Computational Aerodynamics Lab Summer Student

Supervisor: Dr. David W. Zingg
Eric Lefort

May 5th 2022—August 26th 2022

Contents

This summer, I worked on a series of smaller projects aimed at both improving existing tools and exploring novel tools and approaches for creating or helping to create meshes.

1. Comparing performance between Diablo and Optima2D for 2D problems
2. Using coarse unstructured mesh generator to define mesh blocks
3. Numerical methods for solving PDEs

1 Diablo vs. Optima2D

1.1 Background

Diablo is the group's current flow solver, which is used within the broader Diablo solver-optimizer package to perform various types of 2D or 3D solves and optimizations. Optima2D is an older software developed within the lab group by Nemec. It is a purely 2D flow solver and optimizer.

1.2 Motivation

Initial review of both softwares informed us that our current flow solver, Diablo is slower when running comparable cases. That is to say that it consumes more CPU time, even if wall times may be shorter due to parallelism, compared to the single-processor solver that is Optima. For simple cases, Optima was outperforming Diablo considerably.

Seeing our older flow solver outperform the newer one under a variety of conditions raises the idea that substantial improvement is possible.

1.3 Objective

The objective of this project was therefore to compare the performance of Diablo and Optima2D in a variety of 2D cases. This should inform future changes to Diablo, allowing for its improvement and reducing computing times, thus accelerating future flow solver computations and research as a whole.

One factor that makes comparison difficult between to two solvers was that Diablo does not allow a single block interface to be split to use different boundary conditions, meaning it is not possible to run a single-block C-mesh simulation. We must therefore use multiblock grids for the Diablo simulations, a difference from those used with Optima2D. As it scales with higher number of blocks and processors, the solver exhibits some amount of parallel slowdown, which may be difficult to quantify accurately.

Optima2D, being a single-threaded software, requires single-block meshes and performs the flow calculations using a single processor. This means that we cannot use the same mesh for both software. Initial tests with Diablo showed that it performed poorly and inconsistently when configured to solve a multiblock mesh with single-threaded execution.

1.4 Methodology

To be able to compare the two pieces of software, we began by studying Diablo's performance at various degrees of parallelization and with varying numbers of blocks and processors to collect data on parallel slowdown and scaling to higher block counts.

Because Diablo does not allow a single block interface to be split to use different boundary conditions, it is not possible to run a single-block C-mesh simulation. For this reason, we decided to use 4 block meshes for Diablo with the majority of our tests.

We considered running both Diablo and Optima2D on a single processor, to allow for more comparable results to be obtained. To verify that this approach would work well with Diablo, we tested the simulation performance of a multiblock mesh using different numbers of processors, obtaining the following results.

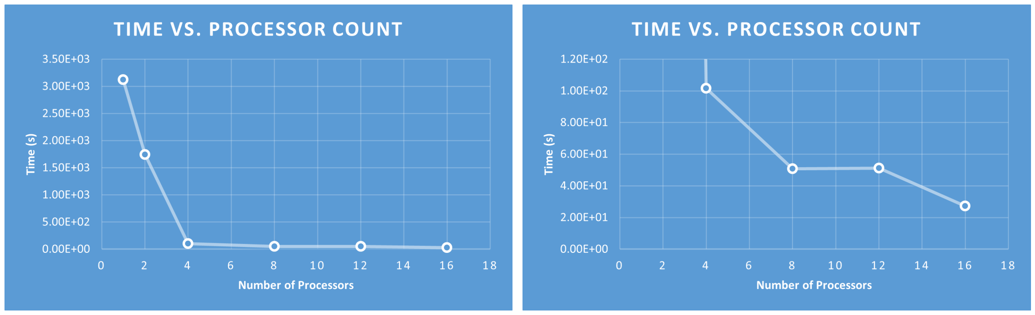


Figure 1: Wall time vs. processor count on 16 block mesh. Right side contains magnified view of same results.

As in Figure 1, Diablo exhibits highly inefficient, nonlinear behavior when decreasing processor count to 1 and 2.

