A Linear Acoustic Perturbation Solver Using PETSc

Ashton Cole
AVC687

April 30, 2024

1 Introduction

This parallel computing final project implements a two-dimensional finite difference solver, which attempts to model the propagation of acoustic perturbations across a steady, approximately-incompressible flow. The solver is written in C, leveraging the PETSc linear algebra library.

The problem is inspired by my senior design project. In it, we were primarily tasked with physically testing an ultrasonic fluid flow rate sensor. The sensor operates with using transducers positioned at the sides of a pipe, with one upstream of the other. As they send signals back and forth, the waves move at the speed of sound relative to the flowing fluid medium. Because of this, the downstream signal travels faster than the upstream one, resulting in a measurable transit time difference, which can be used to infer the average fluid velocity.

In addition, we also wanted to model this effect computationally. First, we tried compressible flow simulations in OpenFOAM. However, when we tried adding viscosity, the solution became unstable. We reached out to Dr. Fabrizio Bisetti, a professor specializing in Computational Fluid Dynamics, and he suggested we try deriving our own linear system of partial differential equations to solve. This would be simpler, and hopefully easier, to solve than the Navier-Stokes equations.

Although we were not able to complete a Python-based finite element method solver during our project timeline, we were able to write a serial finite difference solver for the system of equations in one dimension, which proved that they work. I also decided to take on this same problem for this class' final project. Solving a system of partial differential equations is an ideal candidate for parallel programming, since the simulations require large matrix-vector computations. In addition, the PETSc library provides a convenient interface for parallel linear algebra operations in C. Instead of the finite element method, I again opted for a simpler finite difference method solver, but now on a two-dimensional structured grid. The program solves the equations in time explicitly and writes them to files within a case directory.

Unfortunately, the results were not particularly satisfying. In two dimensions, it seems that the solution very easily becomes unstable. However, speedup tests were at least able to show that the parallelization through PETSc offers an advantage for large systems.

1.1 Setup and Execution

In addition to a standard C compiler, this project requires an installation of PETSc, and by extension an implementation of MPI. Importantly, the `$PETSC_DIR` and `$PETSC_ARCH` environment variables should be set to the PETSc intallation location and build subdirectory, respectively.

To set up a particular case, constant values are defined in `driver.c`, while functions are defined in `driver_functions.c`. Once these are configured appropriately, the program may be built with CMake. This program may then either be executed in serial or parallel, using appropriate terminal commands. The case output directory will be written on a path relative to the present working directory.

2 Methodology

In this section, the derivation and discretization of the system of partial differential equations are discussed, as well as the implementation of the solver in C.

2.1 Derivation of Partial Differential Equations

After extended discussion with Dr. Bisetti, we decided to take the rather ambitious and exploratory route of deriving our own equations to solve. He presented the following idea, based on an acoustics textbook: start with the Euler equations, i.e. the inviscid, but compressible form of the Navier-Stokes equations (Equations 1 and 2). Assume that the velocity and density are the sum of a steady, compressible, viscous flow, and far smaller unsteady acoustic fluctuations. An isentropic pressure relation (Equation 3) replaces the energy equation. This essentially means that we want to neglect nonlinear effects and dissapation over long distances, focusing on the transit time difference. Finally, the fluid is assumed to be an ideal gas. This is not representative of liquids like what our sensor would actually measure, but it was convenient to introduce the speed of sound into the system (Equation 4). If anything, it would be just be a good place to start, and hopefully yield some meaningful results.

$$\frac{\partial \rho}{\partial t} + \boldsymbol{\nabla} \cdot (\rho \mathbf{u}) = 0 \tag{1}$$

$$\frac{\partial (\rho \mathbf{u})}{\partial t} + \boldsymbol{\nabla} \cdot (\rho \mathbf{u} \otimes \mathbf{u}) = -\boldsymbol{\nabla} p \tag{2}$$

$$p = C\rho^{\gamma} \tag{3}$$

$$c = \sqrt{\gamma \frac{p}{\rho}} \tag{4}$$

The following variable conventions are used for the above equations.

