Spring 3-29-2023

Using Physics-Informed Neural Networks for Multigrid in Time Coarse Grid Equations

Jonathan P. Gutierrez
University of New Mexico - Main Campus

Jonathan Gutierrez

Candidate

Mathematics and Statistics

Department

This thesis is approved, and it is acceptable in quality and form for publication:

Approved by the Thesis Committee:

Dr. Jehanzeb Chaudhary , Chairperson

Dr. Jacob Schroder

Dr. Stephen Lau

Using Physics-Informed Neural Networks for Multigrid in Time Coarse Grid Equations

by

Jonathan Gutierrez

B.S., Mathematics of Computation, University of New Mexico, 2020

THESIS

Submitted in Partial Fulfillment of the
Requirements for the Degree of

Master of Science
Mathematics

The University of New Mexico

Albuquerque, New Mexico

May, 2023

Dedication

To those who encouraged and never gave up on me along the way.

Acknowledgments

I would like to personally thank my advisors Dr. Jehanzeb Chaudhry and Dr. Jacob Schroder for their continual support and guidance offered throughout the entirety of my time as a student here.

Using Physics-Informed Neural Networks for Multigrid in Time Coarse Grid Equations

by

Jonathan Gutierrez

B.S., Mathematics of Computation, University of New Mexico, 2020

M.S., Mathematics, University of New Mexico, 2023

Abstract

For parallel-in-time integration methods, the multigrid-reduction-in-time (MGRIT) method has shown promising results in both improved convergence and increased computational speeds when solving evolution problems. However, one problem the MGRIT algorithm currently faces is it struggles solving hyperbolic problems efficiently. In particular, hyperbolic problems are generally solved using explicit methods and this causes issues on the coarser multigrid levels, where larger (coarser) time step sizes can violate the stability condition. In this thesis, physics-informed neural networks (PINNs) are used to evaluate the coarse grid equations in the MGRIT algorithm with the goal to improve convergence for problems with hyperbolic behavior, as well as improve stability for the time-stepping on coarser levels.

Contents

List of Figures

List of Tables

Glossary

$\Phi(\cdot)$	Fine scale time propagator.
$\Phi_\Delta(\cdot)$	Coarse scale time propagator.
$\mathbf{u}_i$	Numerical solution to PDE at the ith point of the temporal discretization.
N_t	Number of discretization intervals in the in the temporal domain.
N_x	Number of discretization intervals in the spatial domain.
δt	Time step size for fine grid.
ΔT	Time step size for coarse grid.
t_i	F-points on the fine grid on the temporal domain, where $i = 0, 1, ...$
T_i	C-points on the coarse grid on the temporal domain, where $i = 0, 1, ...$
A	Fine-level space-time operator
A_Δ^*	Ideal (or perfect) coarse-level space-time operator
A_Δ	Approximate coarse-level space-time operator used by MGRIT

Glossary

$\mathbf{e}_i$	Error at the ith point between numerical and exact solution of the PDE where $i = 0, 1, ...$
$\mathbf{r}_i$	Residual at the ith point of our numerical scheme of the PDE where $i = 0, 1, ...$
E_Δ^F	Error propagator for F-relaxation scheme.
E_Δ^{FCF}	Error propagator for FCF-relaxation scheme.
λ_ω	Eigenvalues for fine-grid propagator Φ.
μ_ω	Eigenvalues for coarse-grid propagator Φ_Δ.
$\sigma(\cdot)$	Activation function for a neural network.
$\mathbf{u}_\theta(x, t)$	PINNs approximation to solution $\mathbf{u}$ with parameters θ.
$\mathbf{u}_{\theta,i}$	Predicted values from neural network.
$C(\theta)$	Cost or loss function with respect to parameters θ.
$\Delta\theta$	A small permutation of θ.
η	Learning rate of optimization method.
$\mathbf{u}_t$	First partial derivative of $\mathbf{u}$ with respect to t. $\mathbf{u}_t := \frac{\partial \mathbf{u}}{\partial t}$.
$\mathbf{u}_{xx}$	Second partial derivative of $\mathbf{u}$ with respect to $\mathbf{x}$. $\mathbf{u}_{xx} := \frac{\partial^2 \mathbf{u}}{\partial x^2}$.
$\mathcal{N}(t, x, u, u_x, u_{xx}, ...)$	Generic nonlinear function of time, space, solution u, and its derivatives.
$\mathcal{D}$	Arbitrary spatial domain.
N_r	Number of residual collocation points for our PINNs model.

Glossary

N_0	Number of initial collocation points for our PINNs model.
N_b	Number of boundary collocation points for our PINNs model.
X^r	Our residual data set for our PINNs model.
X^0	Our initial data set for our PINNs model.
X^b	Our boundary data set for our PINNs model.
$\Phi_\theta^r(\cdot)$	Mean squared residual with parameters θ.
$\Phi_\theta^0(\cdot)$	Mean squared misfit with respect to initial conditions with parameters θ.
$\Phi_\theta^b(\cdot)$	Mean squared misfit with respect to boundary conditions with parameters θ.
$\Phi_\theta(X)$	PINNs loss function with full training data X.
a	Advection velocity coefficient.
ϵ	Diffusion coefficient.
$\mathbf{F}(\cdot)$	Forcing function of our PDE.
α	Boundary condition at $\mathbf{x} = 0$.
β	Boundary condition at $\mathbf{x} = 1$.

Chapter 1

Introduction

1.1 Overview

Considering recent changes in computer architectures, processor clock speeds have become stagnant, while performance improvements have come from the addition of ever more cores. Thus, future performance gains will be primarily coming from greater use of parallelism. In this thesis, we consider this impact for evolution equations, which are traditionally solved using time marching approaches (i.e., sequentially). As a result, parallelism has traditionally been limited to spatial parallelism. Thus, to increase parallelism, we consider adding parallelism to the temporal domain with parallel-in-time methods. Such parallel-in-time methods have been the subject of active research in recent years as these methods are fairly new with the first multigrid space-time method developed in 1984 [1].

We consider the multigrid-reduction-in-time (MGRIT) method, which is a recent time-parallel algorithm used for a number of problems in science and engineering [2, 3, 4, 5, 6, 7, 8, 9]. MGRIT improves the convergence and computational speeds of many problems, but struggles with hyperbolic problems [4] which we discuss in

section 2.4. This motivates our research on using machine learning techniques within MGRIT to mitigate these issues. In particular, we employ physics-informed neural networks (PINNs) [10, 11, 12, 13, 14, 15, 16] and explore the accuracy and stability of our new approach. Since hyperbolic PDEs tend to be solved using explicit methods, we investigate if using PINNs in our MGRIT algorithm will increase the stability of an explicit method. MGRIT creates coarse time grids with larger time-step sizes making it likely to violate a CFL (see chapter 4) condition resulting in stability issues. We do a series of tests to gauge the usefulness and applicability of using PINNs in our algorithm. In these tests, we compare MGRIT using PINNs on the coarse grid to what we refer to as naive MGRIT. Naive MGRIT refers to the standard MGRIT model using finite differencing on both fine grid and coarse grid, but with a larger step size. Naive does not imply that it is worse, it simply implies that it is the intuitive choice that may or may not be optimal.

This thesis compares naive MGRIT to MGRIT using PINNs on the coarse grid by studying the number of iterations and wall-clock time needed for MGRIT solutions to converge. To get a more in depth look at the theory behind MGRIT and its parallelism, please refer to *Parallel Time Integration with Multigrid* [2] and the *XBraid Parallel Multigrid in Time* user's manual [17].

Chapter 2

The Multigrid-Reduction-in-Time (MGRIT) Algorithm

For this thesis, we use the multigrid-reduction-in-time method (MGRIT) to achieve parallelism in time. The key idea is instead of solving an evolution problem, i.e., a time-dependent differential equation, using sequential time marching, we instead solve it iteratively such that the solution for all time values is computed simultaneously in parallel. We describe this more in detail below.

MGRIT uses a hierarchy of temporal grids from fine to coarse, the finest being the one that contains all the time points in the simulation. Each subsequent grid contains fewer time points. The coarsest grid has a trivial number of time points which means it can be solved quickly and exactly with sequential time marching. We then use the solutions from coarser grids to correct the solutions on finer grids.

The idea is that the coarsest grid is cheap to solve sequentially and the more computationally heavy fine grids will be corrected (solved) in parallel. Thus the overall performance should (in theory) be faster than solving the fine grid sequentially when enough processors are available. This is shown to the be the case in both

theory and practice [2] using XBraid [17], a parallel-in-time software package that implements MGRIT.

2.1 A Two-Level MGRIT Method

We start off by describing a two-level MGRIT method for a system of ordinary differential equations

$$\mathbf{u}'(t) = \mathbf{f}(t, \mathbf{u}(t)), \quad \mathbf{u}(0) = \overline{\mathbf{u}_0}, \quad t \in [0, T], \tag{2.1}$$

where $\mathbf{u}, \mathbf{f} \in \mathbb{R}^n$. The algorithm has two levels, a coarse level and a fine level. The discretization for the fine level is carried out using uniform time steps $t_i = i\delta t$ for $i = 0, 1, ..., N_t$ with a step size of $\delta t = T/N_t$. Let u_i be an approximate solution to $\mathbf{u}(t_i)$. A general sequential one-step method time discretization for (2.1) is given as

$$\begin{aligned} &\mathbf{u}_0 = \mathbf{u}(0) \\ &\mathbf{u}_i = \Phi_i(\mathbf{u}_{i-1}) + \mathbf{g}_i, \quad i = 1, 2, ..., N_t. \end{aligned} \tag{2.2}$$

In the linear setting, $\Phi_i(\cdot)$ corresponds to a matrix-vector product (e.g. $\Phi_i u_{i-1}$), but for this thesis we will also be focusing on the nonlinear case. This is due to PINNs yielding nonlinear Φ_i operators. Thus we will introduce MGRIT from the linear perspective for simplicity, while noting key differences needed when moving to nonlinear multigrid cycling, also called full approximation scheme (FAS) multigrid [18]. An example of Φ is given for backward Euler when discussing the coarse grid.

