Numerically Solving Navier Stokes equation with GPUs

Gašper Golob

June 2, 2024

1 Introduction

The goal of the project is to use the GPU to accelerate the execution of two methods for solving the Navier Stokes equation. Specifically the two methods that are being accelerated are the pressure projection and artificial compressibility method.

The performance and accuracy of the GPU based versions are then tested on the lid driven cavity problem. The lid driven cavity problem is the case where the fluid is in a square case, where one of the sides induces some movement. The case is schematically presented in the figure 1, from [2]. It is often used to test numerical methods for fluid simulations as the velocities converge towards

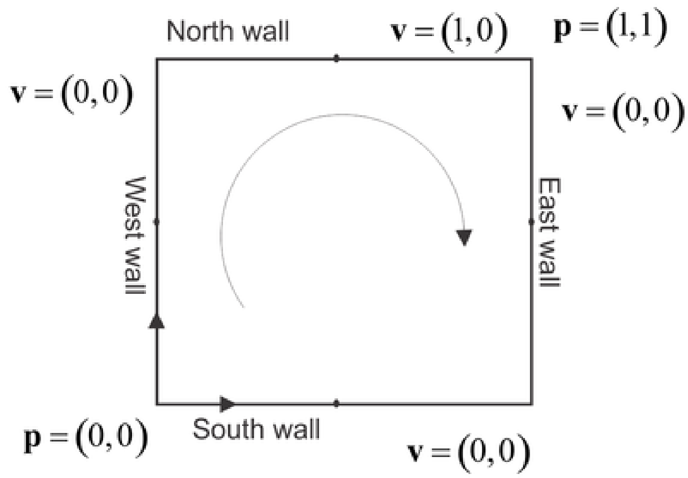

Figure 1: The schematic of the lid driven cavity problem.

some value, which can tell us if implemented method is stable.

The methods work iteratively, where at each time step, they compute new velocities using the Navier Stokes equation. Afterwards they correct the pressure using an assumed pressure volume invariant.

For the purposes of this project the details of solving the partial differential equations are not as important, since the project is mostly concerned with accelerating the solving of systems and various matrix operations.

The GPU versions are also compared to their base versions to gauge the level of improvement and to see if they remain accurate.

The project is made using C++, where the discretization of the differential equations is done using the medusa library [1]. To solve linear systems and work with matrices on the CPU the Eigen library is used. Similarly, to work with matrices and vectors on the GPU side, CUDA libraries such as cuSolver, cuSparse, cuDSS and Thrust are used.

2 Pressure projection method

The pressure correction method explicitly steps through time using the equation

$$\frac{\rho}{\Delta t}\left(\vec{v}(t+\Delta t) - \vec{v}(t)\right) = -\boldsymbol{\nabla} p + \mathcal{F}(\vec{v}(t)), \tag{1}$$

where $t,\ \Delta t,\ \vec{v},\ p,\ \rho,\ \mathcal{F}$ represent the time, time step, velocity field, pressure field, density and all the non-pressure forces. Every time step is split into two parts, the first gives us an intermediate velocity which does not consider the pressure gradient

$$\vec{v}^{*} = \vec{v}(t) + \frac{\Delta t}{\rho}\mathcal{F}(\vec{v}(t)), \tag{2}$$

while the second applies the pressure correction to the intermediate step

$$\vec{v}(t+\Delta t) = \vec{v}^{*} - \frac{\Delta t}{\rho}\boldsymbol{\nabla} p. \tag{3}$$

To compute the pressure correction we use the equality

$$\boldsymbol{\nabla}\cdot\vec{v}^{*} = \frac{\Delta t}{\rho}\nabla^2 p. \tag{4}$$

The equations are solved with the help of matrices generated using the medusa library. For the purposes of this project we are mostly concerned with solving the linear systems belonging to the equations 2 and 4.

For equation 2 the matrix changes at each time step, so we need a there isn't any structure we could exploit to speed up the execution. On the other hand the matrix belonging to the equation 4 remains fixed, so we need a system solver that leverages this. This is especially important, since the pressure correction can be executed up to 100 times per time step.