- $\mathbf{x}$: position vector
- t: time
- $\rho(\mathbf{x}, t) = \overline{\rho} + \rho'(\mathbf{x}, t)$: density scalar field of the fluid, decomposed into a known constant $\overline{\rho}$ and a small acoustic perturbation ρ', i.e. $\overline{\rho} \gg \rho'$
- $\mathbf{u}(\mathbf{x}, t) = \overline{\mathbf{u}}(\mathbf{x}) + \mathbf{u}'(\mathbf{x}, t)$: velocity vector field of the fluid, decomposed into a known steady $\overline{\mathbf{u}}$ and a small acoustic perturbation $\mathbf{u}'$, i.e. $\overline{\mathbf{u}} \gg \mathbf{u}'$
- $p(\mathbf{x}, t)$: pressure scalar field of the fluid, not decomposed for this derivation
- γ: ratio of specific heats of the fluid
- c: the ideal-gas speed of sound of the fluid

After substitutions, expansions, and neglecting small terms, the following linear, hyperbolic system of partial differential equations results (Equations 5 and 6).

$$\frac{\partial \rho'}{\partial t} + \overline{\rho} \boldsymbol{\nabla} \cdot \mathbf{u}' + \overline{\mathbf{u}} \cdot \boldsymbol{\nabla} \rho' = 0 \tag{5}$$

$$\frac{\partial \mathbf{u}'}{\partial t} + \overline{\mathbf{u}} \cdot \boldsymbol{\nabla} \mathbf{u}' + \frac{c^2}{\overline{\rho}} \boldsymbol{\nabla} \rho' = -\overline{\mathbf{u}} \cdot \boldsymbol{\nabla} \overline{\mathbf{u}} \tag{6}$$

2.2 Problem Definition

The problem is to solve the above system of equations in two dimensions over a rectangular domain $\Omega = [x_a, x_b] \times [y_a, y_b]$ from time t_a to time t_b. The solution has an initial state $\rho'(\mathbf{x}, t_a) = \rho'_0(\mathbf{x})$ and $\mathbf{u}'(\mathbf{x}, t_a) = \mathbf{u}'_0(\mathbf{x})$. Each of the four boundaries has Dirichlet conditions imposed on ρ' and $\mathbf{u}'$, which are permitted to be functions of both space and time. Finally, the incompressible flow field solutions $\overline{\rho}$ and $\overline{\mathbf{u}}$ are prescribed.

2.3 Discrete Formulation

The solution is solved discretely at rectilinear, regularly-spaced grid points (x_i, y_j). The solution variable values at these points are combined into a single vector $\mathbf{z} = (\rho'_{0,0}, u'_{0,0}, v'_{0,0}, \rho'_{0,1}, u'_{0,1}, v'_{0,1}, \dots)$, with an indexing method $e = 3n_y i + 3j + k$, where k is the variable index and n_y is the number of solution points along the y-direction. This discretization allows the system of partial differential equations, deploying second-order-accurate central finite differences, to be approximately rewritten as a system of ordinary differential equations (Equation 7). The exception is at the boundary nodes, but these equations will be adjusted later.

$$\frac{\mathrm{d}\mathbf{z}}{\mathrm{d}t} = \mathbf{A}\mathbf{z} + \mathbf{b} \tag{7}$$

Then, central finite differences are again used to approximate the time derivative. This allows for an explicit definition for $\mathbf{z}$ at the next time step n (Equation 9).

$$\frac{\mathbf{z}^n - \mathbf{z}^{n-2}}{\Delta t} = \mathbf{A}\mathbf{z}^{n-1} + \mathbf{b} \tag{8}$$

$$\mathbf{z}^n = \mathbf{z}^{n-2} + \Delta t(\mathbf{A}\mathbf{z}^{n-1} + \mathbf{b}) \tag{9}$$

Because this method requires two solution steps to calculate, the first new step is calculated with the forward Euler method.

2.4 Implementation with PETSc

The Portable, Extensible Toolkit for Scientific Computation (PETSc) library is used to conduct all necessary matrix-vector operations in C. As an added benefit, it supports computations in parallel, being built on top of the Message-Passing Interface (MPI) standard. Thus, this solver supports vectors, matrices, and calculations spread out across multiple concurrently-operating processors, expediting the solution of large problems.

The whole program uses the SPMD model characteristic of MPI parallelization, with each processor executes the same instructions. Some commands, like printing to the terminal or looping for vector and matrix value assignment, are only executed in serial on the lowest-rank processor.