Having established that it would not be ideal to run Diablo on a single processor, we decided to use a 4 block mesh on 4 processors for all further comparisons. Though that complicates comparisons between the results obtained from the two different pieces of software, we will be expecting quadruple speed from Diablo as a baseline, while keeping in mind that it will probably be slightly worse for reason of imperfect scaling and parallel slowdown.

Moving forwards, we decided to study how Diablo scales with increasing block counts in meshes. As a baseline, we expect to get slightly worse than linear scaling at low block counts, with this drop-off becoming more pronounced as the number of blocks becomes large.

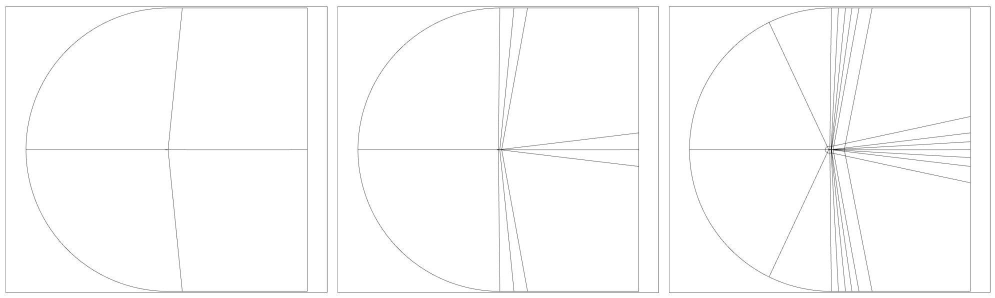


Figure 2: Block topology for Diablo grids with 4, 16, and 64 blocks.

As shown in Figure 3, total processor time (measured in right-hand-side evaluation equivalents) increases substantially with increasing block/processor count.

Following our study of Diablo, we created 2 NACA 0012 c-meshes for both Diablo and Optima2D. The meshes for Diablo had four blocks, while those for Optima2D had only 1. This was the only significant difference, as the blocking topologies were very similar. The first meshes had 18 361 nodes while the second set had 58 201.

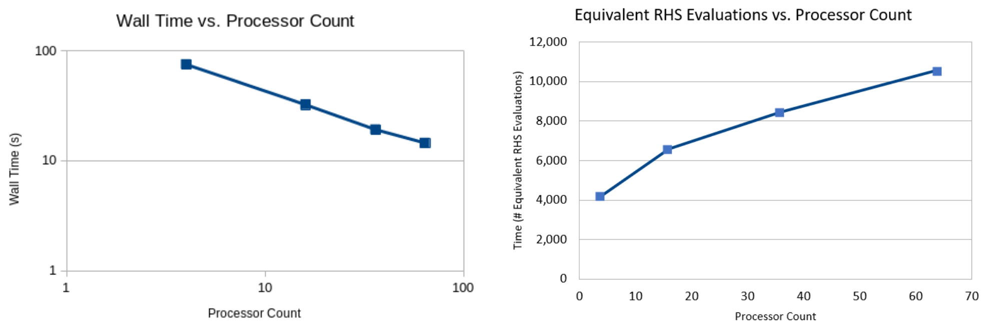


Figure 3: Plots showing nonlinear scaling. Flow solution requires significantly more total resources as processor count increases.

1.5 Results

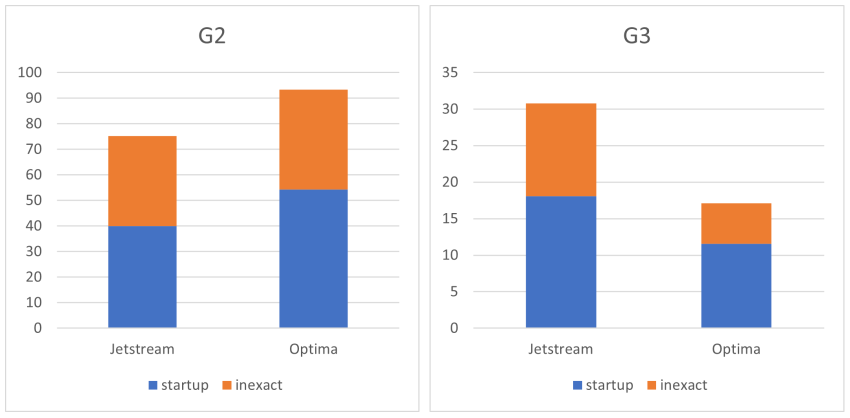


Figure 4: Wall time for flow solution on grids G2 (58 201 nodes) and G3, (18 361 nodes) separated into startup and inexact phases.

