Speeding up numerical integration using numpy's vectorization (1/9)

- numerical integration methods (even if implemented in Python) are already very fast for a single problem instance
- but in some applications (e.g. navigation systems) numerical integration problems have to be solved for many instances with varying functions and interval boundaries
- so speeding up numerical integration is important and we will see that it is not difficult to achieve
- the two methods for numerical integration have a very simple structure
- view them as methods which, for a vector of evenly spaced points x_i, computes a vector of function values $f(x_i)$ which are summed up

Speeding up numerical integration using numpy's vectorization (2/9)

x_0	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	x_{10}	x_{11}	x_{12}	x_{13}	x_{14}	x_{15}
$f(x_0)$	$f(x_1)$	$f(x_2)$	$f(x_3)$	$f(x_4)$	$f(x_5)$	$f(x_6)$	$f(x_7)$	$f(x_8)$	$f(x_9)$	$f(x_{10})$	$f(x_{11})$	$f(x_{12})$	$f(x_{13})$	$f(x_{14})$	$f(x_{15})$

$$\sum$$

- this is a typical structure of numerical algorithms
- it is amendable to vectorization, which compute such vectors and their sum extremely fast
- the speedup is achieved by using instructions of the processor which can compute a constant number (usually 4) of function values and their sum in a single CPU-cycle
- such instructions are available and easy to use via `numpy`, a very widely used module of Python
- to apply `numpy` to the velocity function v, we have to implement a version which uses `numpy`-methods instead of `math`-methods:

Speeding up numerical integration using numpy's vectorization (3/9)

```
import numpy as np
def np_velocity(t):
  return 3 * t * t * (np.power(np.e,t * t * t))
```

- the trapezoid computes $d \cdot \left(\frac{1}{2}\left(f(p) + f(q)\right) + \sum_{i=1}^{n-1} f(x_i)\right)$ which can be implemented in 4 lines of Python code

```
def np_approx_integral_trpz(f, p, q, n):
  d = (q-p)/n
  x_array = np.linspace(p + d, q - d, n-1)
  return d * (0.5 * (f(p) + f(q)) + np.sum(f(x_array)))
```

- the interface is the same as before, but requires that the function to integrate only uses `numpy`-methods besides basic arithmetic operations
- the first step is to determine (as done previously) the distance of two consecutive interval boundaries inside the integration interval $[p, q]$,

Speeding up numerical integration using numpy's vectorization (4/9)

- the second step computes an array of $n-1$ evenly spaced points using the `linspace`-method of `numpy`
- besides n, this method requires the specification of the
  - first value $p+d$ of the vector (i.e. x_1),
  - last value $q-d$ of the vector (i.e. x_{n-1})
- applying the function `f` to each value of this array `x_array` is expressed by applying `f` to `x_array`: so the iteration is implicit
- `f(x_array)` returns a new array of function values which are summed up using the `sum`-method from `numpy`
- for $n=16$, the structure can be depicted as follows:

Speeding up numerical integration using numpy's vectorization (5/9)

$$\overbrace{\begin{array}{|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|}\hline x_1 & x_2 & x_3 & x_4 & x_5 & x_6 & x_7 & x_8 & x_9 & x_{10} & x_{11} & x_{12} & x_{13} & x_{14} & x_{15} \\ \hline \end{array}}^{\text{x\_vector} = \text{np.linspace(...)}}$$

$$\underbrace{\underbrace{\begin{array}{|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|}\hline f(x_1) & f(x_2) & f(x_3) & f(x_4) & f(x_5) & f(x_6) & f(x_7) & f(x_8) & f(x_9) & f(x_{10}) & f(x_{11}) & f(x_{12}) & f(x_{13}) & f(x_{14}) & f(x_{15}) \\ \hline \end{array}}_{f(x\_vector)}}_{\text{np.sum}}$$

- we now want to measure the runtime of the different methods to verify if the effort was worth it
- we use the class `Timer` from module `timeit` and the `partial`-method from `functools`
- additionally we have to import the integration methods and the functions we want to integrate

Speeding up numerical integration using numpy's vectorization (6/9)

```
from timeit import Timer
from approx_integral import approx_integral_trpz, \
                            np_approx_integral_trpz
from funcdefs import velocity, np_velocity
from functools import partial
```

- we cannot directly supply the timer with a function call
- instead we need to create a partial object, that behaves like the corresponding function call, when actually called
- such a partial object is created by the method `partial`, which takes a function and a list of its arguments as parameter
- to reuse it, we encapsulate the creation of the partial object and the call to the timer in the following function

Speeding up numerical integration using numpy's vectorization (7/9)

```
def runtime_get(func,*args):
  partial_object = partial(func,*args)
  times = Timer(partial_object).repeat(3,1)
  return min(times)
```

- it returns the minimum of the runtime of three repetitive calls to the given function with the given argument
- for the runtime measurement, we specify the concrete boundaries, the number of steps, and provide `runtime_get` with the function, for which we want to measure the runtime

Speeding up numerical integration using numpy's vectorization (8/9)

```
p = 0.0
q = 1.0
n = 10000000
t = runtime_get(approx_integral_trpz,
                velocity,p,q,n)
print('runtime approx_integral_trpz: {:.2f} s'
      .format(t))
t = runtime_get(np_approx_integral_trpz,
                np_velocity,p,q,n)
print('runtime np_approx_integral_trpz: {:.2f} s'
      .format(t))
```

– for the chosen value of $n = 10\,000\,000$ we see that the numpy-based integration method is faster by a factor of ≈ 14 compared the direct implementation using its own for-loop

```
runtime approx_integral_trpz: 6.97 s
runtime np_approx_integral_trpz: 0.50 s
```

runtime for pure C-version of approx_integral_trpz: 0.3 s

Speeding up numerical integration using numpy's vectorization (9/9)

- from the vectorization, we would expect a speedup of a factor of at most 4 (because the vectorization handles four floating point values in one computation cycle)
- the additional speedup comes from the fact that the entire iterations of `np_approx_integral_mid` are performed inside the methods `np.linspace` and `np.sum`
- their calls are executed very fast by corresponding library functions not implemented in Python, but C
- the Python interpreter is not involved in the execution of these methods, except that it provides the methods with their arguments and receives their results