2.1 Solvers

All solvers are initialized using the Eigen `SparseMatrix` type for sparse matrices. To solve a system they receive the right-hand side vector which has the Eigen type `VectorXd`, which is a dense vector whose elements have the type `double`. Ultimately they return a vector of the same type.

2.1.1 SparseLU solver

The base CPU version uses the Eigen library to solve the linear systems. For both of the systems the SparseLU solver is used, which implements the supernodal LU factorization of the matrix, and then solves the system using the computed triangular matrices.

2.1.2 QR solver

The QR solver uses the function `cusolverSpDcsrlsvqr` from the cuSolverSp library. The function takes as arguments a sparse matrix A in the compressed sparse rows format alongside two dense vectors x and b. It solves the system $Ax = b$ on the GPU. In the background the method computes a sparse QR factorization of the matrix A, while also providing some reordering, to minimize zero fill in. For the purposes of this project, the `symrcm` reordering scheme is used.

2.1.3 cuDSS solver

The cuDSS solver is the main solver currently implemented in the cuDSS library. Since the matrix doesn't have any extra properties it uses the LDU decomposition. Same as the previous solver the matrix is in the CSR format, while the right-hand side and solution vectors are dense. All the matrices and vectors are transferred to the GPU where the decomposition is computed and the systems are solved.

2.1.4 RF solver

The refactorization solver is meant to sequentially solve multiple systems of the form $Ax_i = b_i$ for multiple vectors x_i, b_i and a fixed matrix A.

The first time it has to solve a system it uses the `cusolverSP_lowlevel_preview` library. This library first computes the sparse LU decomposition (with some reordering) of the matrix and then solves the first system.

Afterwards the sparse LU decomposition is extracted and passed to the `cuSolverRF` library, which can then use it to solve later systems.

The first iteration is done completely on the CPU, as the `cusolverSP_lowlevel_preview` library currently lacks GPU versions of the functions. Later, the systems are solved using the functions from `cuSolverRF` which works on the GPU.

2.2 Benchmarks

To speed up the solving of the system belonging to equation 2 we use the QR and cuDSS solvers, while the second system is only accelerated using the RF solver, as this is the only one that allows for preprocessing. The execution time of the programs, when using different solvers can be seen in the figure 2.

We can see that the lid driven cavity version which is the reference CPU implementation is the slowest, except possibly right at the beginning, where there is still not that much parallelism, so the CPU better single thread performance gives it an advantage.

The second-best case is when we use the QR solver to solve the system of 2. It starts of a bit slower than the CPU version but quickly becomes faster, as the size of the problem and the potential for parallelism increases.

The best case is when using the cuDSS solver to solve the system with the equation 2. For large enough N the performance increase seem to be about 100x, which is far better than even the QR solver. This could be either because the cuDSS library is made for a newer version of CUDA and so uses more modern features or maybe the QR decomposition has more zero fill-in which would result in a slower execution time.

The last case uses the cuDSS solver for the system belonging to 2 and the RF solver for the system belonging to 4. Unfortunately it seems that solving the LU system on the GPU slows down the execution, which makes some sense as solving triangular systems is a very sequential operation.

Regardless, it might prove useful to run the benchmarks on even larger cases, as there might be some performance improvements for very large N. The limiting factor here might however be, that the LU decomposition is done on the CPU, so the setup time might take too long.