Applying the one-step method in (2.2) for all time-steps i corresponds to a forward solve on the block (non-)linear system

$$A(\mathbf{u}) \equiv \begin{pmatrix} I & & & \\ -\Phi_1 & I & & \\ & \ddots & \ddots & \\ & & -\Phi_{N_t} & I \end{pmatrix} \begin{pmatrix} \mathbf{u}_0 \\ \mathbf{u}_1 \\ \vdots \\ \mathbf{u}_{N_t} \end{pmatrix} = \begin{pmatrix} \mathbf{g}_0 \\ \mathbf{g}_1 \\ \vdots \\ \mathbf{g}_{N_t} \end{pmatrix} \equiv \mathbf{g} \tag{2.3}$$

where $\mathbf{g}_0 = \mathbf{u}(0)$. Without loss of generality, we assume Φ is constant (time-independent discretization) for describing MGRIT, i.e., we get the system,

$$A(\mathbf{u}) \equiv \begin{pmatrix} I & & & \\ -\Phi & I & & \\ & \ddots & \ddots & \\ & & -\Phi & I \end{pmatrix} \begin{pmatrix} \mathbf{u}_0 \\ \mathbf{u}_1 \\ \vdots \\ \mathbf{u}_{N_t} \end{pmatrix} = \begin{pmatrix} \mathbf{g}_0 \\ \mathbf{g}_1 \\ \vdots \\ \mathbf{g}_{N_t} \end{pmatrix} \equiv \mathbf{g}. \tag{2.4}$$

Solving this using forward substitution is optimal and has complexity $\mathcal{O}(N_t)$, but it is sequential. For our MGRIT approach, we introduce parallelism by solving this system iteratively using a multigrid method rather than solving it directly. Since MGRIT is based on the multigrid method, we initially look at the two-grid or two-level method. A key idea in multigrid methods is using relaxation (block Jacobi) on fine meshes together with correction terms from coarse meshes to solve the problem. Every multigrid method uses an approach to transfer values between meshes. The multilevel algorithm follows from recursive application of the two-level algorithm.

The algorithm is based the two disjoint sets of F-points and C-points, which we now define. We define a time step size for our coarser grid as $\Delta T = m\delta t$. Then the C-points will be defined as

$$T_i = i\Delta T \quad \text{for} \quad i = 0, ..., N_\Delta \quad \text{where} \quad N_\Delta = N_t/m,$$

and m is the coarsening factor. The F-points are then defined as

$$\{t_i\} \backslash \{T_j\} \quad \text{for} \quad i = 0, 1, ..., N_t \quad \text{and} \quad j = 0, 1, ..., N_\Delta.$$

A schematic is shown in figure 2.1. Thus the union of the F- and C-points form the fine grid. Going from the fine grid to the coarse grid is done by coarsening in time with a factor of m. Now that we have an understanding of our F and C-points, we consider a way to coarsen equation (2.2) so that it only involves C-points, i.e., we shrink the problem size. We can get rid of the F-points and isolate the C-points in

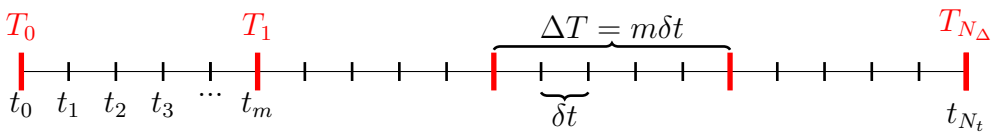


Figure 2.1: The fine and coarse grid with coarsening factor m. The C points are highlighted in red and are present on both the fine and coarse grid whereas the F are highlighted in black and are only present on the fine grid.

our one-step method by using block elimination in the time-stepping problem. For the linear case, this involves a recursive application of (2.2) as shown below.

$$\begin{aligned}
\mathbf{u}_{km} &= \Phi\mathbf{u}_{km-1} + \mathbf{g}_{km} \\
&= \Phi(\Phi\mathbf{u}_{km-2} + \mathbf{g}_{km-1}) + \mathbf{g}_{km} \\
&= \Phi^2\mathbf{u}_{km-2} + \Phi\mathbf{g}_{km-1} + \mathbf{g}_{km} \\
&= \Phi^3\mathbf{u}_{km-3} + \Phi^2\mathbf{g}_{km-2} + \Phi\mathbf{g}_{km-1} + \mathbf{g}_{km} \\
&\vdots
\end{aligned}$$

$$\mathbf{u}_{km} = \Phi^m\mathbf{u}_{km-m} + \tilde{\mathbf{g}}_{km}, \quad k = 1, 2, ..., N_\Delta \tag{2.5}$$

where

$$\tilde{\mathbf{g}}_{km} = \Phi^{m-1}\mathbf{g}_{km-m+1} + \ldots + \Phi\mathbf{g}_{km-1} + \mathbf{g}_{km}. \tag{2.6}$$

This is the so-called "ideal" coarse-grid problem at the C-points given as

$$\begin{aligned}
\mathbf{u}_0 &= \mathbf{g}_0 \\
\mathbf{u}_{km} &= \Phi^m\mathbf{u}_{km-m} + \tilde{\mathbf{g}}_{km}, \quad k = 1, 2, ..., N_\Delta,
\end{aligned} \tag{2.7}$$

which is called ideal, because its solution is identical to that of the original problem in equation (2.2).

We can also represent this ideal coarse-grid problem in matrix form as

$$A^*_\Delta \mathbf{u}_\Delta = \begin{pmatrix} I & & & \\ -\Phi^m & I & & \\ & \ddots & \ddots & \\ & & -\Phi^m & I \end{pmatrix} \begin{pmatrix} \mathbf{u}_{\Delta,0} \\ \mathbf{u}_{\Delta,1} \\ \vdots \\ \mathbf{u}_{\Delta,N_\Delta} \end{pmatrix} = R_\Phi \mathbf{g} \equiv \mathbf{g}_\Delta \tag{2.8}$$

where $\mathbf{u}_{\Delta,j} = \mathbf{u}_{jm}$ on the first coarse level and R_Φ is the rectangular restriction operator

$$R_\Phi = \begin{pmatrix} I & & & & & & & \\ & \Phi^{m-1} & \dots & \Phi & I & & & \\ & & & & & \ddots & & \\ & & & & & & \Phi^{m-1} & \dots & \Phi & I \end{pmatrix}. \tag{2.9}$$

Comparing to the fine-level matrix problem of (2.4), we can see that equation (2.8) corresponds to the solution u of (2.4) at the C-points $i = jm, j = 0, 1, ..., N_\Delta$.

Thus, the algorithm we are developing has one fine level and (possibly) several coarse levels, if the coarsening procedure above is repeated. For the notation, the subscript Δ indicates we are on the first coarse level. The corresponding non-linear problem $A^*_\Delta(\mathbf{u}_\Delta) = \mathbf{g}_\Delta$ is derived from block elimination of (2.2), but instead of multiplying Φ to the forcing terms, we instead evaluate the forcing terms internally, evaluating Φ m-times, denoted $\Phi^{(m)}$.

$$\begin{aligned} \mathbf{u}_{km} &= \Phi(\mathbf{u}_{km-1}) + \mathbf{g}_{km} \\ &= \Phi(\Phi\mathbf{u}_{km-2} + \mathbf{g}_{km-1}) + \mathbf{g}_{km} \\ &\;\;\vdots \\ &= \Phi(\dots(\Phi(\mathbf{u}_{km-m}) + \mathbf{g}_{km-m+1}) + \mathbf{g}_{km-m+2}) + \dots) + \mathbf{g}_{km-1}) + \mathbf{g}_{km} \\ &=: \Phi^{(m)}(\mathbf{u}_{km-1}). \end{aligned}$$

One thing to keep in mind is that evaluating $\Phi(\cdot)$ m-times is expensive. Also,

solving (2.7) is just as expensive as (2.2), i.e., we do not have a practical coarse-grid equation, yet. In order to make the method practical, we approximate $\Phi^{(m)}$ with a cheaper function. The naive choice would be to choose Φ_Δ, which denotes a rediscretization using a time step size of ΔT. For example, we apply backward Euler's method (see Appendix B) to a linear ODE $\dot{\mathbf{u}} = A\mathbf{u} + \mathbf{b}$, where A is a matrix and $\mathbf{u}$ and $\mathbf{b}$ are vectors, with step size δt. The resulting fine scale solution is

$$u_i = (I - \delta t A)^{-1}(u_{i-1} + \delta t b_i), \quad i = 0, ..., N_t.$$

For this discretization, $\Phi = (I - \delta t A)^{-1}$. Then, to define a cheaper coarse scale operator, we rediscritize with ΔT, $\Phi_\Delta = (I - m\delta t A)^{-1}$. For the inversion of Φ, various linear and nonlinear methods are common e.g. Newton's method, sparse LU, spatial multigrid, etc. Our new space-time coarse-grid operator can be expressed as

$$A_\Delta = \begin{pmatrix} I & & & \\ -\Phi_\Delta & I & & \\ & \ddots & \ddots & \\ & & -\Phi_\Delta & I \end{pmatrix}. \tag{2.10}$$

$A_\Delta \approx A_\Delta^*$ since $\Phi_\Delta \approx \Phi^{(m)}$, and is ideally much cheaper to compute than evaluating Φ m-times. Later in chapter 3, we use a trained neural network for the coarse-grid equations.