Optima did significantly outperform Diablo in our tests. For the coarser mesh (G3), it performed nearly twice as fast as Diablo, while using one processor compared to the four being used by Diablo. With the finer grid (G2), Diablo ran faster then Optima, however, its total CPU time consumed was still significantly higher due to using four processors.

Note that Diablo performed comparatively better with the finer grid, suggesting that it may be more well suited to slightly different problems than Optima is.

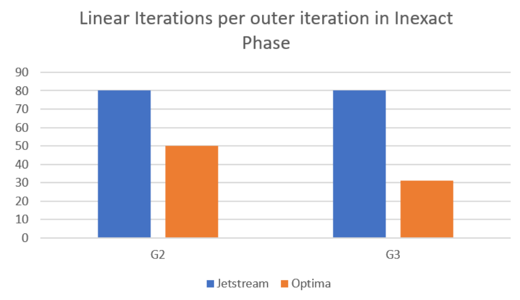


Figure 5: Linear iterations per outer iteration for both Diablo and Optima on both grids.

Unfortunately, we did not have to opportunity to closely examine factors of the performance differences.

1.6 Conclusion & Future Work

Despite not having thoroughly examined the factors in the performance differences, we have identified a few possible sources of slowdown:

1. Root node ordering. Optima and Diablo should both use reverse Cuthill-McKee algorithm, beginning with the node that is furthest downstream, to most efficiently order the nodes in a block. Improper selection of the root node could cause performance issues such as those we are seeing in Diablo.

2. Parallel slowdown and scaling – Schur preconditioner

It would also be useful to repeat the comparative study between Diablo and Optima using an O-mesh, which can be made to be run in both Diablo and Optima with a single-block mesh.

2 Automated Unstructured Block Generation

2.1 Background

The UTIAS Computational Aerodynamics Group's flow solver, Diablo, can perform computations with meshes having arbitrary block arrangements provided the following requirement is met: the blocking is conformal, meaning all block faces interface to exactly one other block face or to a single boundary.

Currently, we use Ansys ICEM as our primary grid generation tool. This tool allows the creation of block structured meshes.

2.2 Motivation

Defining the mesh blocks requires topological specification, which is slow and demands a high level of skill for unconventional aircraft and other complex 3D geometries.

The ability to automate the blocking process would save considerable time.

2.3 Objective

This project intended to make use of an existing automated unstructured mesh generator to create a very coarse, unstructured, hexahedral mesh. This could be used in conjunction with ICEM's *Create blocking from unstructured mesh* import option to define the blocking topology for a mesh.

2.4 Methodology

Initially, we acquired an academic license to the Cadence OMNIS software package (formerly Numeca OMNIS), specifically their unstructured meshing tool, Hexpress. To determine the software's fitness for our purposes, we began by running tutorial cases.

2.5 Results

After generating a 3D grid for the LANN wing, we noticed that the software generated meshes where multiple blocks can interface to a single block, making the mesh non-conformal. (possessing hanging nodes)

We conducted further tests using very low cell counts to more closely simulate realistic block counts, but had difficulty getting the software to generate valid blocks, leading us to hit a lower bound around 500-1000 with the fairly simple geometries we were using. Furthermore, the blocks were highly skewed and their shapes were not well suited to refinement of the mesh while approaching the airfoil.

