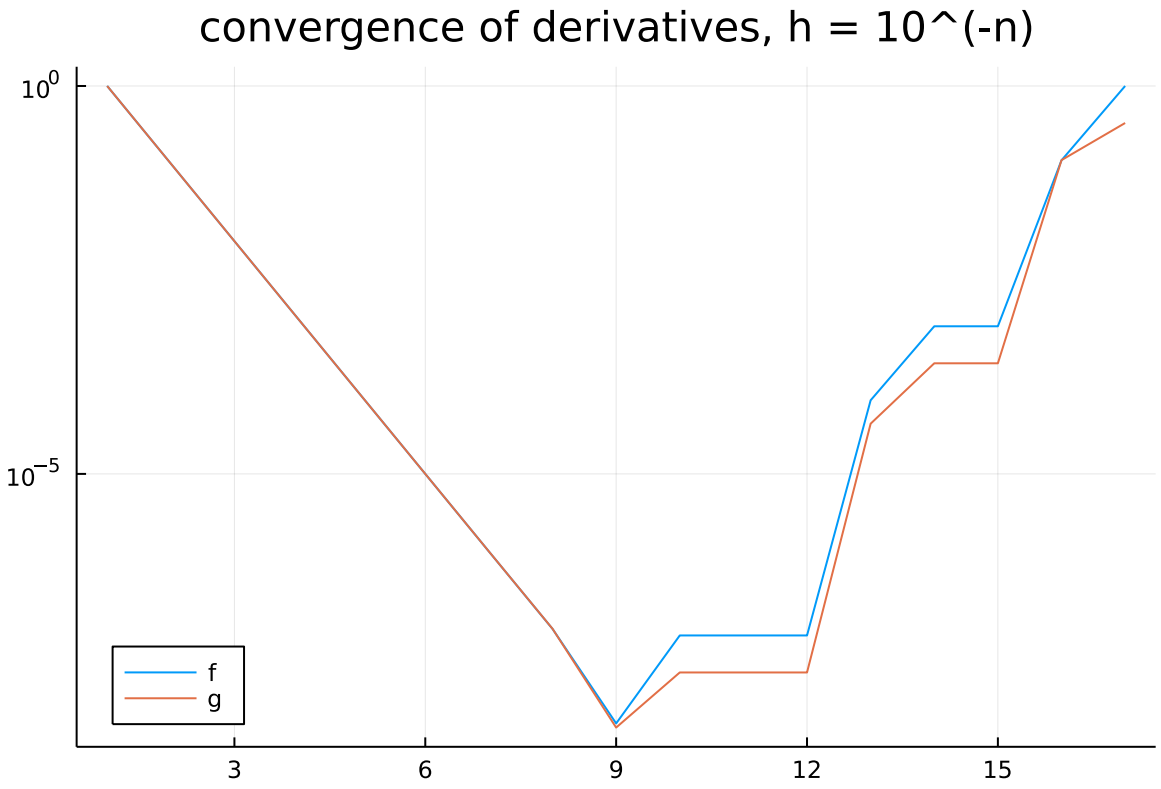

```
[ Info: Precompiling GR_jll [d2c73de3-f751-5644-a686-071e5b155ba9]
[ @ Base loading.jl:1423
```

In the case of f it is a success: we approximate the true derivative *exactly* provided we take $h = 2^{-n}$ for $26 < n \leq 52$. But for g it is a huge failure: the approximation starts to converge, but then diverges exponentially fast, before levelling off!

It is clear that f is extremely special. Most functions will behave like g, and had we not taken h to be a power of two we also see divergence for differentiating f:

In [ ]:
```
h = 10.0 .^ (0:-1:-16)  # [1,1/10,1/100,…]
plot(abs.((f.(h) .- f(0)) ./ h .- 1); yscale=:log10, title="convergence of
plot!(abs.((g.(h) .- g(0)) ./ h .- 1/3); yscale=:log10, label = "g")
```

Out[ ]:



For these two simple examples, we can understand why we see very different behaviour.

