University of Zurich[UZH]
Faculty of Science

Software-based Speed-up Exercises

26 June 2025

Licence: CC-by-sa 4.0

Before you start:
- create a suitable directory for this exercise
- download the zipped material from `http://www.physik.uzh.ch/~python/python/lecture_c++/`

Exercise 1: Root Finder

Find iteratively a root of a function f by calculating $x_n = x_{n-1} - f(x_{n-1})/f'(x_{n-1}$. Select a start value x_0 and stop the iteration once $f(x_n)$ is close enough to zero *e.g.* $|f(x_n)| < \varepsilon$ with $\varepsilon = 0.001$. This is the so-called Newton method.

Design the algorithm first in pure Python and then in Cython. Compare the performance. You can use $f(x) = x^3 + x$ as a function for test purposes.

Tip: You can start with a simple version and then improve the functionality, *e.g.* defining the function or the precision of the derivative calculation at runtime.

Exercise 2: Differential Equation Solver

Implement the functionality of a differential equation solver taking as arguments the starting value of the independent, $t = t_0$, and the dependent variable, $x = x_0$, as well as the actual equation as a function $x' = f(x, t)$.
Create first the pure Python and then the Cython implementation. You can take your favourite solving algorithm (Euler ($x(t+\Delta t) = x(t)+x'(t)\Delta t$), Runge-Kutta, ...; `en.wikipedia.org/wiki/Numerical_methods_for_ordinary_differential_equations`).

Tip: Create it first for a single equation and afterwards expand it to an array of coupled equations also allowing for higher-orders. The output of the solver could be an array of $x(t)$ at the different time steps or just the value of x at the final time.
You can use as a test bed for the single equation $\mathrm{d}T/\mathrm{d}t + C(T - T_0) = 0$ corresponding to the temperature change of an object in an environment with ambient temperature T_0. For a set of functions you can use the differential equation of a damped oscillator $\mathrm{d}^2x/\mathrm{d}t^2+\beta\mathrm{d}x/\mathrm{d}t+\omega^2x = 0$ (*i.e.* with the second trivial equation $x' = \mathrm{d}x/\mathrm{d}t$).

Exercise 3: Auto-Regressive Model

We want to simulate an auto-regressive time series X_t of the form

$$X_t = c + \alpha X_{t-1} + \varepsilon_t$$

where ε_t is a uniform random number between -1 and 1, $X_0 = 0$, c is a constant and $|\alpha| < 1$.

Create an algorithm returning the n-th value of the series. Use pure python as well as Cython. How large is the gain in CPU time?
Tip: You can import into Cython C's standard library random number generator for positive integers and the maximum value wiht `from libc.stdlib cimport rand, RAND_MAX`.

Exercise 4: Exception Handling in Cython

This exercise has the goal to play a bit with the exception behaviour in Cython.

You can find in the folder `exception` for this part the file `except1.pyx` a function throwing an error if its input is zero. Otherwise it returns the input value. Compile this Cython code running the corresponding setup script `setup_except1.py` as we have learnt in the lecture. (Be aware that you might need to change the shared object name due to additional extensions!) Afterwards you can use the `runExcept1.py` script to run the function in an except clause with `$ python runExcept1.py <input number>`. What happens if the input is 0, 1 or 2? And why?
Add the `except *` statement in the definition of `pythonError` in `except1.pyx`. Recompile the file and run it again. What happens now?
Now change it to `except 1` and recompile it. What is now the outcome and why?

Now we want also study the handling of C++ exceptions. You can find a bunch of functions raising C++ exceptions in `except.h`. The file `except2.pyx` wraps them. So compile this Cython code with the `setup_except2.py` file and import the created module in a Python session. Try out what happens if you call the different wrapper functions. How can you change this annoying behaviour (seen in the lecture)? How can you force the wrapper functions to throw a specific Python error? Try it out!

In the last part of this exercise we want to discuss a specific difference between Python and C. You can find in the corresponding exercise folder the file `div.c`. Compile this program with `$ g++ div.c -o div`. This program takes two numbers as arguments and calculates the ratio (*e.g.* `$ ./div 3 4`). What happens if you make a division by zero? Now start Python and perform a division by zero. What happens?
Create a function in Cython that calculates the ratio of two numbers. What is the output in case of a division by zero here?
Cython has a way to force C behaviour on this issue. The `cython` module has a decorator `@cython.cdivision(bool)`, which can be called before the function. Try it out!

Exercise 5: Playing with STL in Cython

Write an algorithm that finds all anagrams (*i.e.* words with the same set of letters) based on a Python dictionary (Python) or a `std::map` in Cython. Use the `/usr/share/dict/ words` file as input. What is the set of letter allowing to build to largest number of words? (If you struggle with the algorithm, have a look at `STL/anagram.py`!)
Compare the Python and the Cython implementation in terms of spent CPU time. Why does the observed result occur? How can you improve the speed of the Cython version?