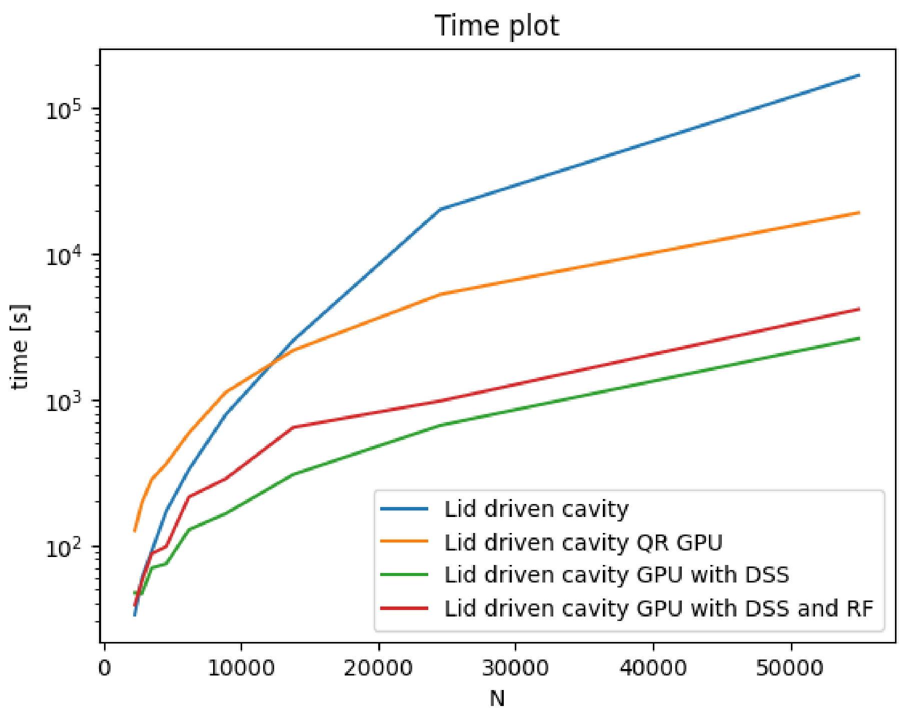

Figure 2: Graph of times to simulate 50 seconds of the lid driven cavity problem using the pressure projection method, for different problem sizes.

From the figure 4 at the end of the report we can see that the method converges toward some fixed value, which is what we expected. As a sanity check we can also look at 7 which tells us that the final velocities at the middle line match for all implementations.

3 Artificial compressibility method

For the artificial compressibility method we use the explicit Euler method to solve

$$\vec{v}^* = \vec{v} + \Delta t \left(\nu \nabla^2 \vec{v} - \vec{v} \cdot \nabla \vec{v} + \vec{g}\right), \tag{5}$$

for the intermediate step, where Δt, $\vec{v}$, ν are the time step, velocity field and viscosity.

Afterwards we iteratively compute the new pressure and velocity using

$$\vec{p} \leftarrow \vec{p} - \Delta t C^2 \rho(\nabla \cdot \vec{v}), \tag{6}$$

$$\vec{v} \leftarrow \vec{v}^* - \frac{\Delta t}{\rho} \nabla \vec{p}, \tag{7}$$

where p is the pressure field and ρ is the density. The magnitude of the artificial compressibility is determined using

$$C = \beta \max(\max_i(\|\vec{v_i}\|_2), \|\vec{v}_{ref}\|_2), \tag{8}$$

where β is the compressibility of the fluid.

From these equations some matrices are constructed for which multiplication needs to be sped up. Additionally, some element-wise operations can also be done on the GPU to save time. In fact, after the matrices are constructed, the rest of the work can be done on the GPU. To achieve this we need the operations described in the next subsection.

3.1 Basic GPU operations and types

For all operations sparse matrices are given in the compressed sparse row format and the vectors are the usual dense vectors given as arrays on the GPU.

3.1.1 VectorGPU and MatrixGPU

These are some basic wrapper classes that convert to and from their Eigen counterparts. Data that is passed to them is transferred to the GPU, where it can be used with other functions.

3.1.2 Matrix multiplication

To multiply sparse matrices with vectors the cuSparse library is used. More specifically the `cusparseSpMV` function is used which takes a matrix in the compressed sparse rows format, a dense vector and returns a dense solution vector. Some multiplication data is initialized before the simulation as it speeds up the later multiplications.