Example (convergence(?) of finite difference) Consider differentiating $f(x) = 1 + x + x^2$ at 0 with $h = 2^{-n}$. We consider 3 different cases with different behaviour, where S is the number of significand bits:

1. $0 \leq n \leq S/2$
2. $S/2 < n \leq S$
3. $S \leq n$

Note that $f^{\text{FP}}(0) = f(0) = 1$. Thus we wish to understand the error in approximating $f'(0) = 1$ by

$$(f^{\text{FP}}(h) \ominus 1) \oslash h \qquad \text{where} \qquad f^{\text{FP}}(x) = 1 \oplus x \oplus x \otimes x.$$

Case 1 ($0 \leq n \leq S/2$): note that $f^{\text{FP}}(h) = f(h) = 1 + 2^{-n} + 2^{-2n}$ as each computation is precisely a floating point number (hence no rounding). We can see this in half-precision, with $n = 3$ we have a 1 in the 3rd and 6th decimal place:

In [ ]:
```
S = 10 # 10 significant bits
n = 3  # 3 ≤ S/2 = 5
h = Float16(2)^(-n)
printbits(f(h))
```

```
0011110010010000
```

Subtracting 1 and dividing by h will also be exact, hence we get

$$(f^{\text{FP}}(h) \ominus 1) \oslash h = 1 + 2^{-n}$$

which shows exponential convergence.

Case 2 ($S/2 < n \leq S$): Now we have (using round-to-nearest)

$$f^{\mathrm{FP}}(h) = (1 + 2^{-n}) \oplus 2^{-2n} = 1 + 2^{-n}$$

Then

$$(f^{\mathrm{FP}}(h) \ominus 1) \oslash h = 1 = f'(0)$$

We have actually performed better than true real arithmetic and converged without a limit!

Case 3 ($n > S$): If we take n too large, then $1 \oplus h = 1$ and we have $f^{\mathrm{FP}}(h) = 1$, that is and

$$(f^{\mathrm{FP}}(h) \ominus 1) \oslash h = 0 \neq f'(0)$$

Example (divergence of finite difference) Consider differentiating $g(x) = 1 + x/3 + x^2$ at 0 with $h = 2^{-n}$ and assume n is even for simplicity and consider half-precision with $S = 10$. Note that $g^{\mathrm{FP}}(0) = g(0) = 1$. Recall

$$h \oslash 3 = 2^{-n-2} * (1.0101010101)_2$$

Note we lose two bits each time in the computation of $1 \oplus (h \oslash 3)$:

In [ ]:
```
n = 0; h = Float16(2)^(-n); printlnbits(1 + h/3)
n = 2; h = Float16(2)^(-n); printlnbits(1 + h/3)
n = 4; h = Float16(2)^(-n); printlnbits(1 + h/3)
n = 8; h = Float16(2)^(-n); printlnbits(1 + h/3)
```

```
0011110101010101
0011110001010101
0011110000010101
0011110000000001
```

It follows if $S/2 < n < S$ that

$$1 \oplus (h \oslash 3) = 1 + h/3 - 2^{-10}/3$$

Therefore

$$(g^{\mathrm{FP}}(h) \ominus 1) \oslash h = 1/3 - 2^{n-10}/3$$

Thus the error grows exponentially with n.

If $n \geq S$ then $1 \oplus (h \oslash 3) = 1$ and we have

$$(g^{\mathrm{FP}}(h) \ominus 1) \oslash h = 0$$

Bounding the error

We can bound the error using the bounds on floating point arithmetic.

