Example 106. Determine the order of the approximation $f'(x) \approx \frac{1}{2h}[f(x+h) - f(x-h)]$.

Comment. This approximation of the derivative is called a **(first) central difference** for $f'(x)$.

Solution. By Taylor's theorem (Theorem 48),

$$\begin{aligned} f(x+h) &= f(x) + hf'(x) + \frac{h^2}{2}f''(x) + \frac{h^3}{6}f'''(x) + \frac{h^4}{24}f^{(4)}(x) + O(h^5), \\ f(x-h) &= f(x) - hf'(x) + \frac{h^2}{2}f''(x) - \frac{h^3}{6}f'''(x) + \frac{h^4}{24}f^{(4)}(x) + O(h^5). \end{aligned} \tag{2}$$

(Note that the second formula just has h replaced with $-h$.) Subtracting the second from the first, we obtain

$$\frac{1}{2h}[f(x+h) - f(x-h)] = f'(x) + \boxed{\frac{h^2}{6}f'''(x) + O(h^3)} = f'(x) + \boxed{O(h^2)}.$$

Hence, the $\boxed{\text{error}}$ is of order 2.

Example 107. Use both forward and central differences to approximate $f'(x)$ for $f(x) = x^2$.

Solution. We get $\frac{1}{h}[f(x+h) - f(x)] = 2x + h$ and $\frac{1}{2h}[f(x+h) - f(x-h)] = 2x$.

Comment. In the forward difference case, the error is of order 1 (also note that $\frac{h}{2}f''(x) = h$). In the central difference case, we find that we get $f'(x)$ without error. In hindsight, with the error formulas in mind, this is not a surprise and reflects the fact that $f'''(x) = 0$.

Example 108. Use both forward and central differences to approximate $f'(2)$ for $f(x) = 1/x$.

Solution. In each case, we use $h = \frac{1}{10}$ and $h = \frac{1}{20}$.

- $h = \frac{1}{10}$: $\frac{1}{h}[f(x+h) - f(x)] = -\frac{5}{21} \approx -0.2381$, error 0.0119
  $h = \frac{1}{20}$: $\frac{1}{h}[f(x+h) - f(x)] = -\frac{10}{41} \approx -0.2439$, error 0.0061 (reduced by about $\frac{1}{2}$)
- $h = \frac{1}{10}$: $\frac{1}{2h}[f(x+h) - f(x-h)] = -\frac{100}{399} \approx -0.25063$, error -0.00063
  $h = \frac{1}{20}$: $\frac{1}{2h}[f(x+h) - f(x-h)] = -\frac{400}{1599} \approx -0.25016$, error -0.00016 (reduced by about $\frac{1}{4}$)

Important comment. The forward difference has an error of order 1. In other words, for small h, it should behave like Ch. In particular, if we replace h by $h/2$, then the error should be about $1/2$ (as we saw above).
On the other hand, the central difference has an error of order 2 and so should behave like Ch^2. In particular, if we replace h by $h/2$, then the error should be about $1/2^2 = 1/4$ (and, again, this is what we saw above).

Example 109. Find a central difference for $f''(x)$ and determine the order of the error.

Solution. Adding the two expansions in (2) to kill the $f'(x)$ terms, and subtracting $2f(x)$, we find that

$$\frac{1}{h^2}[f(x+h) - 2f(x) + f(x-h)] = f''(x) + \boxed{\frac{h^2}{12}f^{(4)}(x) + O(h^3)} = f''(x) + \boxed{O(h^2)}.$$

The $\boxed{\text{error}}$ is of order 2.

Alternatively. If we iterate the approximation $f'(x) \approx \frac{1}{2h}[f(x+h) - f(x-h)]$ (in the second step, we apply it with x replaced by $x \pm h$), we obtain

$$f''(x) \approx \frac{1}{2h}[f'(x+h) - f'(x-h)] \approx \frac{1}{4h^2}[f(x+2h) - 2f(x) + f(x-2h)],$$

which is the same as what we found above, just with h replaced by $2h$.

Armin Straub
straub@southalabama.edu

Example 110. Obtain approximations for $f'(x)$ and $f''(x)$ using the values $f(x)$, $f(x+h)$, $f(x+2h)$ as follows: determine the polynomial interpolation corresponding to these values and then use its derivatives to approximate those of f. In each case, determine the order of the approximation and the leading term of the error.