3.1.3 Transforms and reduces

For element-wise operations on dense vectors thrust's `transform` and `transform_reduce` functions are used.

The first use case is to allow element-wise vector multiplication, for which we can use the `transform` function and the built-in `multiplies` operator.

The second use case is to define a new operator `axpy_functor` which is initialized using some value α and which then for given vectors x and y computes $y = \alpha x + y$. The operator is again applied using the `transform` function.

Another case is the definition of the operator `abs_functor` which for a given vector computes the element-wise absolute value.

The last two operators that are defined are `u_tuple_functor` and `tuple_max_functor`, which are then used with the `transform_reduce` function. The `u_tuple_functor` is initialized using an array which represents a vector field. Running the functor on some set of indices will return an array of pairs of said indices and the corresponding second coordinate values. Then the `tuple_max_functor` finds the maximum value among the values which have corresponding indices. This is used to find the maximum velocity in the y direction at the middle line of the lid driven cavity case.

Another function from Thrust that is used is `max_element` which simply returns the maximum element of an array.

3.2 Benchmarks

We use the previously described functions to port the CPU code to the GPU. Additionally, we also use CUDA streams to execute multiple operations on the graphics card at once, where this is possible.

We can see the execution times of the base CPU version in comparison the GPU version on figure 3. Once again the CPU version is better at the smallest values of N, but it quickly becomes better as N grows larger.

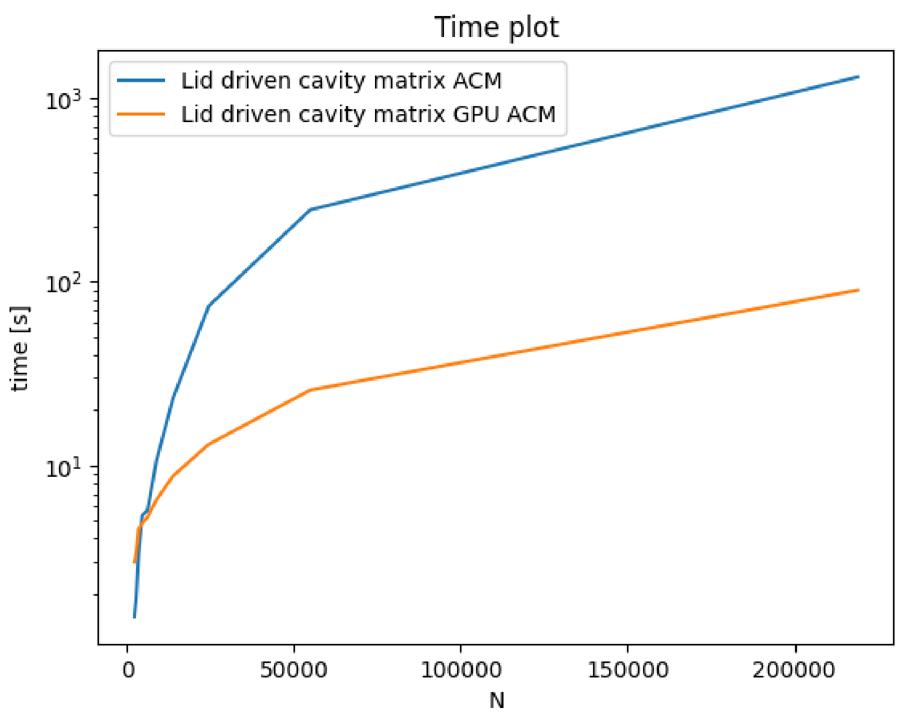

Figure 3: Graph of times to simulate 50 seconds of the lid driven cavity problem using the artificial compressibility method, for different problem sizes.

Much like for the implicit method we can also observe from 6 that the methods converge towards some fixed value. Their middle line velocities also match, which we can see from 7.

4 Further work

While the project is nearing completion there is some more work to be done. For an instance it would be good to benchmark the performance for some even larger N, to see if the trends continue as the amount of data on the GPU increases. It might also make sense to do some more detailed benchmarks to see how fast particular parts of the code are.