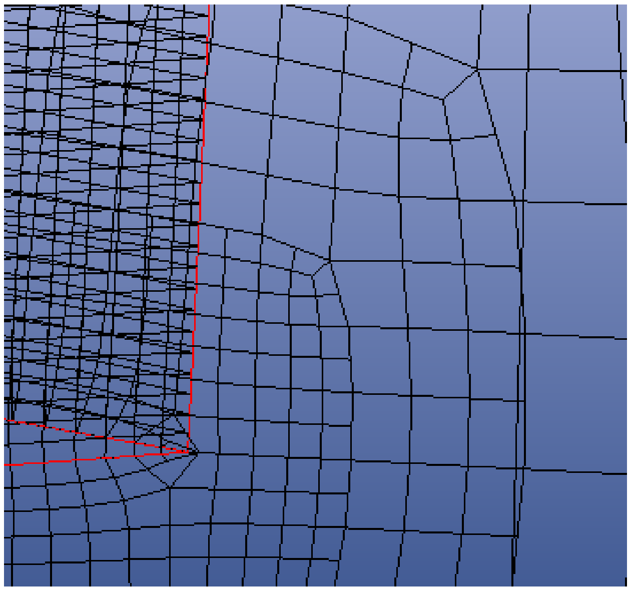


Figure 6: A 3D mesh of an extruded NACA0012, made with Hexpress. Hanging nodes are visible.

2.6 Conclusion

All in all, the results we obtained from our tests using Hexpress demonstrate that the software is not ideal for our purposes. After contacting customer service to inquire about the software's ability to generate conformal meshes, we confirmed that it is not possible.

Aside from this, Hexpress is an older software which is no longer being maintained by Cadence.

2.7 Future Work

As it is no longer updating its older Hexpress software, Cadence had shifted its focus to newer tools, including its Pointwise meshing tool. While it is generally a mixed unstructured mesh generator, it can be configured to produce purely Hexahedral meshes.

After some discussion with customer service, Pointwise seems to have multiple features which may prove useful and warrant further examination, including the ability to take an existing mesh and a larger domain and generate a mesh to bridge the gap, while generating nodes to match boundary nodes of the first mesh. These features may allow us to generate hybrid meshes (with both structured and unstructured parts) that are able to be solved using our current tools.

3 Numerical Methods for Solving the Linear Convection Equation

3.1 Background

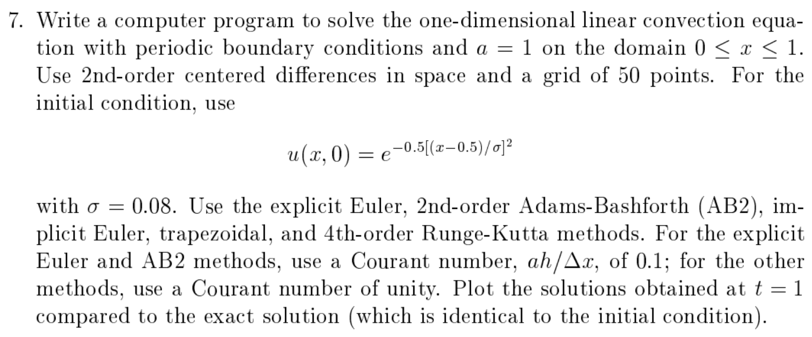
As a summer student having just completed year 2 of a bachelor's degree in engineering science, I had a limited understanding of numerical methods that are useful in computational fluid dynamics. My knowledge was mostly limited to Explicit Euler's and Runge-Kutta methods.

Given my own lack of background in numerical methods and finite difference methods specifically, we determined that it would be highly valuable for me to study *Fundamentals of computational fluid dynamics* by Lomax, Pulliam, and Zingg.

3.2 Objective

The objective of this project was to read and understand sections 3 and 4 as well as most of section 6. Following this, I would solve problems from section involving the numerical solution of PDEs on a 1D grid of a certain size using different boundary conditions and time-marching methods.

There was a particular focus on problem 6.7.