Before we introduce the two-level algorithm, we must first introduce relaxation to compliment coarse grid correction which is typical for multigrid methods. For now, we look at two relaxation schemes, F-relaxation and C-relaxation. F-relaxation is given by

$$\begin{aligned} u_{km+1} &= \Phi(u_{km}) + g_{km+1}, && k = 0, 1, ..., N_\Delta - 1 \\ u_{km+2} &= \Phi(u_{km+1}) + g_{km+2}, && k = 0, 1, ..., N_\Delta - 1 \\ &\vdots \\ u_{km+m-1} &= \Phi(u_{km+m-2}) + g_{km+m-1}, && k = 0, 1, ..., N_\Delta - 1. \end{aligned} \tag{2.11}$$

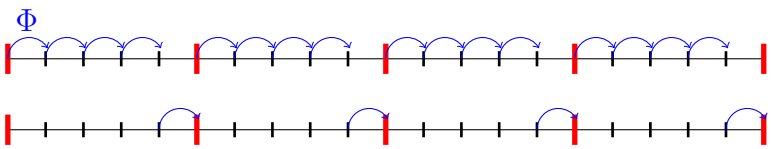


Figure 2.2: The top schematic illustrates F-relaxation with a coarsening factor of 5 and the bottom illustrates C-relaxation. F-relaxation updates the fine points with initial condition given on the previous C-points, and C-relaxation updates the coarse grid with initial conditions given on the previous F-points.

And C-relaxation is given by

$$u_{km} = \Phi(u_{km-1}) + g_{km}, \quad k = 1, 2, ..., N_\Delta. \tag{2.12}$$

This is equivalent to apply block Jacobi on the F- and C-points respectively. F-relaxation is relaxation only on the F-points. Since time-stepping methods require information from previous points, F-relaxation only requires information from the C-points to begin. In this method, the C-points are not altered at all. F-relaxation updates the F-points on the sub-intervals (T_i, T_{i+1}). In C-relaxation, analogous to F-relaxation, the C-points are being updated. The first time point, C_0, is given as an initial condition, whereas the rest of the C-points need to be updated by time-stepping the left-adjacent time point. F- and C-relaxation can be computed in parallel for each k. This means that if we were to magically have the exact solution at all C-points, MGRIT would solve the problem in a single iteration!

In practice, one often uses so-called FCF-relaxation scheme [2], which we also use in our numerical experiments. FCF-relaxation is simply an F-relaxation followed by a C-relaxation, and then another F-relaxation. Though FCF-relaxation is more expensive, it typically offers faster convergence as the C-points are being corrected even before the residual correction [3].

The last algorithmic component to discuss is transferring values between grids using restriction and prolongation (interpolation) operators. Following [19], we define

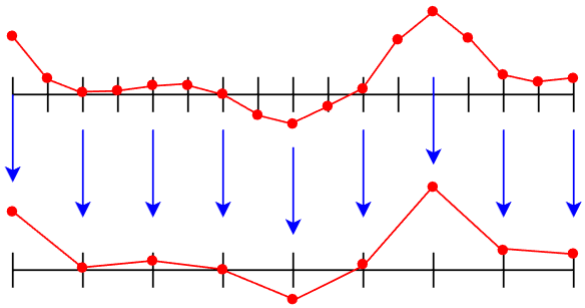


Figure 2.3: Restriction on the time grid via injection.

a restriction operator using simple injection $R_\Delta \mathbf{v} = \mathbf{v}_\Delta$ following the rule

$$v_{\Delta,j} = v_{mj}, \quad j = 0, ..., N_\Delta. \tag{2.13}$$

In other words, injection is simply taking the values directly from the corresponding fine-grid points and placing them in the coarse points. Thus we get our restriction operator R_Δ,

$$R_\Delta = \begin{bmatrix} I & 0 & \dots & 0 & & & & & \\ & & & & I & 0 & \dots & 0 & \\ & & & & & & & & \ddots \end{bmatrix}.$$

The spacing between each I is $m-1$ block rows. This operator injects the values at the C-points from the fine grid to the coarse grid. A schematic is shown in figure 2.3. After an F-relaxation sweep, the residual at all the F-points is 0, thus it makes sense to use injection for our restriction operator [17]. For this reason, Xbraid always does an F-relaxation sweep before the application of R_Δ which is equivalent to using ideal restriction, R_Φ.

Injection for coarse grid to fine grid is simply the reverse of figure 2.3. Similar to R_Φ, we can define P_Φ which injects the coarse grid to the C-points on the fine grid,

followed by an F-relaxation sweep, that is,

$$P_\Phi = \begin{bmatrix} I & & \\ \Phi & & \\ \Phi^2 & & \\ \vdots & & \\ \Phi^{m-1} & & \\ & I & \\ & \Phi & \\ & \Phi^2 & \\ & \vdots & \\ & \Phi^{m-1} & \\ & & \ddots \end{bmatrix}. \tag{2.14}$$

In the MGRIT algorithm, we make use of the fact that ideal restriction, R_Φ, is equivalent to one F-relaxation followed by simple injection (fine to coarse), and that ideal interpolation is equivalent to simple injection (coarse to fine) followed by F-relaxation [2, 3]. This simplifies the algorithm as F-relaxation and simple injection are relatively easy to program kernels.

Two-level MGRIT

The two-level method first applies an F(CF)-relaxation sweep, then applies a coarse grid correction (described below), finally it fills in the values at the F-points with another F-relaxation. The two-level algorithm gets an approximate solution to the coarse system (2.8) using A_Δ (2.10), then computes the solution at the fine points with (2.11) and (2.12) in parallel on each subinterval $(t_{im}, t_{im+m-1}), i = 1, ..., N_\Delta$. The coarse grid correction is computed using the residual correction scheme (FAS multigrid) which we describe next.

For a nonlinear system $A(\mathbf{u}) = \mathbf{f}$, we start off with an initial guess $\mathbf{v} \approx \mathbf{u}$. The residual is given as $\mathbf{r} = \mathbf{f} - A(\mathbf{v})$. If we replace $\mathbf{f}$ with $A(\mathbf{u})$ we get the residual equation

$$A(\mathbf{u}) - A(\mathbf{v}) = \mathbf{r}. \tag{2.15}$$

We can modify this equation using the error $\mathbf{e} = \mathbf{u} - \mathbf{v}$ by setting $A(\mathbf{u}) = A(\mathbf{v} + \mathbf{e})$. This leads us to the modified residual equation

$$A(\mathbf{v} + \mathbf{e}) - A(\mathbf{v}) = \mathbf{r}. \tag{2.16}$$

We then allow the $A(\mathbf{v})$ term to "lag", and be A applied to the previous $\mathbf{v}$ value, and place it on the right-hand-side, yielding,

$$A(\mathbf{v} + \mathbf{e}) = \mathbf{r} + A(\mathbf{v}), \tag{2.17}$$

which can be solved for $\mathbf{v}+\mathbf{e}$. Finally, we compute an approximate error, $(\mathbf{v}+\mathbf{e}) - \mathbf{v}$ using the lagged $\mathbf{v}$ term. MGRIT uses this process (note, we start off with an initial guess $\mathbf{u}$ then update the values. For MGRIT, $\mathbf{u}_\Delta$, the course-grid solution plays the role of $\mathbf{v}$. The coarse-grid equation solved by MGRIT,

$$A_\Delta(\mathbf{u}_\Delta + \mathbf{e}_\Delta) = \mathbf{r}_\Delta + A_\Delta(\mathbf{u}_\Delta), \tag{2.18}$$

$$u_{\Delta,j} = u_{mj}, \quad e_{\Delta,j} = e_{mj}, \quad r_{\Delta,j} = r_{mj}, \quad j = 0, ..., N_\Delta.$$

After solving the coarse-grid, we compute the coarse-grid approximate error

$$\mathbf{e}_\Delta \approx (\mathbf{u}_\Delta + \mathbf{e}_\Delta) - \mathbf{u}_\Delta$$

and interpolate this error to the fine-grid using injection at the C-points

$$u_{mi} = u_{mi} + e_{\Delta,i} \quad \text{for } i = 0, ..., N_\Delta$$

followed by a fine-grid F-relaxation to update the F-points. This completes the two-level coarse-grid correction process.

This gives us a better approximation to our problem. We repeat this process, but this time using the updated approximation as our initial guess. We do this until we reach a residual norm less than a given tolerance. With this being said, we present the FAS algorithm in the next section.

2.2 MGRIT Algorithm

We now describe the multilevel MGRIT algorithm. The multilevel algorithm is a recursive application of the 2-level algorithm with a hierarchy of time discritization meshes $\Omega_{\Delta l}, l = 0, 1, ..., L = \log_m(N_t)$, with $l = 0$ being the finest level and $l = L$ being the coarsest level. Note that we stop coarsening if the number of points in the discretization is less than the coarsening factor m. Each level uses constant spacing δt, for level 0, $m\delta t$ for level 1, $m^2\delta t$ for level 2, etc., where $m > 0$ is the temporal coarsening factor. We denote the multigrid level as $A_{\Delta l}(\mathbf{u}_{\Delta l}) = \mathbf{g}_{\Delta l}$ for levels $l = 0, 1, ..., L$. $A_{\Delta l}$ is the time discretization on $\Omega_{\Delta l}$ using the time-propagator matrix $\Phi_{\Delta l}$ for each level.