Theorem (finite-difference error bound) Let f be twice-differentiable in a neighbourhood of x and assume that $f^{\mathrm{FP}}(x) = f(x) + \delta_x^f$ has uniform absolute accuracy in that neighbourhood, that is:

$$|\delta_x^f| \leq c\epsilon_{\mathrm{m}}$$

for a fixed constant c. Assume for simplicity $h = 2^{-n}$ where $n \leq S$ and $|x| \leq 1$. Then the finite-difference approximation satisfies

$$(f^{\mathrm{FP}}(x+h) \ominus f^{\mathrm{FP}}(x)) \oslash h = f'(x) + \delta_{x,h}^{\mathrm{FD}}$$

where

$$|\delta_{x,h}^{\mathrm{FD}}| \leq \frac{|f'(x)|}{2}\epsilon_{\mathrm{m}} + Mh + \frac{4c\epsilon_{\mathrm{m}}}{h}$$

for $M = \sup_{x \leq t \leq x+h} |f''(t)|$.

Proof

We have (noting by our assumptions $x \oplus h = x + h$ and that dividing by h will only change the exponent so is exact)

$$\begin{aligned}(f^{\mathrm{FP}}(x+h) \ominus f^{\mathrm{FP}}(x)) \oslash h &= \frac{f(x+h) + \delta_{x+h}^f - f(x) - \delta_x^f}{h}(1+\delta_1) \\ &= \frac{f(x+h) - f(x)}{h}(1+\delta_1) + \frac{\delta_{x+h}^f - \delta_x^f}{h}(1+\delta_1)\end{aligned}$$

where $|\delta_1| \leq \epsilon_{\mathrm{m}}/2$. Applying Taylor's theorem we get

$$(f^{\mathrm{FP}}(x+h) \ominus f^{\mathrm{FP}}(x)) \oslash h = f'(x) + \underbrace{f'(x)\delta_1 + \frac{f''(t)}{2}h(1+\delta_1) + \frac{\delta_{x+h}^f - \delta_x^f}{h}(1+\delta_1)}_{\delta_{x,h}^{\mathrm{FD}}}$$

The bound then follows, using the very pessimistic bound $|1+\delta_1| \leq 2$.

∎

The three-terms of this bound tell us a story: the first term is a fixed (small) error, the second term tends to zero as $h \to 0$, while the last term grows like ϵ_{m}/h as $h \to 0$. Thus we observe convergence while the second term dominates, until the last term takes over. Of course, a bad upper bound is not the same as a proof that something grows, but it is a good indication of what happens *in general* and suffices to motivate the following heuristic to balance the two sources of errors:

Heuristic (finite-difference with floating-point step) Choose h proportional to $\sqrt{\epsilon_{\mathrm{m}}}$ in finite-differences.

In the case of double precision $\sqrt{\epsilon_{\mathrm{m}}} \approx 1.5 \times 10^{-8}$, which is close to when the observed error begins to increase in our examples.

Remark While finite differences is of debatable utility for computing derivatives, it is extremely effective in building methods for solving differential equations, as we shall see later. It is also very useful as a "sanity check" if one wants something to compare with for other numerical methods for differentiation.

2. Dual numbers (Forward-mode automatic differentiation)

Automatic differentiation consists of applying functions to special types that determine the derivatives. Here we do so via *dual numbers*.

Definition (Dual numbers) Dual numbers $\mathbb{D}$ are a commutative ring over the reals generated by 1 and ϵ such that $\epsilon^2 = 0$. Dual numbers are typically written as $a + b\epsilon$ where a and b are real.

This is very much analoguous to complex numbers, which are a field generated by 1 and i such that $\mathrm{i}^2 = -1$. Compare multiplication of each number type:

$$(a + b\mathrm{i})(c + d\mathrm{i}) = ac + (bc + ad)\mathrm{i} + bd\mathrm{i}^2 = ac - bd + (bc + ad)\mathrm{i}$$
$$(a + b\epsilon)(c + d\epsilon) = ac + (bc + ad)\epsilon + bd\epsilon^2 = ac + (bc + ad)\epsilon$$

And just as we view $\mathbb{R} \subset \mathbb{C}$ by equating $a \in \mathbb{R}$ with $a + 0\mathrm{i} \in \mathbb{C}$, we can view $\mathbb{R} \subset \mathbb{D}$ by equating $a \in \mathbb{R}$ with $a + 0\epsilon \in \mathbb{D}$.