Besides that, the repository still needs to be cleaned up, and the code should be made easier to run as it is currently setup specifically for the server, which was used for the project's development.

References

[1] Jure Slak and Gregor Kosec. 2021. Medusa: A C++ Library for Solving PDEs Using Strong Form Mesh-free Methods. ACM Trans. Math. Softw. 47, 3, Article 28 (September 2021), 25 pages. https://doi.org/10.1145/3450966

[2] G. Kosec; A local numerical solution of a fluid-flow problem on an irregular domain, Advances in engineering software, vol. 120, 2018 DOI: 10.1016/j.advengsoft.2016.05.010

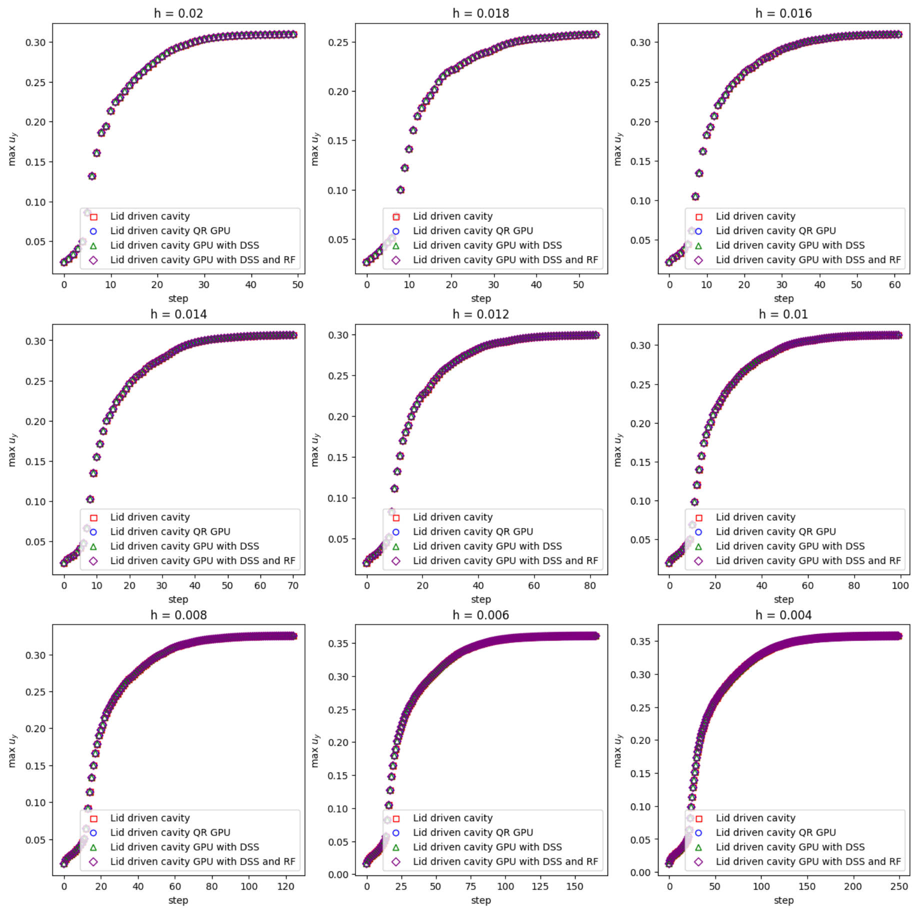

Figure 4: Graph of max u_y in regard to number of steps taken using the implicit method.