7. Write a computer program to solve the one-dimensional linear convection equation with periodic boundary conditions and $a = 1$ on the domain $0 \leq x \leq 1$. Use 2nd-order centered differences in space and a grid of 50 points. For the initial condition, use

$$u(x,0) = e^{-0.5[(x-0.5)/\sigma]^2}$$

with $\sigma = 0.08$. Use the explicit Euler, 2nd-order Adams-Bashforth (AB2), implicit Euler, trapezoidal, and 4th-order Runge-Kutta methods. For the explicit Euler and AB2 methods, use a Courant number, $ah/\Delta x$, of 0.1; for the other methods, use a Courant number of unity. Plot the solutions obtained at $t = 1$ compared to the exact solution (which is identical to the initial condition).

3.3 Methodology

The first step in this process was learning to obtain derivatives, of various orders and to varying orders of accuracy, using finite difference operators. I learned how to use Taylor series and kernels of different sizes to derive the finite difference coefficients and how these can be assembled into banded matrices to form derivative operators.

I also learned how to modify these operators to enforce inflow/outflow and periodic boundary conditions.

Following this, I learned about some semi-discrete approaches which attempt to exactly solve sets of coupled ODEs formed from the discrete approximations made during the spatial discretization.

Finally, I learned different methods for time-marching, including single and two step linear methods, both implicit and explicit, and predictor-corrector methods, such as the ubiquitous 4th order Runge-Kutta method.

I proceeded to write code in MATLAB to create 2nd and 4th order difference operators of arbitrary size that would compute numerical approximations of the first derivative. I then developed code that would perform explicit Euler time-marching. Following that, I implemented an arbitrary two-step linear method (explicit or implicit) and RK4.

3.4 Results

3.4.1 Problem 6.7

In Figure 7, the solutions are shown when using particular time marching methods.

Additionally, we examined additional two-step linear time-marching methods, such as the “leap frog,” “a-contractive,” “milne,” and “most accurate explicit” methods.

3.4.2 Problem 6.8

Problem 6.8 serves as a continuation of 6.7, advancing through to t = 10 with the simulated wave while using fourth-order spatial derivative approximations as well the fourth-order Runge-Kutta time-marching method. In addition, I compared the results obtained from solving the same problem using second-order spatial differential operators using the same time-marching method.