Connection with differentiation

Applying a polynomial to a dual number $a + b\epsilon$ tells us the derivative at a:

Theorem (polynomials on dual numbers) Suppose p is a polynomial. Then

$$p(a + b\epsilon) = p(a) + bp'(a)\epsilon$$

Proof

It suffices to consider $p(x) = x^n$ for $n \geq 1$ as other polynomials follow from linearity. We proceed by induction: The case $n = 1$ is trivial. For $n > 1$ we have

$$(a + b\epsilon)^n = (a + b\epsilon)(a + b\epsilon)^{n-1} = (a + b\epsilon)(a^{n-1} + (n-1)ba^{n-2}\epsilon) = a^n + bna^{n-1}\epsilon.$$

∎

We can extend real-valued differentiable functions to dual numbers in a similar manner. First, consider a standard function with a Taylor series (e.g. $\cos$, $\sin$, $\exp$, etc.)

$$f(x) = \sum_{k=0}^{\infty} f_k x^k$$

so that a is inside the radius of convergence. This leads naturally to a definition on dual numbers:

$$\begin{aligned} f(a + b\epsilon) &= \sum_{k=0}^{\infty} f_k (a + b\epsilon)^k = \sum_{k=0}^{\infty} f_k (a^k + ka^{k-1}b\epsilon) = \sum_{k=0}^{\infty} f_k a^k + \sum_{k=0}^{\infty} f_k k a^{k-1} b\epsilon \\ &= f(a) + bf'(a)\epsilon \end{aligned}$$

More generally, given a differentiable function we can extend it to dual numbers:

Definition (dual extension) Suppose a real-valued function f is differentiable at a. If

$$f(a+b\epsilon) = f(a) + bf'(a)\epsilon$$

then we say that it is a *dual extension at* a.

Thus, for basic functions we have natural extensions:

$$\begin{aligned}
\exp(a+b\epsilon) &:= \exp(a) + b\exp(a)\epsilon \\
\sin(a+b\epsilon) &:= \sin(a) + b\cos(a)\epsilon \\
\cos(a+b\epsilon) &:= \cos(a) - b\sin(a)\epsilon \\
\log(a+b\epsilon) &:= \log(a) + \frac{b}{a}\epsilon \\
\sqrt{a+b\epsilon} &:= \sqrt{a} + \frac{b}{2\sqrt{a}}\epsilon \\
|a+b\epsilon| &:= |a| + b\operatorname{sign} a\,\epsilon
\end{aligned}$$

provided the function is differentiable at a. Note the last example does not have a convergent Taylor series (at 0) but we can still extend it where it is differentiable.

Going further, we can add, multiply, and compose such functions:

Lemma (product and chain rule) If f is a dual extension at $g(a)$ and g is a dual extension at a, then $q(x) := f(g(x))$ is a dual extension at a. If f and g are dual extensions at a then $r(x) := f(x)g(x)$ is also dual extensions at a. In other words:

$$\begin{aligned}
q(a+b\epsilon) &= q(a) + bq'(a)\epsilon \\
r(a+b\epsilon) &= r(a) + br'(a)\epsilon
\end{aligned}$$

Proof For q it follows immediately:

$$q(a+b\epsilon) = f(g(a+b\epsilon)) = f(g(a)+bg'(a)\epsilon) = f(g(a)) + bg'(a)f'(g(a))\epsilon = q(a) + bq$$

For r we have

$$r(a+b\epsilon) = f(a+b\epsilon)g(a+b\epsilon) = (f(a)+bf'(a)\epsilon)(g(a)+bg'(a)\epsilon) = f(a)g(a) + b(f'($$

∎

A simple corollary is that any function defined in terms of addition, multiplication, composition, etc. of functions that are dual with differentiation will be differentiable via dual numbers.