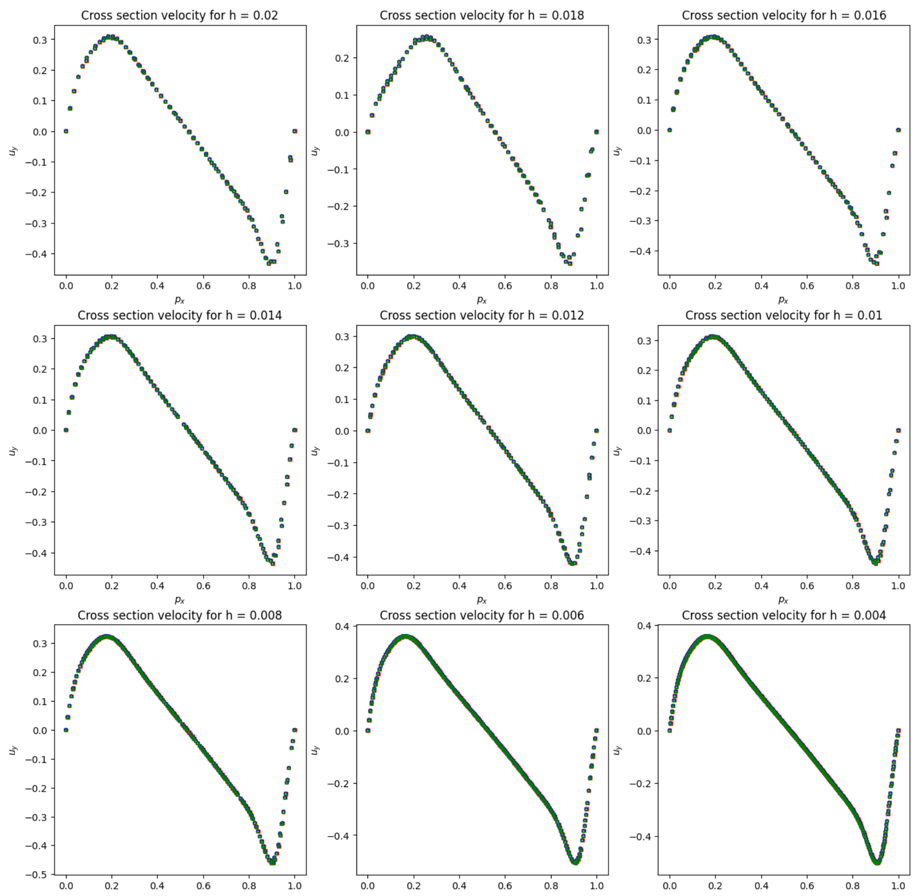

Figure 5: Graph of velocities at the middle line after 50 seconds using the implicit method.

