Example 143. Python Let us implement Euler's method to redo and extend Example 140.

```
>>> def euler(f, x0, y0, xmax, n):
        h = (xmax - x0) / n
        ypoints = [y0]
        for i in range(n):
            y0 = y0 + f(x0,y0)*h
            x0 = x0 + h
            ypoints.append(y0)
        return ypoints

>>> def f(x, y):
        return (2*x-3*y)**2 + 2/3
```

If we choose the number of steps `n` to be 3 and `xmax` to be 2 (because we want $x_n = 2$), then the following matches exactly our computation in Example 140 (with 3 steps):

```
>>> euler(f, 1, 1/3, 2, 3)

    [0.3333333333333333, 0.8888888888888888, 1.1111111111111112, 1.3333333333333335]
```

As expected, increasing the number of steps provides better approximations to the exact solution $y(x) = \frac{2}{3}x - \frac{1}{3(3x-2)}$ with $y(2) = \frac{5}{4} = 1.25$.

```
>>> euler(f, 1, 1/3, 2, 10)

    [0.3333333333333333, 0.5, 0.6156666666666667, 0.7129142333333334, 0.8008567296803291,
    0.8833184418108875, 0.9622382358968495, 1.0387196700012582, 1.1134429074281287,
    1.1868524913418415, 1.259252430333012]

>>> euler(f, 1, 1/3, 2, 100)[-1]

    1.2508710385019504
```

If `ypoints` is a list, then its elements can be accessed as `ypoints[0]`, `ypoints[1]`, ... Moreover, we can access the last element as `ypoints[-1]`. For instance, above, we used `euler(f, 1, 1/3, 2, 100)[-1]` to get the last element of the 101 approximations $y_0, y_1, ..., y_{100}$. That last element is the approximation of $y(2) = 1.25$.

The following convincingly illustrates that the error in Euler's method is $O(h)$.

```
>>> [euler(f, 1, 1/3, 2, 10**n)[-1] - 1.25 for n in range(7)]

    [0.7499999999999998, 0.00925243033301193, 0.0008710385019503608, 8.668868812233832e-05,
    8.664791898871371e-06, 8.664409443248644e-07, 8.66125893228542e-08]
```

However, note that our computer had to work pretty hard to get the final approximation, because that entailed computing 10^6 values. We clearly need a higher order method in order to compute to higher accuracy.

Taylor methods

(Taylor method of order k) The following is an order k method for solving IVPs:

$$\begin{aligned} x_{n+1} &= x_n + h \\ y_{n+1} &= y_n + f(x_n, y_n)h + \tfrac{1}{2}f'(x_n, y_n)h^2 + \cdots + \tfrac{1}{k!}f^{(k-1)}(x_n, y_n)h^k \end{aligned}$$

where $f^{(n)}(x, y)$ is short for $\frac{\mathrm{d}^n}{\mathrm{d}x^n}f(x, y(x))$ (expressed in terms of f and its partial derivatives).

For instance. $f'(x,y) = \frac{\mathrm{d}}{\mathrm{d}x}f(x,y(x)) = f_x(x,y) + f_y(x,y)y'(x) = f_x(x,y) + f_y(x,y)f(x,y)$

Especially for higher derivatives, it is easier to compute these for specific f. See next example.

Comment. As for Euler's method, being an order k method means that the method has a global error that is $O(h^k)$ (while the local truncation error is $O(h^{k+1})$; note that we can see this because we truncate the Taylor expansion of $y(x)$ after h^k so that the next term is $O(h^{k+1})$).

Example 144. Spell out the Taylor method of order 2 for numerically solving the IVP

$$y' = \cos(x)y, \quad y(0) = 1.$$

Solution. The Taylor method of order 2 is based on the Taylor expansion

$$y(x+h) = y(x) + y'(x)h + \tfrac{1}{2}y''(x)h^2 + O(h^3),$$

where we have a local truncation error of $O(h^3)$ so that the global error will be $O(h^2)$.
From the DE we know that $y'(x) = \cos(x)y$, which is $f(x,y)$. We differentiate this to obtain

$$\begin{aligned} y''(x) &= \frac{\mathrm{d}}{\mathrm{d}x}\cos(x)y = -\sin(x)y + \cos(x)y' = -\sin(x)y + \cos^2(x)\,y \\ &= (-\sin(x) + \cos^2(x))y, \end{aligned}$$

which is $f'(x,y)$. Hence, the Taylor method of order 2 takes the form:

$$\begin{aligned} y_{n+1} &= y_n + f(x_n, y_n)h + \tfrac{1}{2}f'(x_n, y_n)h^2 \\ &= y_n + \cos(x_n)y_n\,h + \tfrac{1}{2}((-\sin(x_n) + \cos^2(x_n))y_n)h^2 \end{aligned}$$

For any choice of h, we can therefore compute $(x_1, y_1), (x_2, y_2), \ldots$ starting with (x_0, y_0) by the above recursive formula combined with $x_{n+1} = x_n + h$.