Solution. We first compute the polynomial $p(t)$ that interpolates the three points $(x, f(x))$, $(x+h, f(x+h))$, $(x+2h, f(x+2h))$ using Newton's divided differences:

	$f[\cdot]$	$f[\cdot,\cdot]$	$f[\cdot,\cdot,\cdot]$
x	$f(x)$		
		$\dfrac{f(x+h)-f(x)}{h} =: c_1$	
$x+h$	$f(x+h)$		$\dfrac{f(x+2h)-2f(x+h)+f(x)}{2h^2} =: c_2$
		$\dfrac{f(x+2h)-f(x+h)}{h}$	
$x+2h$	$f(x+2h)$		

Hence, reading the coefficients from the top edge of the triangle, the interpolating polynomial is

$$p(t) = f(x) + c_1(t-x) + c_2(t-x)(t-x-h).$$

- **(approximating $f'(x)$)** Since $p'(t) = c_1 + c_2(2t-2x-h)$, we have

$$\begin{aligned} p'(x) &= c_1 - hc_2 = \frac{f(x+h)-f(x)}{h} - \frac{f(x+2h)-2f(x+h)+f(x)}{2h} \\ &= \frac{-f(x+2h)+4f(x+h)-3f(x)}{2h}. \end{aligned}$$

  This is our approximation for $f'(x)$. To determine the order and the error, we combine

$$\begin{aligned} f(x+h) &= f(x) + f'(x)h + \frac{1}{2}f''(x)h^2 + \frac{f'''(x)}{6}h^3 + O(h^4), \\ f(x+2h) &= f(x) + 2f'(x)h + 2f''(x)h^2 + \frac{4f'''(x)}{3}h^3 + O(h^4) \end{aligned}$$

  (note that the latter is just the former with h replaced by $2h$) to find

$$-f(x+2h)+4f(x+h)-3f(x) = 2f'(x)h - \frac{2f'''(x)}{3}h^3 + O(h^4).$$

  Hence, dividing by $2h$, we conclude that

$$\frac{-f(x+2h)+4f(x+h)-3f(x)}{2h} = f'(x) - \frac{f'''(x)}{3}h^2 + O(h^3).$$

  Consequently, the approximation is of order 2.

- **(approximating $f''(x)$)** Since $p''(t) = 2c_2$, we have $p''(x) = 2c_2 = \dfrac{f(x+2h)-2f(x+h)+f(x)}{h^2}$.
  This is our approximation for $f''(x)$. To determine the order and the error, we proceed as before to find

$$f(x+2h) - 2f(x+h) + f(x) = f''(x)h^2 + f'''(x)h^3 + O(h^4).$$

  Hence, dividing by h^2, we conclude that

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

  Consequently, the approximation is of order 1.

Example 111. (homework) Obtain approximations for $f'(x)$ and $f''(x)$ using the values $f(x-2h)$, $f(x)$, $f(x+3h)$ as follows: determine the polynomial interpolation corresponding to these values and then use its derivatives to approximate those of f. In each case, determine the order of the approximation and the leading term of the error.

Solution. We first compute the polynomial $p(t)$ that interpolates the three points $(x-2h, f(x-2h))$, $(x, f(x))$, $(x+3h, f(x+3h))$ using Newton's divided differences:

	$f[\cdot]$	$f[\cdot,\cdot]$	$f[\cdot,\cdot,\cdot]$
$x-2h$	$f(x-2h)$		
		$\frac{f(x)-f(x-2h)}{2h} =: c_1$	
x	$f(x)$		$\frac{2f(x+3h)-5f(x)+3f(x-2h)}{30h^2} =: c_2$
		$\frac{f(x+3h)-f(x)}{3h}$	
$x+3h$	$f(x+3h)$		

Hence, reading the coefficients from the top edge of the triangle, the interpolating polynomial is

$$p(t) = f(x) + c_1(t-x+2h) + c_2(t-x+2h)(t-x).$$

- **(approximating $f'(x)$)** Since $p'(t) = c_1 + c_2(2t-2x+2h)$, we have

$$\begin{aligned} p'(x) &= c_1 + 2h\,c_2 = \frac{f(x)-f(x-2h)}{2h} + \frac{2f(x+3h)-5f(x)+3f(x-2h)}{15h} \\ &= \frac{4f(x+3h)+5f(x)-9f(x-2h)}{30h}. \end{aligned}$$

  This is our approximation for $f'(x)$. To determine the order and the error, we combine

$$\begin{aligned} f(x+h) &= f(x) + f'(x)h + \frac{1}{2}f''(x)h^2 + \frac{f'''(x)}{6}h^3 + O(h^4), \\ f(x-2h) &= f(x) - 2f'(x)h + 2f''(x)h^2 - \frac{4f'''(x)}{3}h^3 + O(h^4), \\ f(x+3h) &= f(x) + 3f'(x)h + \frac{9}{2}f''(x)h^2 + \frac{9f'''(x)}{2}h^3 + O(h^4) \end{aligned}$$

  to find

$$4f(x+3h) + 5f(x) - 9f(x-2h) = 30f'(x)h + 30f'''(x)h^3 + O(h^4).$$

  Hence, dividing by $30h$, we conclude that

$$\frac{4f(x+3h)+5f(x)-9f(x-2h)}{30h} = f'(x) + f'''(x)h^2 + O(h^3).$$

  Consequently, the approximation is of order 2.