Example (differentiating non-polynomial)

Consider $f(x) = \exp(x^2 + \mathrm{e}^x)$ by evaluating on the duals:

$$f(1+\epsilon) = \exp(1 + 2\epsilon + \mathrm{e} + \mathrm{e}\epsilon) = \exp(1+\mathrm{e}) + \exp(1+\mathrm{e})(2+\mathrm{e})\epsilon$$

and therefore we deduce that

$$f'(1) = \exp(1+\mathrm{e})(2+\mathrm{e}).$$

Implementation as a special type

We now consider a simple implementation of dual numbers that works on general polynomials:

In [ ]:
```
# Dual(a,b) represents a + b*ϵ
struct Dual{T}
    a::T
    b::T
end

# Dual(a) represents a + 0*ϵ
Dual(a::Real) = Dual(a, zero(a)) # for real numbers we use a + 0ϵ

# Allow for a + b*ϵ syntax
const ϵ = Dual(0, 1)

import Base: +, *, -, /, ^, zero, exp

# support polynomials like 1 + x, x - 1, 2x or x*2 by reducing to Dual
+(x::Real, y::Dual) = Dual(x) + y
+(x::Dual, y::Real) = x + Dual(y)
-(x::Real, y::Dual) = Dual(x) - y
-(x::Dual, y::Real) = x - Dual(y)
*(x::Real, y::Dual) = Dual(x) * y
*(x::Dual, y::Real) = x * Dual(y)

# support x/2 (but not yet division of duals)
/(x::Dual, k::Real) = Dual(x.a/k, x.b/k)

# a simple recursive function to support x^2, x^3, etc.
function ^(x::Dual, k::Integer)
    if k < 0
        error("Not implemented")
    elseif k == 1
        x
    else
        x^(k-1) * x
    end
end

# Algebraic operationds for duals
-(x::Dual) = Dual(-x.a, -x.b)
+(x::Dual, y::Dual) = Dual(x.a + y.a, x.b + y.b)
-(x::Dual, y::Dual) = Dual(x.a - y.a, x.b - y.b)
*(x::Dual, y::Dual) = Dual(x.a*y.a, x.a*y.b + x.b*y.a)

exp(x::Dual) = Dual(exp(x.a), exp(x.a) * x.b)
```

Out[ ]:
```
exp (generic function with 16 methods)
```

We can also try it on the two polynomials as above:

In [ ]:
```
f = x -> 1 + x + x^2
g = x -> 1 + x/3 + x^2
f(ϵ).b, g(ϵ).b
```

Out[ ]:
```
(1, 0.3333333333333333)
```

The first example exactly computes the derivative, and the second example is exact up to the last bit rounding! It also works for higher order polynomials:

In [ ]:
```
f = x -> 1 + 1.3x + 2.1x^2 + 3.1x^3
```

```
f(0.5 + ϵ).b - 5.725
```

Out[ ]:
```
8.881784197001252e-16
```

It is indeed "accurate to (roughly) 16-digits", the best we can hope for using floating point.

We can use this in "algorithms" as well as simple polynomials. Consider the polynomial $1 + \ldots + x^n$:

In [ ]:
```
function s(n, x)
    ret = 1 + x # first two terms
    for k = 2:n
        ret += x^k
    end
    ret
end
s(10, 0.1 + ϵ).b
```

Out[ ]:
```
1.2345678999999998
```

This matches exactly the "true" (up to rounding) derivative:

In [ ]:
```
sum((1:10) .* 0.1 .^(0:9))
```

Out[ ]:
```
1.2345678999999998
```

Finally, we can try the more complicated example:

In [ ]:
```
f = x -> exp(x^2 + exp(x))
f(1 + ϵ)
```

Out[ ]:
```
Dual{Float64}(41.193555674716116, 194.362805189629)
```

What makes dual numbers so effective is that, unlike finite differences, they are not prone to disasterous growth due to round-off errors.