Example 145. Spell out the Taylor method of order 3 for numerically solving the IVP

$$y' = \cos(x)y, \quad y(0) = 1.$$

Solution. The Taylor method of order 3 is based on the Taylor expansion

$$y(x+h) = y(x) + y'(x)h + \tfrac{1}{2}y''(x)h^2 + \tfrac{1}{6}y'''(x)h^3 + O(h^4),$$

where we have a local truncation error of $O(h^4)$ so that the global error will be $O(h^3)$.
From the DE we know that $y'(x) = \cos(x)y$, which is $f(x,y)$. As in the previous example, we differentiate this to obtain

$$\begin{aligned} y''(x) &= \frac{\mathrm{d}}{\mathrm{d}x}\cos(x)y = -\sin(x)y + \cos(x)y' = -\sin(x)y + \cos^2(x)\, y \\ &= (-\sin(x) + \cos^2(x))y, \end{aligned}$$

which is $f'(x,y)$. Once more differentiating this, we obtain

$$\begin{aligned} y'''(x) &= \frac{\mathrm{d}}{\mathrm{d}x}(-\sin(x) + \cos^2(x))y \\ &= (-\cos(x) - 2\cos(x)\sin(x))y + (-\sin(x) + \cos^2(x))y' \\ &= (\cos^2(x) - 3\sin(x) - 1)\cos(x)y, \end{aligned}$$

which is $f''(x,y)$. Hence, the Taylor method of order 3 takes the form:

$$\begin{aligned} y_{n+1} &= y_n + f(x_n,y_n)h + \tfrac{1}{2}f'(x_n,y_n)h^2 + \tfrac{1}{6}f''(x_n,y_n)h^3 \\ &= y_n + \cos(x_n)y_n\, h + \tfrac{1}{2}((-\sin(x_n) + \cos^2(x_n))y_n)h^2 + \tfrac{1}{6}((\cos^2(x_n) - 3\sin(x_n) - 1)\cos(x_n)y_n)h^3 \end{aligned}$$

Comment. The exact solution of this IVP is $y(x) = e^{\sin(x)}$. We use this in the next example for comparison.

Example 146. Python Let us use Python to approximate the solution $y(x)$ of the IVP from the previous example for $x \in [0,2]$.

```
>>> from math import e, cos, sin

>>> def taylor_3_cosy(x0, y0, xmax, n):
        h = (xmax - x0) / n
        ypoints = [y0]
        for i in range(n):
            y0 = y0 + cos(x0)*y0*h + 1/2*(cos(x0)**2-sin(x0))*y0*h**2 + \
                 1/6*(cos(x0)**2-3*sin(x0)-1)*cos(x0)*y0*h**3
            x0 = x0 + h
            ypoints.append(y0)
        return ypoints
```

Since the exact solution is $y(x) = e^{\sin(x)}$, we have $y(2) = e^{\sin(2)} \approx 2.48258$.

```
>>> taylor_3_cosy(0, 1, 2, 4)

   [1, 1.625, 2.3475297541746047, 2.7350418255304874, 2.476391322837691]
```

For comparison, the exact values of the solution at these four steps are:

```
>>> [e**sin(i/2) for i in range(5)]

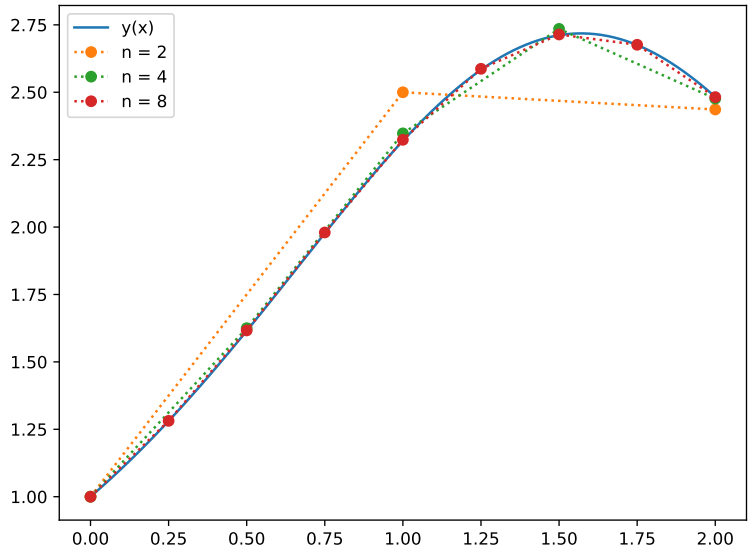
   [1.0, 1.6151462964420837, 2.319776824715853, 2.7114810176821584, 2.4825777280150003]
```

The following convincingly illustrates that the error is indeed $O(h^3)$.

```
>>> [taylor_3_cosy(0, 1, 2, 10**n)[-1] - e**sin(2) for n in range(5)]
```

```
[2.5174222719849997, -0.0002461575553160955, -1.6375769584797695e-07,
-1.5647971807197791e-10, 2.0339285811132868e-13]
```

The following is a plot of the exact solution together with our approximations when using 2, 4 and 8 steps. Already for 4 steps, we obtain an approximation that, at least visually, is remarkably good.



On the other hand, for comparison, the following plot shows the corresponding approximations when using Euler's method instead.

