COMPARISON OF DIFFERENT PYTHON DEVELOPMENT LIBRARIES TO SOLVE FLUID FLOW PROBLEMS

RESUMO – With the advance of programming languages, the availability of better hardware, and the emergence of new libraries for numerical development, it has been possible to use high level abstractions to prototype and solve complex problems in many fields of knowledge. In the context of numerical simulation using the Finite Volume Method, those new tools can be used, so that the developer is able to make experiments with less effort. This work compares several libraries from the Python programming language (Pure Python implementation, Cython, NumPy, Numba, Scipy, petsc4py, and pure C/C++), aiming to solve fluid flow problems and comparing execution performance from each tools.

1. INTRODUCTION

With the boost of computer performance over the years, interpreted languages such as Python became more and more popular for all types of software development. The arrival of those languages helped programmers to skip the compilation step on the daily development workflow, at the cost of a slower execution time. Programmers are, then, able to focus on solving the problem, avoiding premature optimization.

The Lid-Driven Cavity Problem is a well known benchmark problem (Ghia et al., 1982) solved in many ways in the literature, and extensively used for case comparisons. The present work intends to compare the computational implementation starting from the generalized transport equations, to solve the lid-driven cavity problem with the Finite Volume Method (Patankar, 1980), specifically using the Python libraries Cython, Numpy and Numba, also comparing with a C++ implementation. The Scipy and PETSc4Py libraries were also used for the system’s architecture.

2. DISCRETIZATION

To solve the 2D lid-driven cavity problem, involving Newtonian incompressible fluids, three conservation equations will be considered. The discretization process is based in Maliska (2004). The general transport equation for a generic variable, \( \phi \), is given by

\[
\frac{\partial (\rho \phi)}{\partial t} + \frac{\partial (\rho u \phi)}{\partial x} + \frac{\partial (\rho v \phi)}{\partial y} = \frac{\partial}{\partial x} \left( \Gamma_{\phi} \frac{\partial \phi}{\partial x} \right) + \frac{\partial}{\partial y} \left( \Gamma_{\phi} \frac{\partial \phi}{\partial y} \right) + S_{\phi} \tag{1}
\]

The discretization process is accomplished by integrating those equations along the control volume, assuming a staggered grid. For the conservation of mass, the integration over time and space, assuming equally-spaced domain, gives the following equation:

\[
u_{e} \Delta y - u_{w} \Delta y + v_{n} \Delta x - v_{s} \Delta x = 0 \tag{2}
\]
For the Navier-Stokes equations, the momentum balance is done using the fully implicit scheme, and the remaining derivatives are expanded using central differences. The transported properties in the advective terms are approximated using the Upwind scheme (Patankar, 1980). The final discretization of the momentum balance for velocity $u$ is:

$$
\left( \rho u_P - \rho^o u_P^o \right) \frac{\Delta x \Delta y}{\Delta t} + \rho u_e \left( (0.5 - \beta_{ue}) u_E + (0.5 + \beta_{ue}) u_P \right) \Delta y - \rho u_w \left( (0.5 - \beta_{uw}) u_P + (0.5 + \beta_{uw}) u_W \right) \Delta y + \rho v_n \left( (0.5 - \beta_{vn}) u_N + (0.5 + \beta_{vn}) u_P \right) \Delta x - \rho v_s \left( (0.5 - \beta_{vs}) u_P + (0.5 + \beta_{vs}) u_S \right) \Delta x = -(P_e - P_w) \Delta y + \mu \frac{u_E - u_P}{\Delta x} \Delta y - \mu \frac{u_P - u_W}{\Delta x} \Delta y + \mu \frac{u_N - u_P}{\Delta x} \Delta x - \mu \frac{u_P - u_S}{\Delta y} \Delta x
$$

(3)

Where $\beta_{uf}$ in face $f$ is determined depending on the velocities associated.

3. SYSTEM ARCHITECTURE

All implementations in this paper will be designed using the same architecture, shown in Fig. 1. The full code is not shown for the sake of simplification, but is available in Fischer e Manoel (2017). There are basically two steps. The first step is the setup, responsible for taking the problem’s input data and building all data structures necessary for the solver.

The second step is the solver itself. Its entry point is the Time Stepper. For each timestep, the solution is obtained by using the Newton Method (Edwards, 1994). The algorithm used is implemented in PETSC4Py and PETSC (Balay et al., 2008), where a Generalized Minimal Residual Method (GMRES) (Lucianetti, 2000) combined with an Incomplete LU factorization (ILU) preconditioner is used for solving the resulting linear system of each Newton iteration.

The residual function holds the physical problem implemented in the residual form $\vec{F}(\vec{X}) = \vec{0}$. For the experiment presented in this paper, the permutation of libraries only affects the residual function - No other aspect of the architecture is changed. It is then, possible to compare the execution time by measuring the time of the whole simulation.

![Figura 1 – System architecture overview](image-url)
4. CONCLUSION AND RESULTS

Before proceeding to the implementation comparisons, it is important to check the correctness of the implemented algorithm. Figure 2 shows the U velocities components at the X-centers of the mesh and V at the Y-centers of the mesh. Results are then compared with (Ghia et al., 1982). Ghia’s reference values are shown in X marks on the images. For the validation, an 129x129 mesh has been used with Reynolds number Re = 400, and both UDS and CDS interpolations for the advective term have been applied.

A C++ implementation of the residual function has been compared with the Python libraries, so that readers with no background on Python may have a source of reference. All implementation have been measured in the same machine, varying mesh size from 8 × 8 up to 128 × 128, with Reynolds number Re = 400. The mean time of 5 simulation runs have been computed to generate each point in the plot (Fig. 3).

Although expressing numerical problems using C++ language is becoming easier with the evolution of the language, the code-compile-execute workflow may still be a problem.
for real world applications. The Python code is the slower of them all, but it should again be noted that it is still suitable for early stages of development for fast code evaluation and experimentation. Numba and Cython may be good alternatives for numerical implementation that needs to have a good maintainability. NumPy’s vectorized code is also an alternative, but the programmers must be aware about expressing code in this format, as it can turn to be difficult to read and maintain in some cases.

5. REFERENCES