It is worth noting that the inter-processor communication routines for vector-matrix operations are hidden under the PETSc interface. For example, consider the following snippet of code from `_build_matrix()` in `acoustic_problem.c` showing specifically the sequence of commands used to assemble the **A** matrix.

```
Mat A;
PetscCall(MatCreate(comm, &A));
PetscCall(MatSetType(A, MATAIJ)); // Or MATMPIAIJ
PetscInt matrix_size = ac.sd.nx * ac.sd.ny * 3;
PetscCall(MatSetSizes(A, PETSC_DECIDE, PETSC_DECIDE, matrix_size, matrix_size));

// Loop through domain, bc, and set values by adding
if (procno == 0) {
        // Call MatSetValues()
}

PetscCall(MatAssemblyBegin(A, MAT_FINAL_ASSEMBLY));
PetscCall(MatAssemblyEnd(A, MAT_FINAL_ASSEMBLY));
```

Essentially, sections of the matrix are allocated on each process, but all of the values are set on the 0-rank process. These have to be distributed appropriately using message passing. The assembly begin and end functions clearly mirror how, in MPI, sending and receiving have to be coordinated with `MPI_isend()` and `MPI_irecv()` across processes to prevent blocking. However, none of these MPI commands have to be actually used. They are already implemented within the PETSc abstract interface.

2.5 Output

The main program outputs the results to a case directory. Within this directory, simulation parameters are written to a JSON metatdata file, while the time and solution vector are written to time step subdirectories. Visualizations are generated using a Python postprocessing script.

3 Results

Both visual inspection of the results and parallel speedup tests are used to evaluate the program.

3.1 Visualizations

First, a two-dimensional version of a wave pulse considered in one dimension is tested (Figure 1). The underlying flow is $\overline{\rho} = 1$ and $\overline{\mathbf{u}} = (0.1, 0)$. Unfortunately, unlike in the one-dimensional case with the same parameters, oscillations quickly develop.

Then, a case is attempted with a circle of positive perturbation at the center of the domain (Figure 2). Here too, instabilities quickly take over.

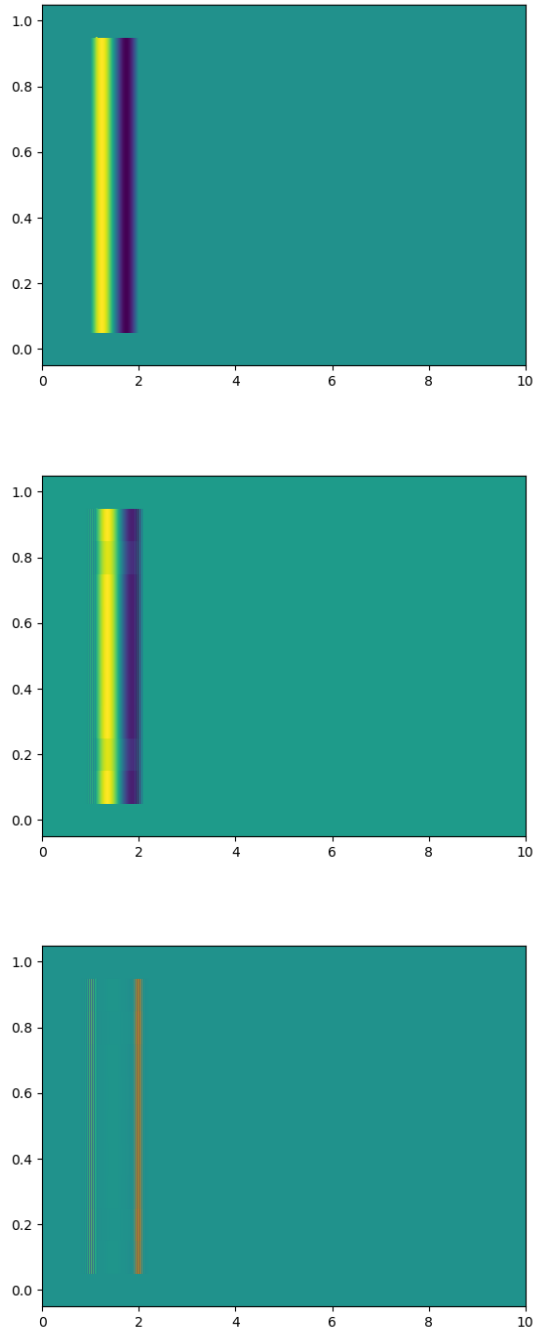

Figure 1: Wave Pulse: Plots for ρ' at times $t = 0, 0.1, 0.2$ s.