Algorithm 1: Multigrid Reduction in Time using a V-cycle

MGRIT$(A, \mathbf{u}, \mathbf{g}, m, l)$
if l is the coarsest level, L **then**
    Solve coarse-grid system $A(\mathbf{u}) = \mathbf{g}$ (sequentially)
    return u
else
    Relax on $A(\mathbf{u}) = \mathbf{g}$ using FCF-relaxation.
    Restrict fine grid approximation and its residual to the coarse grid
    $u_{\Delta,i} \leftarrow u_{mi}, \quad r_{\Delta,i} \leftarrow g_{mi} - A(\mathbf{u})_{mi} \quad \text{for } i = 0, ..., N_{\Delta l}.$
    Define the right hand side for coarse grid problem: $\mathbf{g}_\Delta = A_\Delta(\mathbf{u}_\Delta) + \mathbf{r}_\Delta$

Solve on next level: $\mathbf{u}_\Delta \leftarrow \mathbf{MGRIT}(A_\Delta, \mathbf{u}_\Delta, \mathbf{g}_\Delta, m, l+1)$.

Compute the coarse grid error approximation: $\mathbf{e}_\Delta = \mathbf{v}_\Delta - \mathbf{u}_\Delta$

Add the error approximation at the C-points:

$u_{mi} = u_{mi} + e_{\Delta,i} \quad \text{for } i = 0, ..., N_{\Delta l}.$

Relax on $A(\mathbf{u}) = \mathbf{g}$ using F-relaxation.

end if

return u

Remark 1. When using F-relaxation, MGRIT computes the exact (assuming exact arithmetic) solution on all of the F-points in the interval (T_0, T_1) in one iteration. FCF-relaxation computes the exact solution for all the F-points on the intervals (T_0, T_1) and (T_1, T_2) and also computes the exact C-point at T_1. Each iteration follows this trend resulting in an exact solution in N_t/m or $N_t/(2m)$ iterations using F-relaxation or FCF-relaxation respectively. In practice, we end the algorithm when it reaches an error tolerance as the exactness principle (given above) is too expensive to be practical.

2.3 MGRIT using XBraid Software

We carry out our experiments using the XBraid Package. XBraid uses the MGRIT algorithm in a straight-forward manner. For our relaxation method, we use FCF-relaxation for our later experiments, but for now, we do a basic example as a demonstration. A demonstration goes as follows. Let's suppose we have a simple initial value problem,

$$\begin{cases} u'(t) = \lambda u(t) + (1-\lambda)\cos(t) - (1+\lambda)\sin(t), \quad t \in [0, 5], \\ u(0) = 1. \end{cases} \tag{2.19}$$

This is an example of a stiff ODE [20]. The larger the magnitude of λ, the more stiff it becomes. This ODE has an analytical solution of

$$u(t) = \sin(t) + \cos(t). \tag{2.20}$$

We solve this ODE using the MGRIT algorithm described above. In this particular case, we use the 2-level algorithm with a coarsening factor of $m = 2$. For our relaxation scheme, we use F-relaxation with backward Euler's method on both levels. Let's set $\lambda = -50$ to make the problem stiff.

Figures 2.5 and 2.6 show that it only takes 2 cycles to solve this initial value problem to a reasonable tolerance. The goal of MGRIT is to converge to the serial solution within tolerance. We see the orange MGRIT approximation gets closer and closer to the blue serial solution until they are about the same. It is worth noting that while it takes very few iterations to solve, each cycle is relatively expensive. For this reason, running MGRIT in serial is typically slower than a typical numerical solution compute sequential in time. But if we have access to multiple processors, then MGRIT will eventually outpace serial solvers by introducing enough parallelism [2].

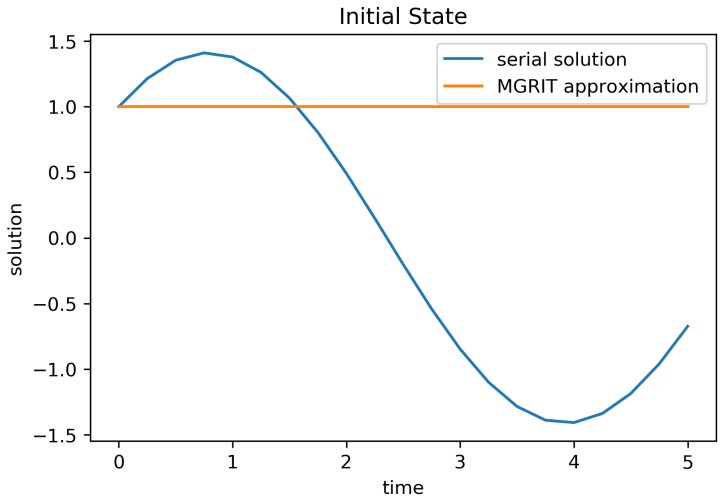


Figure 2.4: Initial state for our MGRIT algorithm.

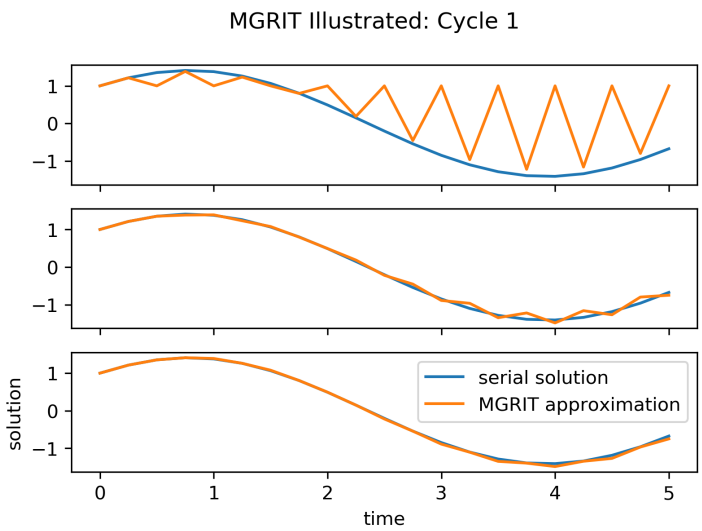


Figure 2.5: 1st cycle in our MGRIT algorithm. Top plot illustrates the first F-relaxation. The middle plot illustrates solution being corrected from the coarse-grid correction. The bottom plot illustrates the second F-relaxation.

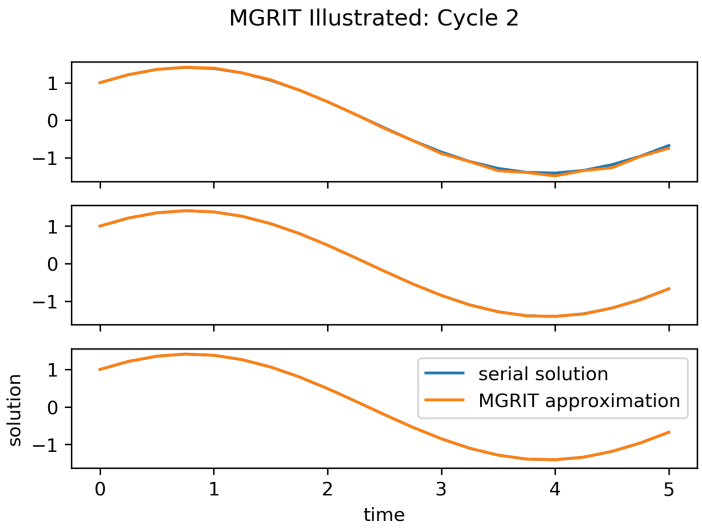


Figure 2.6: 2nd cycle in our MGRIT algorithm. Top plot illustrates the first F-relaxation. The middle plot illustrates solution being corrected from the coarse-grid correction. The bottom plot illustrates the second F-relaxation.

2.4 Convergence Theorem

We can also understand MGRIT performance through established convergence theory. Let v_i be an approximate solution of (2.2) and u_i is given as the exact solution at time point i. Then the error is defined as $e_i = u_i - v_i$. The residual then satisfies

$$r_0 = g_0 - v_0$$
$$r_i = g_i - v_i + \Phi v_{i-1}, \quad i = 1, 2, ..., N_t.$$

We next review some key results from [3, 2]. The two-level MGRIT error propagators (iteration matrices) in the linear setting are given by

$$E_\Delta^F = I - A_\Delta^{-1} A_\Delta^*, \qquad \text{for F-relaxation,}$$
$$E_\Delta^{FCF} = (I - A_\Delta^{-1} A_\Delta^*)(I - A_\Delta^*), \quad \text{for FCF-relaxation.}$$

E_Δ^F is just a residual correction at the C-points whereas E_Δ^{FCF} adds an extra relaxation step with $(I - A_\Delta^*)$. Next, we list Theorem 2.3 from *Two-Level Convergence*

Theory for Parallel Time Integration with Multigrid [3].