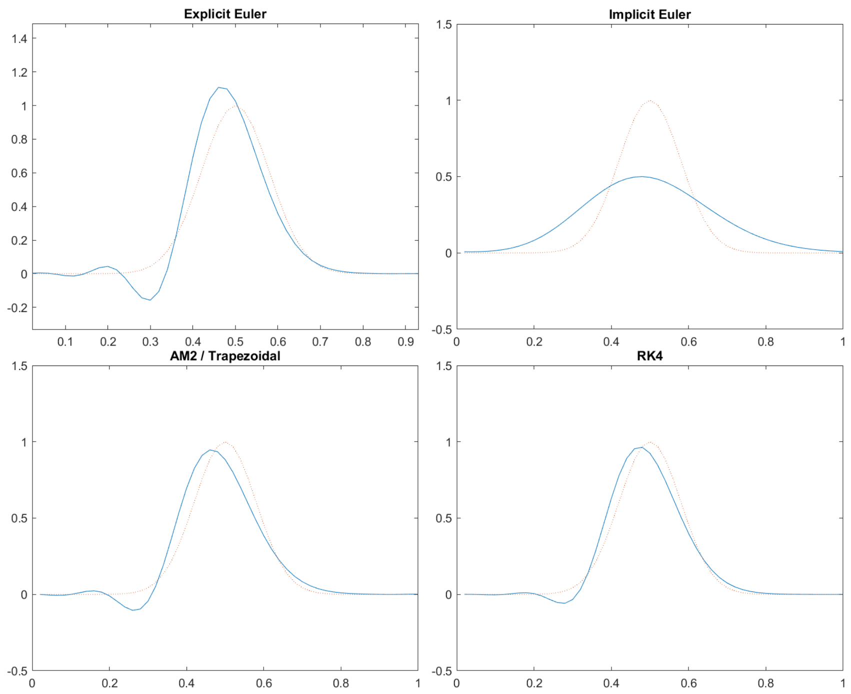
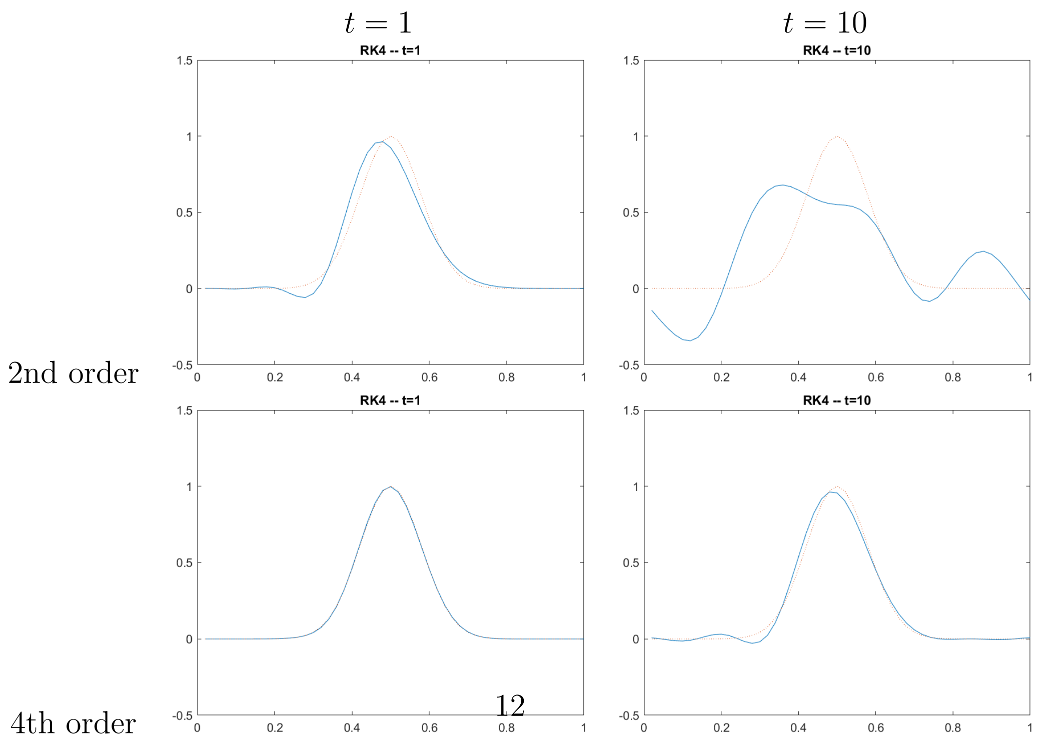


Figure 7: Part of solution to problem 6.7, showing some of the time-marching methods used



3.4.3 Problem 6.9

This problem consisted of numerically solving the convection equation at time t = 1 and measuring RMS error compared to the exact solution. Plotting this error against the number of grid points in the domain mesh shows how error scales with increasing the number of points we are using in the computations for a given order of accuracy.

