Coding excursion: iteration vs recursion

The goal of this excursion 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 134. 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) than the iterative implementation. Unless performance was critical, such a difference could 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. That limit can be increased using the function `setrecursionlimit` from the `sys` module.

Example 135. 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.

Bonus. For a bonus point, send me your solutions to these before the last week of classes.

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

Numerical methods for solving differential equations

The general form of a first-order differential equation (DE) is $y' = f(x, y)$.

Comment. Recall that higher-order differential equations can be written as systems of first-order differential equations: $\boldsymbol{y}' = f(x, \boldsymbol{y})$ in terms of $\boldsymbol{y} = (y_1, y_2, y_3, \ldots)$ where we set $y_1 = y$, $y_2 = y'$, $y_3 = y''$, ...

It therefore is no loss of generality to develop methods for first-order differential equation. While we will focus on the case of a single function $y(x)$, the methods we discuss extend naturally to the case of several functions $\boldsymbol{y}(x) = (y_1(x), y_2(x), \ldots)$.

In order to have a unique solution $y(x)$ that we can numerically approximate, we will add an initial condition. As such, we discuss methods for solving first-order initial value problems (IVPs)

$$y' = f(x, y), \quad y(x_0) = y_0.$$

Comment. Recall from your Differential Equations class that such an IVP is guaranteed to have a unique solution under mild assumptions on $f(x, y)$ (for instance, that $f(x, y)$ is smooth around (x_0, y_0)).

Comment. There would be no loss of generality in only considering initial conditions of the form $y(0) = y_0$. Indeed, suppose the initial condition is $y(x_0) = y_0$. Then, by replacing x by $x + x_0$ in the DE and rewriting the DE in terms of $\tilde{y}(x) = y(x + x_0)$, we obtain an IVP with initial condition $\tilde{y}(0) = y_0$.

Review of the simplest differential equations

Let's start with one of the simplest (and most fundamental) differential equation (DE). It is **first-order** (only a first derivative) and **linear** (with constant coefficients).

Example 136. Solve $y' = 3y$.

Solution. $y(x) = Ce^{3x}$

Check. Indeed, if $y(x) = Ce^{3x}$, then $y'(x) = 3Ce^{3x} = 3y(x)$.

Comment. Recall we can always easily check whether a function solves a differential equation. This means that (although you might be unfamiliar with certain techniques for solving) you can use computer algebra systems to solve differential equations without trust issues.

To describe a unique solution, additional constraints need to be imposed.

Example 137. Solve the **initial value problem** (IVP) $y' = 3y$, $y(0) = 5$.

Solution. This has the unique solution $y(x) = 5e^{3x}$.

The following is a **non-linear** differential equation. In general, such equations are much more complicated than linear ones. We can solve this particular one because it is **separable**.

Example 138. Solve $y' = xy^2$.

Solution. This DE is separable: $\frac{1}{y^2}\mathrm{d}y = x\,\mathrm{d}x$. Integrating, we find $-\frac{1}{y} = \frac{1}{2}x^2 + C$.

Hence, $y = -\frac{1}{\frac{1}{2}x^2 + C} = \frac{2}{D - x^2}$. [Here, $D = -2C$ but that relationship doesn't matter.]

Comment. Note that we did not find the singular solution $y = 0$ (lost when dividing by y^2). We can obtain it from the general solution by letting $D \to \infty$.

Euler's method

Euler's method is a numerical way of approximating the (unique) solution $y(x)$ to the IVP

$$y' = f(x, y), \quad y(x_0) = y_0.$$

It follows from Taylor's theorem (Theorem 52) that

$$y(x+h) = y(x) + y'(x)h + \frac{1}{2}y''(\xi)h^2.$$

Choose a step size $h > 0$. Write $x_n = x_0 + nh$. Our goal is to provide approximations y_n of $y(x_n)$ for $n = 1, 2, \ldots$.

Since we know $y(x_0) = y_0$, we approximate:

$$\begin{aligned}
y(x_0+h) &\approx y(x_0) + y'(x_0)h \overset{\text{DE}}{=} y(x_0) + f(x_0, y(x_0))h \\
y(x_0+2h) &\approx y(x_0+h) + y'(x_0+h)h \overset{\text{DE}}{=} y(x_0+h) + f(x_0+h, y(x_0+h))h \\
y(x_0+3h) &\approx y(x_0+2h) + y'(x_0+2h)h \overset{\text{DE}}{=} y(x_0+2h) + f(x_0+2h, y(x_0+2h))h \\
&\vdots
\end{aligned}$$

Comments.