THEOREM. Let Φ and Φ_Δ be simultaneously diagonalizable by a unitary transformation X, with eigenvalues λ_ω and μ_ω respectively. Furthermore, assume that both time-stepping operators are stable, i.e. $|\lambda_\omega| < 1$ and $|\mu_\omega| < 1$. Then the global space-time error vector at the C-points $\bar{e} = [e_0^T, e_m^T, e_{2m}^T, ..., e_{N_T m}^T]$ satisfies

$$\|E_\Delta^F \bar{e}\|_2 \leq \max_\omega \left\{ |\lambda_\omega^m - \mu_\omega| \frac{1 - |\mu_\omega|^{N_T}}{1 - |\mu_\omega|} \right\} \|\bar{e}\|_2, \quad \text{for F-relaxation,}$$

and

$$\|E_\Delta^{FCF} \bar{e}\|_2 \leq \max_\omega \left\{ |\lambda_\omega^m - \mu_\omega| \frac{1 - |\mu_\omega|^{N_T - 1}}{1 - |\mu_\omega|} |\lambda_\omega|^m \right\} \|\bar{e}\|_2, \quad \text{for FCF-relaxation.}$$

From this theorem, we can see that when the eigenvalues, μ_ω, of Φ_Δ are close to m applications of Φ, i.e. λ_ω^m, then the quantity $|\lambda_\omega^m - \mu_\omega|$ is small, thus making the error bound small. Conversely, if $|\mu_\omega|$ is close to 1, then the quantity $\frac{1}{1-|\mu_\omega|}$ is large, which can (but not necessarily) make the error bound large. In the FCF-relaxation scheme, we have an extra factor of $|\lambda_\omega|^m$, where λ_ω is the eigenvalue of Φ. Thus any error for eigenvalues of Φ that are "far away from 1" will be quickly damped by FCF-relaxation.

Putting this together, we get good MGRIT convergence when the eigenvalues of the coarse-grid propagator(s) are close to the eigenvalues of the fine-grid propagator. And secondly, the magnitude of the eigenvalues of coarse and fine time-propagators should be far away from 1 for a smaller bound. One reason why hyperbolic problems tend to struggle is that they typically have almost purely imaginary eigenvalues which tend to have a the magnitude very close to 1 (resulting in a large convergence bound in the above Theorem). Another factor to consider, is that hyperbolic problems are generally solved using explicit methods, and explicit time-stepping schemes tend to become unstable (especially for multilevel coarse grids).

Finally, the motivation for our use of PINNs is to use PINNs to explicitly target a

stable coarse time-propagator that approximates well the eigenvalues of the fine-grid propagator, and thus leads to fast MGRIT convergence. By matching the fine scale eigenvalues λ_ω^m with our coarse grid eigenvalues μ_ω, we can expect/hope for good performance. This is particularly true for explicit methods, where typical redistritizations of Φ use larger time-step sizes are not possible due to stability issues. Thus in such cases, the coarse grid Φ_Δ must come from a more "creative" discretizations, i.e. PINNs.

Chapter 3

Neural Networks and Physics-Informed Neural Networks

3.1 Deep Neural Networks

Before we discuss Physics-Informed Neural Networks(PINNs), we must briefly introduce deep neural networks (DNNs). Neural networks are a subset of machine learning techniques which are a subset of artificial intelligence. Artificial intelligence or A.I. is a semblance of intelligence in machines induced in an artificial way (e.g. programmed). A simple A.I. can be a program running various "if else" statements all being explicitly defined. On the other hand, machine learning is a more implicit artificial intelligence, that is, we do not explicitly program the intelligence. Neural networks are machine learning techniques which aim to learn implicitly by mimicking the biology of the brain with neurons and synapses passing information to each other by "exciting" one another with an electric pulse. In a neural network, we have neurons which "fire" or get "excited" through the use of an activation function. It should be noted, however, that most activation functions do not mimic human

biology, but conceptually are inspired it.

Every feed-forward neural networks has a basic structure which consists of an input layer, hidden layer(s), and an output layer. In each layer, every neuron from one layer is connected to every neuron in the subsequent layer.[1] Each individual neuron contains a scalar value, and every connection to other neurons will have a weight and bias given to it. The value of the neurons in the next layer will be determined by the sum of the products of each neuron and their respective weights. We do this from the first (input) layer and move to the final (output) layer. This process is known as forward propagation and by itself would only be a linear transformation which is not enough to approximate any function i.e. a universal function approximator [21]. For this reason, we introduce an activation function to add non-linearity in the network. We apply the activation function after each linear transformation in every layer which we discuss in the next section. Finally, for the sake of terminology, a deep neural network is simply a neural network with multiple hidden layers. DNNs generally involve some use of forward propagation, an activation function, a loss function, backward propagation, and an optimization process to achieve the desired result. Each of these will be briefly covered in the subsequent sections. A visual of a simple DNN shown in the figure 3.1.

3.1.1 Activation Function

Activation functions help determine the output of a neural network by introducing non-linearity to the network as mentioned above. A common activation function is the sigmoid function which mimics the firing or non-firing of neurons. An additional benefit of the sigmoid is that it is differentiable, which we will need for the back-

[1]It should be mentioned that some networks are designed in a way where the neurons from one layer only connect to specific neurons in the next layer, but this delves outside the scope of this thesis.

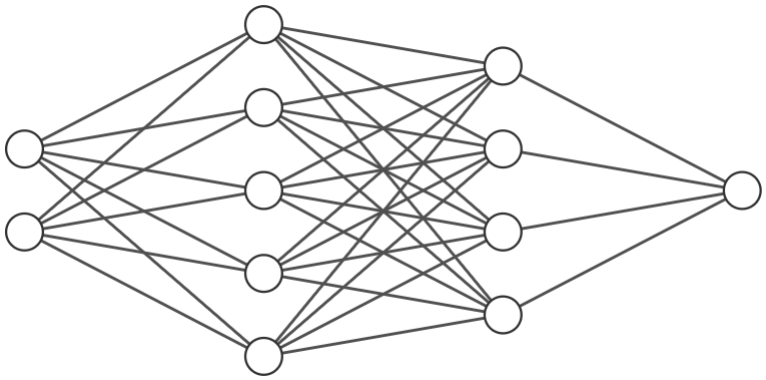


Figure 3.1: A simple illustration of a DNN with two hidden layers. The input layer has 2 neurons. The first hidden layer has 5 neurons. The second hidden layer has 4 neurons. The output layer has 1 neuron.

propagation training algorithm described in the next section. The sigmoid function is defined as

$$[\sigma(\mathbf{x})]_i = \frac{1}{1+e^{-\mathbf{x}_i}}$$

which is evaluated element wise if $\mathbf{x} \in \mathbb{R}^n$. The sigmoid function has a range of (0,1) which, by construction, should simulate the state of being on, off, or somewhere in between (a weak electric signal). Since we often work with big data in machine learning, our inputs will generally be vectors, but for illustration purposes, the figures will adjusted using only scalar values. If $\sigma : \mathbb{R}^n \to \mathbb{R}^n$, then we can scale and shift the sigmoid function using the weights (which are stored as a matrix), $W \in \mathbb{R}^{n\times n}$, and biases (which are stored as vectors), $\mathbf{b} \in \mathbb{R}^n$. We then scale and shift the activation function $\sigma(\mathbf{x})$ by applying a linear transformation to our input $\sigma(W\mathbf{x}+\mathbf{b})$. A simple illustration is shown in figure 3.2.

In general, if we have L layers, then our neural network is a composition of linear and nonlinear functions

$$u_\theta(\mathbf{x}) := W^{[L]}\sigma^{[L]}(W^{[L-1]}\sigma^{[L-1]}(\dots\sigma^{[1]}(W^{[0]}\mathbf{x}+\mathbf{b}^{[0]}))+\mathbf{b}^{[L-1]})+\mathbf{b}^{[L]}.$$

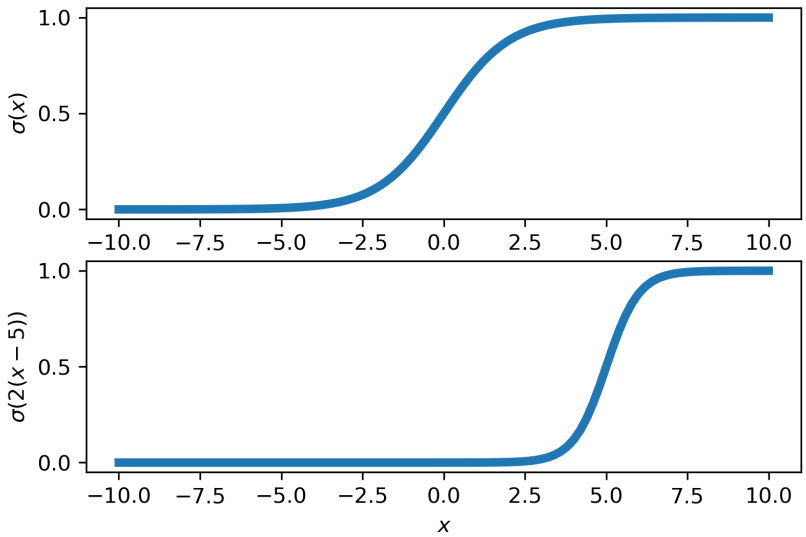


Figure 3.2: A 1D depiction of sigmoid function. The top figure is unscaled and unbiased. The bottom is weighted by $W = 2$ and biased by $b = -5$. I.e. $\sigma(2(x-5))$. The weights affect the steepness of the plot, while the bias shifts the plot.

Given a set of inputs, our network will give us an output. In other words, a neural network is just a mathematical function. Adjusting the weights and biases will scale and shift the output. The process of adjusting the weights and biases to achieve a desired output from any input data is known as training the neural network. Training is accomplished by minimizing what is known as the loss function, which measues how much the network's output differs from the expected output. In this sense, training a neural network can be thought of as an optimization problem.