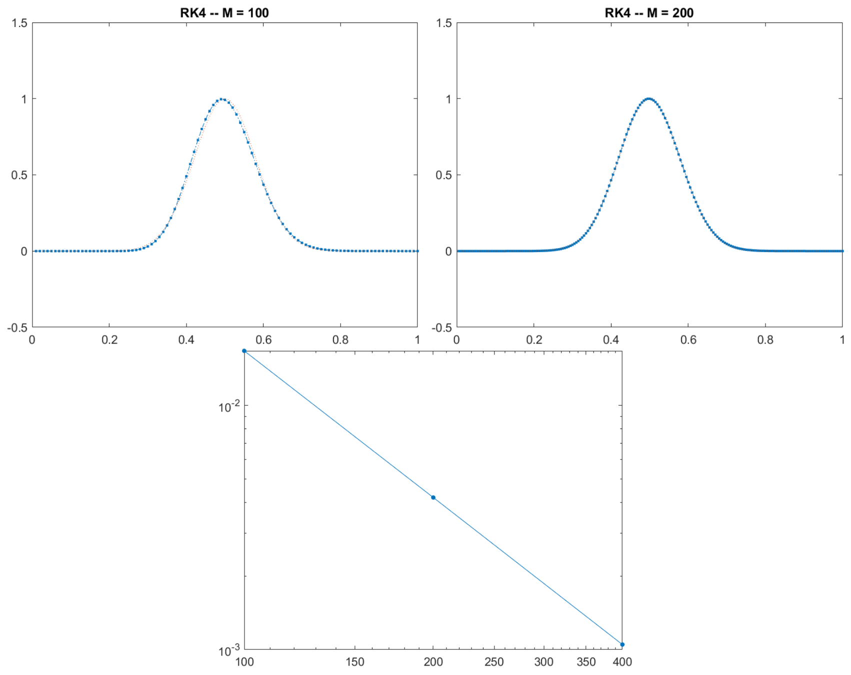


Figure 8: Linear convection equation numerically solved for time t = 1 with fixed Courant number on grids of 100 and 200 nodes. Logarithmic plot of error as a function of number of grid points shows a quadratic decrease in error vs. grid density (for seecond-order spatial differences)

3.4.4 Problem 6.10

This problem was solved like 6.9, using fourth-order spatial derivative operators rather than second-order. Error decreased as the node density to the fourth power.

3.4.5 Problem 6.11

This problem consisted of solving the linear convection equation using an inflow boundary condition rather than a periodic one. Once again we plotted the error, which consistently decreased with grid density according to the order of the method being used.

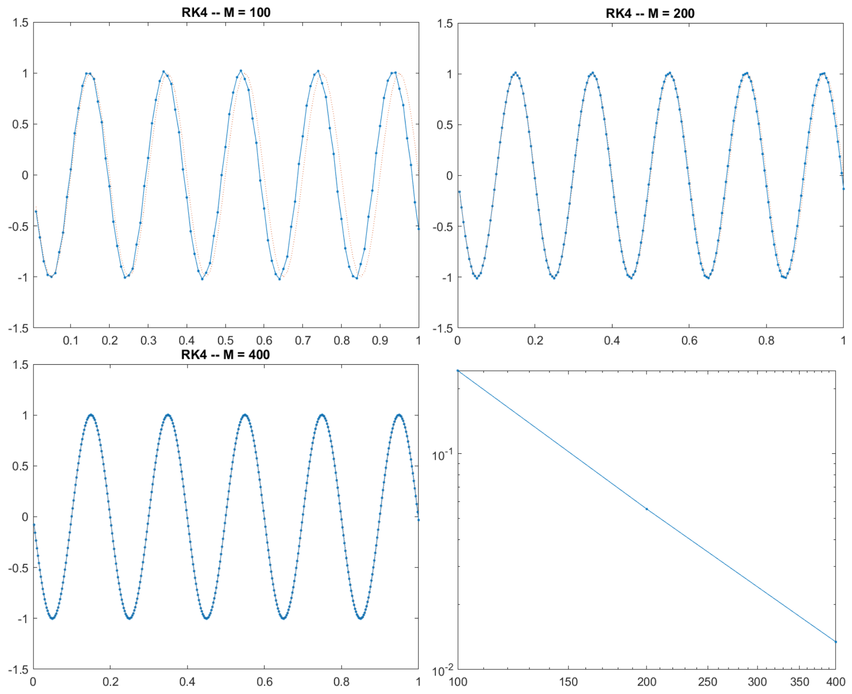


Figure 9: Linear convection equation numerically solved for time t = 1 with fixed Courant number on grids of 100 and 200 nodes.

3.4.6 Problem 6.12

This problem was solved identically to 6.11, save for the use of fourth-order spatial derivative operators instead of second-order ones, with particular considerations for treatment of the matrix edges, using forward- and backward-biased kernels.

3.5 Conclusion

Ultimately, this project was a success. The learning objectives were met and the code produced consistent results. This project provided a good introduction to numerical methods for the solution of differential equations and to scientific computing more generally.