- **(approximating $f''(x)$)** Since $p''(t) = 2c_2$, we have $p''(x) = 2c_2 = \frac{2f(x+3h)-5f(x)+3f(x-2h)}{15h^2}$.
  This is our approximation for $f''(x)$. To determine the order and the error, we proceed as before to find

$$2f(x+3h) - 5f(x) + 3f(x-2h) = 15f''(x)h^2 + 5f'''(x)h^3 + O(h^4).$$

  Hence, dividing by $15h^2$, we conclude that

$$\frac{2f(x+3h)-5f(x)+3f(x-2h)}{15h^2} = f''(x) + \frac{1}{3}f'''(x)h + O(h^2).$$

  Consequently, the approximation is of order 1.

Python assignment #4

The goal of this assignment is to gain some first acquaintance with recursion as a coding technique. First, in the case of the factorial function, we observe that recursion provides a simple and elegant way to implement certain kinds of functions. In a second example, we then encounter a potential issue of recursion that one needs to be aware of.

Example 112. Python We can quickly implement a function for computing $n!$ in an iterative fashion as follows (such a function is also available in the Python `math` library as well as in `numpy` and `scipy`).

```
>>> def factorial_iterative(n):
        f = 1
        for k in range(1,n+1):
            f = f*k
        return f

>>> factorial_iterative(3)

    6
```

On the other hand, a recursive implementation would take the following form:

```
>>> def factorial_recursive(n):
        if n == 0: return 1
        return n * factorial_recursive(n-1)

>>> factorial_recursive(4)

    24
```

Sometimes it is useful to measure how fast code is running. One tool for doing this in Python is the `timeit` module. The following measures how many seconds it takes to compute $100!$ using our two functions 10^4 many times.

```
>>> from timeit import timeit

>>> timeit('factorial_iterative(100)', number=10**4, globals=globals())

    0.051031902898103

>>> timeit('factorial_recursive(100)', number=10**4, globals=globals())

    0.11693716002628207
```

As we can see, the recursive implementation is a bit slower (by about a factor of two in this case of computing $100!$) than the iterative implementation. However, unless performance was critical, such a difference should be considered relatively minor and not a reason for us to go out of our way to avoid one or the other (the time spent coding can be much more valuable than the milliseconds saved by optimized code).

Comment. Finally, just out of curiousity, here is a comparison with the factorial function implemented in the standard `math` library that comes with Python. Not surprisingly, that is the fastest (at about 8 times faster than our iterative implementation).

```
>>> from math import factorial

>>> timeit('factorial(100)', number=10**4, globals=globals())

    0.0072873481549322605
```

Advanced comment. If you try our recursive function on a large input, you will run into an error about exceeding the recursion limit. If necessary, that limit can be increased using the function `setrecursionlimit` from the `sys` module.

Example 113. Python The Fibonacci numbers F_n are defined by the formula $F_{n+1} = F_n + F_{n-1}$ together with the initial conditions $F_0 = F_1 = 1$. Below is a recursive implementation that directly mirrors the mathematical description.

```
>>> def fibonacci_recursive(n):
        if n == 0: return 0
        if n == 1: return 1
        return fibonacci_recursive(n-1) + fibonacci_recursive(n-2)

>>> [fibonacci_recursive(n) for n in range(10)]

   [0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

However, there is trouble!

```
>>> fibonacci_recursive(34)

   5702887
```

This computation of F_{34} took more than a second (at least on my little laptop). And the computation of, say, F_{40} will take ages. Try it and see your machine sweat!

(a) Can you explain why the recursive implementation above is becoming so slow?

(b) Provide an iterative implementation of the Fibonacci numbers that avoids the above issue.

Advanced comment. An alternative approach around our issue is to keep the recursive logic but to store previously computed values. Typically, one uses a dictionary of previously computed values and, at the beginning of the function, checks whether the current function input has occurred before. In Python one can also add this behaviour to a function by using a suitable "decorator".

https://docs.python.org/3/library/functools.html#functools.cache

After you have submitted your code on Replit, send me an email with the following:

(a) The exact value of $100!$ (the value we used for the timings in the first example).

(b) The exact value of F_{100}.

(c) A sentence or two explaining why the recursive implementation of the Fibonacci numbers becomes unusably slow so quickly.