3.1.2 Loss Function

In supervised learning, we have labeled training data to "teach" our network, where each training item contains input data and associated labeled correct output. Here, we use an objective function called the loss function to guide the network towards correct predictions on the training data. In optimization theory, an objective function

is simply the function we are trying to minimize. The loss function is the error between the prediction our network makes and the given labeled correct output. There are many ways to define a loss function, but a common one to use is the mean squared error. The mean squared error is defined as,

$$\text{MSE} = \frac{1}{N}\sum_{i=1}^{N}(\mathbf{y}_i - \mathbf{u}_{\theta,i})^2,$$

where $\mathbf{y}_i$ are the labeled values and $\mathbf{u}_{\theta,i}$ are the predicted values from the network, and N is the number of data points we are using. The predicted values, $\mathbf{u}_{\theta,i}$ come from our network and must take into account all of the weights and biases (parameters) in the network (denoted with subscript θ). For this reason, the loss function is also known as the cost function. For our neural network, we define our input to be $\mathbf{a}^{[1]} = \mathbf{x} \in \mathbb{R}^{N_1}$ for some size N_1. The superscript [1] indicates that we are on the 1st layer. We define propagations through subsequent layers as

$$\mathbf{a}^{[l]} = \sigma(W^{[l]}\mathbf{a}^{[l-1]} + \mathbf{b}^{[l]}), \quad l = 1, 2, ..., L.$$

Given data points $\{x^{\{i\}}\}_{i=1}^{N}$, we can define the cost function of our network to be

$$\text{Cost}(W^{[L]}, ..., W^{[1]}, \mathbf{b}^{[L]}, ..., \mathbf{b}^{[1]}) = \frac{1}{N}\sum_{i=1}^{N}\frac{1}{2}\left\|\mathbf{y}(x^{\{i\}}) - \mathbf{a}^{[L]}(x^{\{i\}})\right\|_2^2. \tag{3.1}$$

The cost is measured by averaging the squared Euclidean norm over the data points. We use the Euclidean norm as our data is given in terms of vectors. The $\mathbf{y}(x^{\{i\}})$ represents our expected labeled values at the given data point. The $\frac{1}{2}$ term is solely for convenience when taking the derivative of the cost function as we will go over shortly. Since rescaling the cost function doesn't effect the minimization process, the $\frac{1}{2}$ term is okay to add [22].

Finally, we need to adjust our parameters so that our loss function gets minimized (ideally the error goes to 0). We discuss this process in the next section.

3.1.3 Backpropagation and Optimization Methods

The backpropagation algorithm and optimization methods are often confused with one another, but they are two separate methods that simultaneously work together to minimize the loss function.

Before we describe the algorithms, let's first go back to basic calculus. How do we minimize a smooth multi-variable function? For simplicity, if we assume that all the parameters (weights and biases for all layers) are represented as a single vector θ, and we want to add a small permutation $\Delta\theta$ to it, what direction should we move in in order to reduce our function? Assuming our cost function $\text{Cost} : \mathbb{R}^s \to \mathbb{R}$ is differentiable, then a Taylor expansion is given as

$$\text{Cost}(\theta + \Delta\theta) = \text{Cost}(\theta) + \nabla\text{Cost}(\theta)^T \Delta\theta + \mathcal{O}(\Delta\theta^2) \tag{3.2}$$

where the gradient of the cost function is given by

$$\nabla\text{Cost}(\theta) = \sum_{r=1}^{s} \frac{\partial\text{Cost}(\theta)}{\partial\theta_r}.$$

The loss function decreases fastest in the direction of $-\nabla\text{Cost}(\theta)$ which we show next. Suppose we have two vectors $\mathbf{f}, \mathbf{g}$, then $|\mathbf{f}^T\mathbf{g}| \leq \|\mathbf{f}\|_2 \|\mathbf{g}\|_2$ via Cauchy-Schwarz inequality. Thus, the most negative $\mathbf{f}^T\mathbf{g}$ can be is $-\|\mathbf{f}\|_2 \|\mathbf{g}\|_2$ which occurs when $\mathbf{f} = -\mathbf{g}$ [22]. With this being said, we choose $\Delta\theta$ to be in the direction of $-\nabla\text{Cost}(\theta)$. Thus if we want to update our parameters in the correct direction, we use the following result,

$$\theta \leftarrow \theta - \eta\nabla\text{Cost}(\theta) \tag{3.3}$$

where $\eta > 0$ is the size of the step we take which is called the learning rate. Equation (3.3) is known as the gradient descent method which is an optimization method. It is common to use gradient methods to iteratively train a neural network.

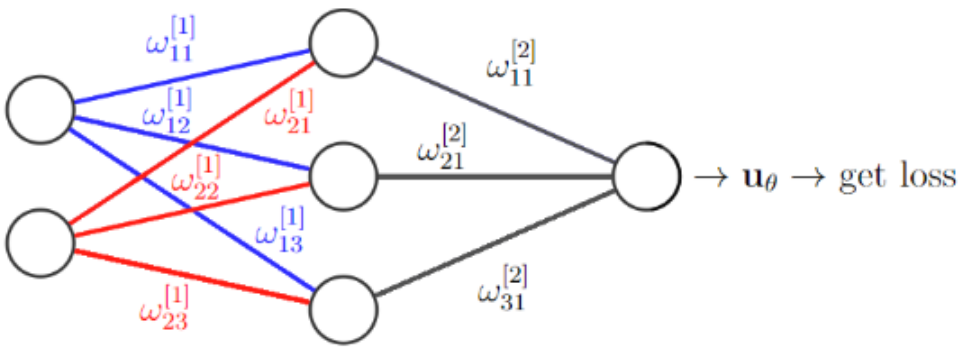


Figure 3.3: A simple neural network labeling the weights. The brackets indicate what layer of the network we are on. The indices indicate which weight came from which neuron. So $w_{21}^{[2]}$ indicates the first weight of the 2nd neuron in the 2nd layer. The final output gives us our predicted value $\mathbf{u}_\theta$ which is used to calculate the loss.

The gradient gives us the direction in which we should adjust our parameters, and we update our parameters via an optimization method. But this raises the question, how do we compute the gradients of the cost function with respect to each weight and bias? This is where the backpropagation algorithm comes into play.

Backpropagation is an algorithm that efficiently computes the gradients of the cost function with respect to the weights and biases. Backpropagation, together with an optimization method, will adjust the weights and biases so to minimize the cost function. Let's discuss how backpropagation works. We start out at the output layer and work "backwards" toward the input layer, hence it's name. If we want to compute the gradient of the cost function with respect to a weight in the hidden layer, since each gradient is dependant upon the weights that come behind them, we can compute these derivatives using the chain rule. Suppose we have a simple neural network given in figure 3.3.

Now suppose we wanted to update the weight $w_{11}^{[2]}$. Using the theory above, we

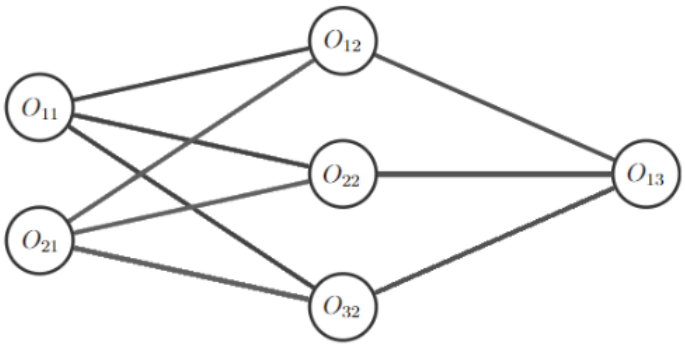


Figure 3.4: Each output is labeled O_{ij} where the i represents the ith neuron, and j represents the jth layer.

update via the optimizer

$$w^{[2]}_{11,new} = w^{[2]}_{11,old} - \eta \frac{\partial \text{Cost}}{\partial w^{[2]}_{11}}$$

Now we need to compute $\frac{\partial \text{Cost}}{\partial w^{[2]}_{11}}$. Let's label the outputs of each neuron after applying the activation function. This can be seen in figure 3.4.

Since the weight $w^{[2]}_{11}$ only affects the final output O_{13}, then we compute the gradient as

$$\frac{\partial \text{Cost}}{\partial w^{[2]}_{11}} = \frac{\partial \text{Cost}}{\partial O_{13}} \frac{\partial O_{13}}{\partial w^{[2]}_{11}}$$

If we were to compute $\frac{\partial \text{Cost}}{\partial w^{[1]}_{11}}$, we have to look at what the weights affect. The weight $w^{[1]}_{11}$ affects the outputs O_{12} and O_{13} respectively. So we compute the gradient as

$$\frac{\partial \text{Cost}}{\partial w^{[1]}_{11}} = \frac{\partial \text{Cost}}{\partial O_{13}} \frac{\partial O_{13}}{\partial O_{12}} \frac{\partial O_{12}}{w^{[1]}_{11}}$$

We compute the gradients in this manner as it is much more efficient than manually trying to calculate each gradient over and over again. This is because we already have all of the information for each output O_{ij} from our forward pass.