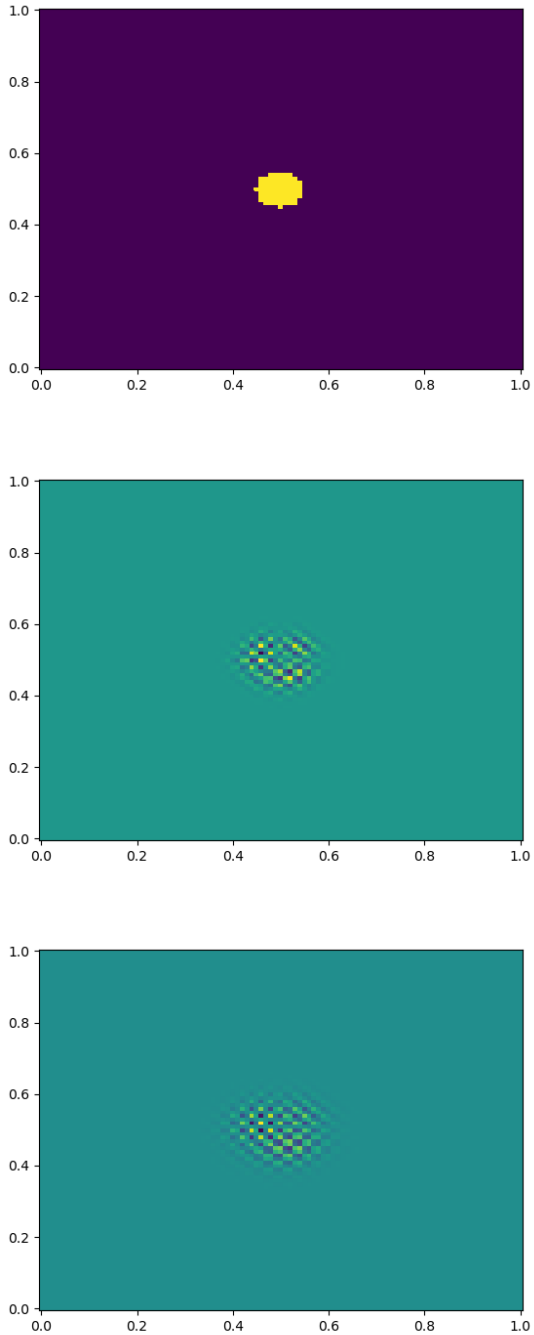

Figure 2: Circular Pulse: Plots for ρ' at times $t = 0, 0.1, 0.2$ s.

3.2 Performance tests

The second case was re-executed at different levels of refinement, with different numbers of processors, with and without file output, for parallel speedup tests.

Table 1: Performance tests for a simple case.

Processes	Number of Lateral Grid Points (n_x, n_y)				
	101	202	303	404	505
	With file writing enabled				
1	13.239 s	52.324 s	116.074 s		
2	13.357 s	51.957 s	116.437 s		
4	13.260 s	53.227 s	114.786 s		
8	14.563 s	56.279 s	128.568 s		
	With file writing disabled				
1	1.310 s	3.781 s	7.705 s	13.343 s	20.549 s
2	1.167 s	3.593 s	7.557 s	13.038 s	19.834 s
4	1.034 s	2.955 s	6.118 s	10.583 s	16.110 s
8	1.123 s	3.138 s	6.347 s	10.791 s	16.176 s

4 Conclusions

Unfortunately, this two-dimensional solver did not provide the satisfying results that were hoped for. The matrix assembly code and matrix itself were inspected carefully to ensure that they were built as expected. The oscillations suggest that the issue is one of numerical stability. It is possible that additional refinements with even smaller time steps will yield stable solutions, or the system may need to be solved with a different, perhaps implicit, method.

However, at least some insights may be gleaned from the parallelization tests. At around $n_x = n_y = 303$, speedup advantage from parallel operations starts to appear. However, the biggest impact is from disabling file writing. This is because, when writing to file, the vector must be ordered correctly. Each process must write the vector in sequence, bringing the execution time back to the level of a sequential program.