- Here we use $y(x+h) \approx y(x) + y'(x)h$ first with $x = x_0$, then with $x = x_0 + h$ and so on.
- Note how, when approximating $y(x_0 + mh)$, we use the previous approximation $y(x_0 + (m-1)h)$. All other quantities on the right-hand side are known to us.
- Clearly, the error in these approximations will accumulate and the approximations are likely worse as we continue (in other words, our approximations of $y(x)$ will be worse as x gets further away from x_0).

Write $x_n = x_0 + nh$. Our goal is to provide approximations y_n of $y(x_n)$ for $n = 1, 2, \ldots$.

Note that we start with x_0 and y_0 from the initial condition.

In terms of x_n and y_n our above approximations become:

$$y(x_n + h) \approx y(x_n) + \underbrace{y'(x_n)}_{f(x_n, y(x_n))} h \approx y_n + f(x_n, y_n)h =: y_n$$

Two kinds of errors. There are two different errors involved here: in the first approximation, the error is from truncating the Taylor expansion and we know that this **local truncation error** is $O(h^2)$. On the other hand, in the second approximation, we introduce an error because we use the previous approximation y_n instead of $y(x_n)$.

Suppose that we approximate $y(x)$ on some interval $[x_0, x_{\max}]$ using n steps (so that $x_n = x_{\max}$).

Then the step size is $h = \frac{x_{\max} - x_0}{n}$. We therefore have $n = \frac{x_{\max} - x_0}{h}$ many local truncation errors of size $O(h^2)$. It is therefore natural to expect that the **global error** is $O(nh^2) = O(h)$.

(Euler's method) The following is an order 1 method for solving IVPs:

$$y_{n+1} = y_n + f(x_n, y_n)h$$

Comment. As explained above, being an order 1 method means that Euler's method has a global error that is $O(h)$ (while the local truncation error is $O(h^2)$).

Comment. While Euler's method is rarely used in practice, it serves as the foundation for more powerful extensions such as the Runge–Kutta methods.

Euler's method applied to e^x

Example 139. Consider the IVP $y' = y$, $y(0) = 1$. Approximate the solution $y(x)$ for $x \in [0, 1]$ using Euler's method with 4 steps. In particular, what is the approximation for $y(1)$?

Comment. Of course, the real solution is $y(x) = e^x$. In particular, $y(1) = e \approx 2.71828$.

Solution. The step size is $h = \frac{1-0}{4} = \frac{1}{4}$. We apply Euler's method with $f(x, y) = y$:

$$\begin{aligned}
x_0 &= 0 & y_0 &= 1 \\
x_1 &= \frac{1}{4} & y_1 &= y_0 + f(x_0, y_0)h = 1 + \frac{1}{4} = \frac{5}{4} = 1.25 \\
x_2 &= \frac{1}{2} & y_2 &= y_1 + f(x_1, y_1)h = \frac{5}{4} + \frac{5}{4} \cdot \frac{1}{4} = \frac{5^2}{4^2} = 1.5625 \\
x_3 &= \frac{3}{4} & y_3 &= y_2 + f(x_2, y_2)h = \frac{5^2}{4^2} + \frac{5^2}{4^2} \cdot \frac{1}{4} = \frac{5^3}{4^3} \approx 1.9531 \\
x_4 &= 1 & y_4 &= y_3 + f(x_3, y_3)h = \frac{5^3}{4^3} + \frac{5^3}{4^3} \cdot \frac{1}{4} = \frac{5^4}{4^4} \approx 2.4414
\end{aligned}$$

In particular, the approximation for $y(1)$ is $y_4 \approx 2.4414$.

Comment. Can you see that, if instead we start with $h = \frac{1}{n}$, then we similarly get $x_i = \frac{(n+1)^i}{n^i}$ for $i = 0, 1, \ldots, n$? In particular, $y(1) \approx y_n = \frac{(n+1)^n}{n^n} = \left(1 + \frac{1}{n}\right)^n \to e$ as $n \to \infty$. Do you recall how to derive this final limit?