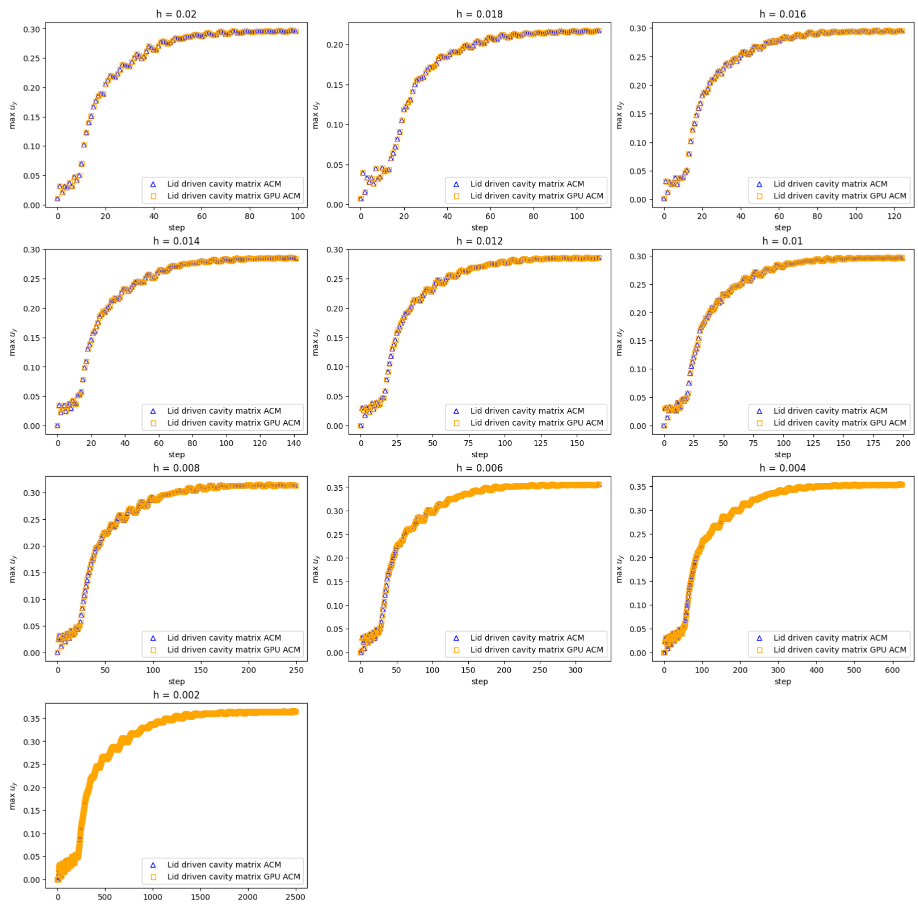

Figure 6: Graph of max u_y in regard to number of steps taken using the explicit method.

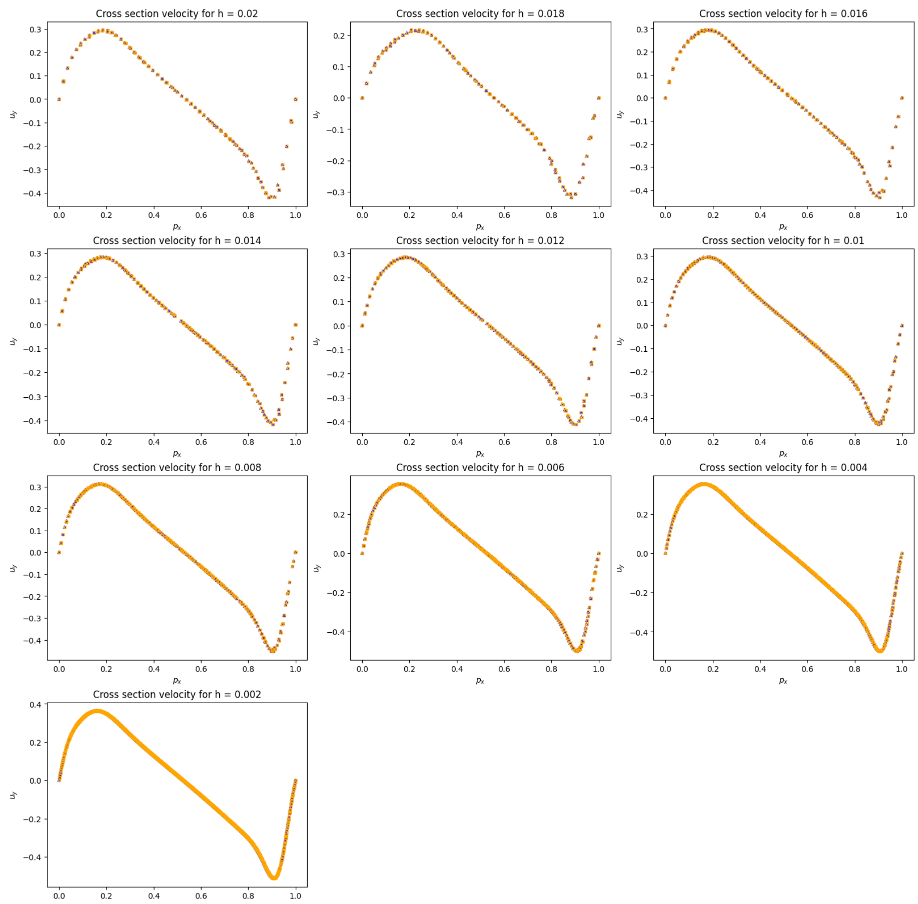

Figure 7: Graph of velocities at the middle line after 50 seconds using the explicit method.