From (3.1), let us define the cost with respect to a single training data item i

$$C_{x^{\{i\}}} = \frac{1}{2} \left\| \mathbf{y}(x^{\{i\}}) - \mathbf{a}^{[L]}(x^{\{i\}}) \right\|_2^2 . \tag{3.4}$$

We can use this to rewrite the gradient of our cost function as

$$\nabla \text{Cost}(\theta) = \frac{1}{N} \sum_{i=1}^{N} \nabla C_{x^{\{i\}}}(\theta) \tag{3.5}$$

Thus, to compute the complete gradient of the cost function, we must compute the gradient at every point in our data set. It is important to note, computing the complete gradient is expensive, especially for larger data sets. Thus we often use the stochastic gradient descent (SGD) method for optimizing neural networks with larger data sets. The SGD method is simple to compute. The idea is that instead of computing the mean (i.e., the complete gradient), we instead take a single training data item at random hoping it approximates the mean. A single step in the algorithm does the following:

1. Pick a uniformly, random integer i from $\{1, 2, ..., N\}$.

2. Update

$$\theta \leftarrow \theta - \eta \nabla C_{x^{\{i\}}}(\theta).$$

As long as our learning rate is not too large, our algorithm should move in the direction of a local minima (hopefully, but not guaranteed to, converge to). Adding to this, the more iterations we take, the method sees more training points which gives a better picture of the full data set.

There are a couple of ways to run this method, but two important ways are sampling *with replacement* and sampling *without replacement*. If we run with replacement, that means that the index i will be put back into the training set, and is just as likely to be chosen again as any other training point. Without replacement means once we've chosen our index i, it can no longer be chosen again, until all points in the training set have been used by SGD. Running this method N times

wihtout replacement results in every training point being visited and is hence called an epoch.

Instead of using a single data point to approximate the gradient, it is common to extend SGD by using a small average over a few data items.

1. Choose m integers $k_1, k_2, ..., k_m$, randomly, uniformly from $\{1, 2, ..., N\}$.

2. Update

$$\theta \leftarrow \theta - \eta \frac{1}{m} \sum_{i=1}^{m} \nabla C_{x^{\{k_i\}}}(\theta).$$

This is the SGD method using minibatches of size m, which tends to give more accurate results than using basic SGD. The set $\{x^{\{k_i\}}\}_{i=1}^{m}$ is the minibatch. There are other techniques such as momentum and acceleration which help improve training networks, but we will not go into detail about them. Information on the Adam optimization method, which also uses types of momentum, can be found here [23].

3.2 Physics-Informed Neural Networks

Physics-informed neural networks (PINNs) have become a popular method to solve partial differential equations recently. Typical learning-based methods approximate a solution by a purely data-driven approach, whereas PINNs takes the underlying physics of the PDE into account. The physics refer to the differential equation including initial and boundary data, as well as any other parameter data (advection velocity for example). For the PINNs method, we follow the works [14, 15].

We want to find the solution $\mathbf{u} : [0, T] \times \mathcal{D} \to \mathbb{R}$ to the nonlinear partial differential

equation of the general form

$$\begin{cases} \mathbf{u}_t + \mathcal{N}(t, \mathbf{x}, \mathbf{u}, \mathbf{u}_x, \mathbf{u}_{xx}, ...) = 0, & (t, \mathbf{x}) \in (0, T] \times \mathcal{D}, \\ \mathbf{u}(0, \mathbf{x}) = \mathbf{u}_0(\mathbf{x}), & \mathbf{x} \in \mathcal{D}, \\ \mathbf{u}(t, \mathbf{x}) = \mathbf{u}_b(t, \mathbf{x}), & (t, \mathbf{x}) \in (0, T] \times \partial\mathcal{D}, \end{cases} \tag{3.6}$$

Where $\mathcal{N}$ is a nonlinear function of time t, space $\mathbf{x}$, solution $\mathbf{u}$, and its derivatives. $\mathcal{D} \subset \mathbb{R}^d$ is our bounded spatial domain. $\partial\mathcal{D}$ is defined to be our boundary of $\mathcal{D}$. T is the final time. $\mathbf{u}_0 : \mathcal{D} \to \mathbb{R}^d$ is our prescribed initial condition function and $\mathbf{u}_b : (0, T] \times \partial\mathcal{D} \to \mathbb{R}^d$ is our Dirichlet boundary condition. We then set up the residual network we aim to minimize. PINNs gets an approximate solution $\mathbf{u}_\theta(t, \mathbf{x})$ of (3.6) where $\mathbf{u}_\theta(t, \mathbf{x}) : [0, T] \times \mathcal{D} \to \mathbb{R}$ is a function realized by our PINNs network with parameters θ. We define the residual of our PINNs network as

$$\mathbf{r}_\theta(t, \mathbf{x}) := \mathbf{u}_{\theta_t} + \mathcal{N}(t, \mathbf{x}, \mathbf{u}_\theta, \mathbf{u}_{\theta_x}, \mathbf{u}_{\theta_{xx}}, ...). \tag{3.7}$$

where $\mathbf{u}_{\theta_t} = \frac{\partial \mathbf{u}_\theta}{\partial t}, \mathbf{u}_{\theta_x} = \frac{\partial \mathbf{u}_\theta}{\partial \mathbf{x}}, \mathbf{u}_{\theta_{xx}} = \frac{\partial^2 \mathbf{u}_\theta}{\partial \mathbf{x}^2}$. (3.7) is used in our loss function to be minimized by evaluating it at a set of collocation points for network training. $\mathcal{N}$ evaluates the right hand side of the PDE. For PINNs, we focus on three main groups of points: The residual (interior) points, the initial condition points, and the boundary condition points. Let N_r be defined as the number of residual points, N_0 be defined as the number of initial points, and N_b be defined as the number of boundary points. Then we define our data sets as,

- $X^r := \{(t_i^r, x_i^r)\}_{i=1}^{N_r} \subset (0, T] \times \mathcal{D}$,
- $X^0 := \{(t_i^0, x_i^0)\}_{i=1}^{N_0} \subset \{0\} \times \mathcal{D}$,
- $X^b := \{(t_i^b, x_i^b)\}_{i=1}^{N_b} \subset (0, T] \times \partial\mathcal{D}$.

The PINNs approach to get the solution to the PDE is to minimize the loss function $\Phi_\theta(X)$ defined below as

$$\Phi_\theta(X) := \Phi_\theta^r(X^r) + \Phi_\theta^0(X^0) + \Phi_\theta^b(X^b), \tag{3.8}$$

where $X = X^r \cup X^0 \cup X^b$ represents the whole training data set, and Φ_θ represents the loss function with respect to the parameters θ. The other terms within (3.8) are defined as

- mean squared residual

$$\Phi_\theta^r(X^r) := \tfrac{1}{N_r}\Sigma_{i=1}^{N_r}|\mathbf{r}_\theta(t_i^r, x_i^r)|^2,$$

  where $\mathbf{r}_\theta$ is the residual PINNs network,

- mean squared misfit with respect to initial conditions

$$\Phi_\theta^0(X^0) := \tfrac{1}{N_0}\Sigma_{i=1}^{N_0}|\mathbf{u}_\theta(t_i^0, x_i^0) - \mathbf{u}_0(x_i^0)|^2,$$

- mean squared misfit with respect to boundary conditions

$$\Phi_\theta^b(X^b) := \tfrac{1}{N_b}\Sigma_{i=1}^{N_b}|\mathbf{u}_\theta(t_i^b, x_i^b) - \mathbf{u}_b(t_i^b, x_i^b)|^2,$$

  where $\mathbf{u}_\theta$ is the neural network approximation of the solution $\mathbf{u} : [0, T] \times \mathcal{D} \to \mathbb{R}^d$.

3.3 The PyTorch Library

We employ Python3 and the machine learning package PyTorch for the implementation of PINNs. PyTorch is an open-sourced Python library designed for deep learning [24]. It has many powerful tools such as automatic differentiation which computes and records all of the gradients of the weights and biases automatically. It is important to note that automatic differentiation is neither a numerical differentiation scheme nor a symbolic differentiation scheme. It is is a mixture of the two. In machine learning, we cannot use numerical differentiation as the round-off and truncation errors make it highly inaccurate when computing gradients with respect to

millions of parameters [25]. Computing derivatives by hand and hard coding them into the neural network is inefficient and prone to error. Symbolic differentiation uses expression manipulation in computer algebra systems such as Maple, Mathematica, and Maxima. Such symbolic differentiation scheme solves the above issues, but is limited in what expressions can be defined, e.g., the work [25] states "Expressions must be closed-forms models which limits algorithmic control flow and expressivity". Furthermore, using symbolic differentiation often results in complex and cryptic expressions dominated by the "expression swell" problem [25]. As mentioned earlier, automatic differentiation can be viewed as partly numerical and partly symbolic. This is because it provides numerical values of derivatives instead of derivative expressions, and it does this by using symbolic rules of differentiation (it keeps track of the derivative value instead of the expression) [25]. Backpropagation is one such example of an automatic differentiation method.

PyTorch also has a variety of built-in optimization methods such as stochastic gradient descent, LBFGS, and Adam to name a few. In our experiments, we use Adam [23] as our optimizer of choice, as it is a powerful and commonly used option. A defining feature of PyTorch is its ability to access the computing power of a system's GPU which dramatically speeds up training (as GPUs are designed for large matrix multiplication), though this is not a focus for our experiments.

Chapter 4

Numerical Experiments

In this section, we perform various tests to confirm our theory as well as determine the benefit of using PINNs to form the coarse-grid time-stepping scheme in MGRIT. We begin defining a steady-state problem with a known analytical solution. Then we move to a more involved problem but with a similar nature. We perform stability tests with the CFL (Courant-Friedrichs-Lewy [26]) conditions using the forward Euler method. The goal is to see later on if MGRIT using PINNs on the coarse-grid offers more stability than standard MGRIT. We then look at how we train our PINNs network and how well this network approximates our solution. Finally we do a series of MGRIT tests with various parameters and compare how naive MGRIT does versus MGRIT using PINNs on the coarse grid.