Exercise 6: Wrap a Function in Cython

The folder `wrapping_func` contains the file `libgcd.c` that includes the function `gcd` that implements the Euler algorithm to find the greatest common divisor.
Write Cython code that wraps this function to expose it in Python (you have seen some similar example in the lecture when we talked about exceptions) and then compile it to create a Python module.
There are the files `wrapping.pyx` and `setup.py` that contain the code to wrap it as well as to build the code, but try it first from scratch.
You can also start by trying to run the example from the lecture material.

Exercise 7: Wrap a Class in Cython

The folder `wrapping_class` contains a C++ class `Vector` and its derived class `UnitVector`. Have a look at the corresponding files. Some of the features are commented out at the moment. Write a wrapper in Cython for the `Vector` class (*cf.* lecture). Compile the code to a shared object and test it in Python.

Include now also the `coordinates` function using `std::vector`.

Include the addition operator.

Also try to include all the other operators.

Finally, write also a wrapper for the `UnitVector` class.

Write some test functions in Python and measure the CPU time. If you are still not bored about wrapping C++ classes, write the same test functions in C++ and compare the used CPU time.
You can also start by trying to run the example from the lecture material.

Exercise 8: Wrap a Class with SWIG

The folder `wrapping_class` contains a C++ class `Vector` and its derived class `UnitVector`. Have a look at the corresponding files. Some of the features are commented out at the moment. Write a SWIG interface file for the `Vector` class (*cf.* lecture). Compile a shared object and test it in Python.

Include sequentially the excluded functions and operators, modify the interface file and recompile the shared object.

Write some test functions in Python to measure the CPU time, compare it to – if done – the Cython wrapping and the same test functions in pure C++.
You can also start by trying to run the example from the lecture material.

Exercise 9: NumPy *vs.* Cython

One topic which was not discussed in the lecture, is the use of arrays in Cython. You can find in the back-up slides some information about this aspect. Typically, this is treated with typed memoryviews allowing an efficient memory access. The declaration of arrays is done as, *e.g.* for a two-dimensional double array `double[:,::1]`. The `:1` part indicates the fastest changing index, *i.e.* in `double[:,::1]` corresponds to a C-contiguous and `double[::1,:]` to a F-contiguous array. `double[:,:]` is compatible with C- and F-contiguous arrays.

Write a function that returns the element-wise squared version of the input array.

Compare it to the NumPy and the Python calculation in terms of CPU time.
Write a function that takes two two-dimensional arrays and calculates the matrix product. Again compare it to the NumPy matrix calculation in terms of speed.

If you still are interested in arrays in Cython, play around with the different alignments and test the impact on the speed of your code.

Exercise 10: Fortran

For those of you who still do it the Old School way: The folder `fortran` contains some files including a README that can be used as a starting point for interfacing Python code with Fortran77 and Fortran90 code. Have fun and play around!

Exercise 11: Importing

We saw in the lecture that we can use classes defined in Cython as base classes for classes implemented in Python. But such implementations have disadvantages in terms of speed.

In this exercise we see how we can develop new Cython-based modules leveraging functionalities implemented in other modules.

The folder `importing` contains two setup script: one to create a module based on the implementation of the function class for integrals discussed in the lecture (defined in `integrate.pyx`) and one to implement further functions leveraging the base class (defined in `polynomial.pyx`).

To do such an implementation a `.pxd` file needs to be created that specifies the interface of the functions and classes we are planing to integrate into new Cython modules. This is a similar concept as with header files in C++.

Run through the different steps and test the code with the corresponding notebook `PythonC_ex_11_Import.ipynb`.

Exercise 12: JAX

Have a look at the corresponding notebook `PythonC_ex_12_JAX.ipynb`.

Generate a 10'000×10'000 array of random variables and run some tests in terms of performance of polynomial functions (*e.g.* $x^3 + 2x^2 - 8x + 3$), $\exp(x)$, $\log(x)$, $\sin(x)$ or $1/x$ applied to JAX arrays versus NumPy arrays. How does applying a just-in-time compilation change the situation.

JAX also provides auto-differentiation functionality. This is in particular useful for nested functions as they are subject to the chain rule. Apply it to the function $f(x) = \exp(\sin(1/x))$ and then use it to calculate the derivatives between 0.01 and 100. Compare the performance of the auto-diff version from JAX to the numerical differentiation, but also to the analytical derivative $\mathrm{d}f/\mathrm{d}x = -\exp(\sin(1/x)) \cdot \cos(1/x) \cdot x^{-2}$. What do you observe in terms of accuracy and why?