Case study definition

Let's define a function with a complicated expression:

$$f\colon (x, y) \to \cos\left[(xy + (x - 4)^2 + \arctan((y + 3)^2))x\right]$$

With $x > 0$ and $y > 0$.

`sympy` wil be used to compute partial derivatives of this function.

Key idea of `sympy`

`sympy` can handle mathematical objects in a formal way, i.e. **without using numerical discretization but relying on the properties of the defined objects**.

With `sympy`, objects are defined with a mathematical meaning that goes beyond the software meaning of traditional Python code. Then some operations are performed on these objects:

- differenciation
- integration
- limits
- etc...

Code

Mathematical objects declaration

Let's define two mathematical variables `x` and `y`. `sympy` is told these variables must take positive real values, using `positive=True`.

In [1]:
```
import sympy as sym

x = sym.Symbol('x', positive=True)
y = sym.Symbol('y', positive=True)
```

Then the function is defined:

In [2]:
```
f = sym.Lambda((x, y),
               sym.cos((x*y + (x-4)**2 + sym.atan((y+3)**2))*x))
f
```

Out[2]:
$$\left((x,\ y) \mapsto \cos\left(x\left(xy + (x - 4)^2 + \operatorname{atan}\left((y + 3)^2\right)\right)\right)\right)$$

Be careful! Functions cos, atan and log are the one of `sympy` !

The function can be evaluated at a specific point:

In [3]:
```
function_call = f(2, 4)
```

Yet, `evalf` must be used to get a numerical approximation. `evalf` returns a `sympy`-related type, it can be converted using `float`:

In [4]:
```
print(type(function_call.evalf()))
print(float(function_call.evalf()))
```

```
<class 'sympy.core.numbers.Float'>
-0.3868788252007259
```

Operations on defined objects

Let's compute the partial derivative of `f` with respect to x, using `diff`:

In [5]:
```
f(x, y).diff(x)
```

Out[5]:
$$-\left(xy + x(2x + y - 8) + (x - 4)^2 + \operatorname{atan}\left((y + 3)^2\right)\right)\sin\left(x\left(xy + (x - 4)^2 + \operatorname{atan}\left((y + 3)^2\right)\right)\right)$$

Or compute the second derivative with respect to x and then the first derivative with respect to y:

In [6]:
```
der = f(x, y).diff(x, 2, y)
der
```

Out[6]:

$$-2x\left(x + \frac{2(y+3)}{(y+3)^4+1}\right)(3x+y-8)\cos\left(x\left(xy+(x-4)^2+\operatorname{atan}\left((y+3)^2\right)\right)\right) + x\left(x + \frac{2(y+3)}{(y+3)^4+1}\right)\left(xy + x(2x+y-8) + (\right.$$

In this expression, x and y are still unknown. Let's replace x using `subs`:

In [7]:
```
der = der.subs({x: 5})
der
```

Out[7]:

$$-10(y+7)\left(\frac{2(y+3)}{(y+3)^4+1}+5\right)\cos\left(25y+5\operatorname{atan}\left((y+3)^2\right)+5\right)-4\left(\frac{y+3}{(y+3)^4+1}+5\right)\left(10y+\operatorname{atan}\left((y+3)^2\right)+11\right)\cos\left(2\right.$$

`numpy` conversion

The evaluation of `der` is not fast. Thus it can be converted to a `numpy` function using `lambdify`. Then evaluation on arrays is possible:

In [8]:
```
import numpy as np
numpy_func = sym.lambdify(y, der)
Y = np.linspace(0, 10, 5)
numpy_func(Y)
```

Out[8]:
```
array([  -1618.36732557,   -8543.34409677,  -44619.86174987,
       -132966.53918376, -279318.28704865])
```

When to use `sympy`

`sympy` can provide exact mathematical solutions **for simple problems only**. Thus, it can be used to check the results of some very specific calculation steps performed in a more numerical way.