4.1 The Problem: Advection Diffusion Equation

We begin by motivating the problem. As mentioned before, MGRIT typically struggles with hyperbolic problems. One of the most common examples of a hyperbolic problem is the advection-diffusion equation. For our initial example, we construct a

steady-state advection-diffusion equation for which we have an analytical solution. We then construct a more involved advection-diffusion equation, but this time, one that is dependent on time. The new problem will have a similar nature to the original equation so that we have a better understanding of its long-term behavior.

4.1.1 Steady-State Advection Diffusion Equation with Forcing Term

Before we set up the time-dependent problem, we first consider the steady-state advection diffusion equation for which we have an analytical solution. The boundary value problem is given as

$$\begin{cases} a\mathbf{u}_x = \epsilon \mathbf{u}_{xx} + \mathbf{F}(\mathbf{x}), & \mathbf{x} \in [0,1], \\ \mathbf{u}(0) = \alpha, \mathbf{u}(1) = \beta, & \alpha, \beta \in \mathbb{R}^n \end{cases} \tag{4.1}$$

where $\mathbf{u} = \mathbf{u}(\mathbf{x}, \mathbf{t}), \mathbf{u}_x = \frac{\partial \mathbf{u}(\mathbf{x},\mathbf{t})}{\partial \mathbf{x}}$, and $\mathbf{u}_{xx} = \frac{\partial^2 \mathbf{u}(\mathbf{x},\mathbf{t})}{\partial \mathbf{x}^2}$.

The solution, $\mathbf{u}$, measures the temperature (at different points in space) of a fluid moving through a pipe. The fluid moves at a constant velocity a, called the advection velocity and disperses at a constant $\epsilon > 0$, the diffusion constant. $\mathbf{F}(\mathbf{x}) \in \mathbb{R}^n$ is a forcing term or a source term heating or cooling the walls of the pipe.

For our example, we choose an advection velocity of $a = -1$, a diffusion constant $\epsilon = 0.01$, a forcing term of $\mathbf{F}(\mathbf{x})_i = -1$ and boundary conditions $\alpha_i = 1, \beta_i = 3$ $i = 0, ..., n$. An exact solution is given on page 44 of [27] as,

$$\mathbf{u}(\mathbf{x}) = \alpha + \mathbf{x} + (\beta - \alpha - 1)\left(\frac{e^{\mathbf{x}/\epsilon} - 1}{e^{1/\epsilon} - 1}\right). \tag{4.2}$$

A visualization of the solution (4.2) is given in figure 4.1. Now that we understand

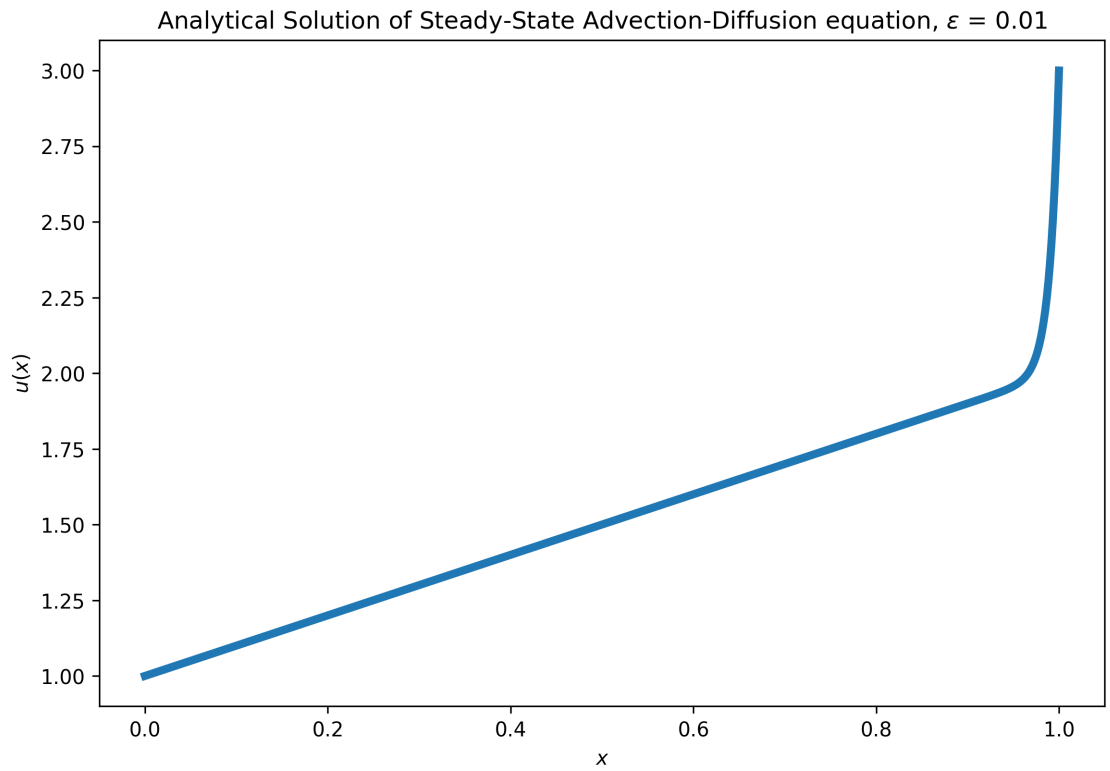


Figure 4.1: The analytical solution for the steady-state advection diffusion equation with a forcing term $F(x) = -1$. Boundary conditions are $u = 1$ at $x = 0$ and $u = 3$ at $x = 1$.

how the steady-state solution behaves, we can add a time derivative to make the problem more difficult, while still understanding the long-term behavior.

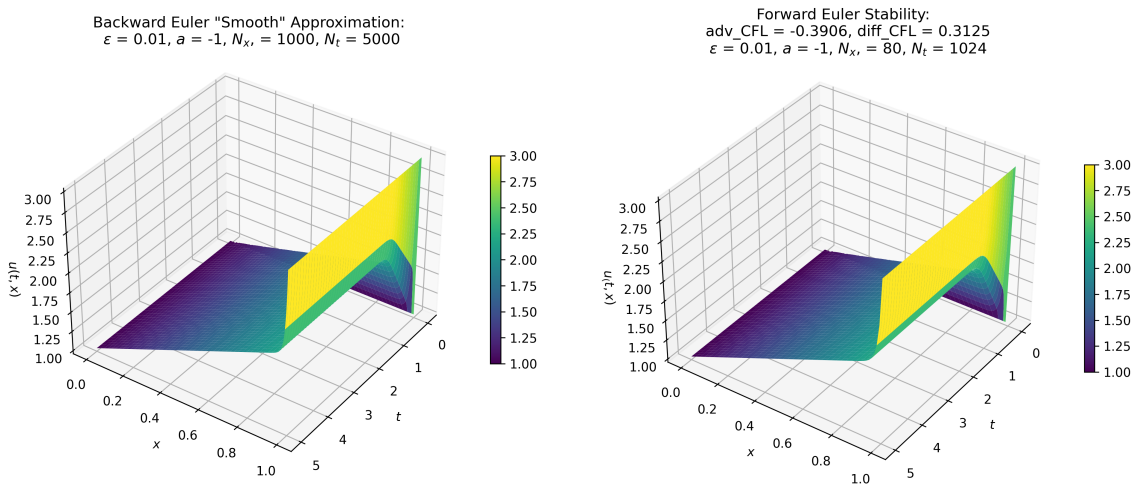
4.1.2 Time-Dependent Advection Diffusion Equation with Forcing Term

The time-dependent advection diffusion equation with forcing term is given as

$$\begin{cases} \mathbf{u}_t + a\mathbf{u}_x = \epsilon \mathbf{u}_{xx} + \mathbf{F}(\mathbf{x}), & t \in [0,5], \mathbf{x} \in [0,1], \\ \mathbf{u}(0,t) = 1, \mathbf{u}(1,t) = 3 & t \in [0,5], \\ \mathbf{u}(\mathbf{x},0) = 1, & \mathbf{x} \in [0,1]. \end{cases} \tag{4.3}$$

For all of the experiments, we set the advection term $a = -1$. Since we do not have an analytical solution, we use a numerical scheme with small step sizes in both space and time to get an approximate exact solution. We use a simple upwinding scheme (given in pages 210-211 in [27]) for our first-order spatial discretization, centered finite differencing for our second-order spatial discretization and either backward Euler' method (see appendix A) or forward Euler's method (see appendix B) for our temporal discretization. Our problem sizes will be defined in terms of N_t and N_x, the number of intervals in our time and space discretization, respectively. The value $\delta t = 5/N_t$ is the time step size and $\delta x = 1/N_x$ is the space step size.

We begin with backward Euler's method to avoid stability issues since it is an A-stable method [20]. We first create a reference solution which we refer to as a "smooth" approximation since we do not have an analytical solution. We use $N_t = 5000$ and $N_x = 1000$ for our problem size. Since our spatial domain is $[0,1]$ and our temporal domain is $[0,5]$, this gives us a time and space step size of $\delta t = \delta x = 0.001$. Figure 4.2a shows the "smooth" approximation to the solution, and figure 4.2b shows the stable reference forward Euler discretization. In later tests, we will examine PINNs as a more stable option for MGRIT than forward Euler.



(a) (b)

Figure 4.2: Figure (a) Backward Euler "smooth" approximate solution with time step size and space step sizes $\delta t = \delta x = 0.001$. Figure (b) Forward Euler solution when both advection and diffusion CFL conditions are met.

4.2 Initial Stability Tests for CFL Conditions

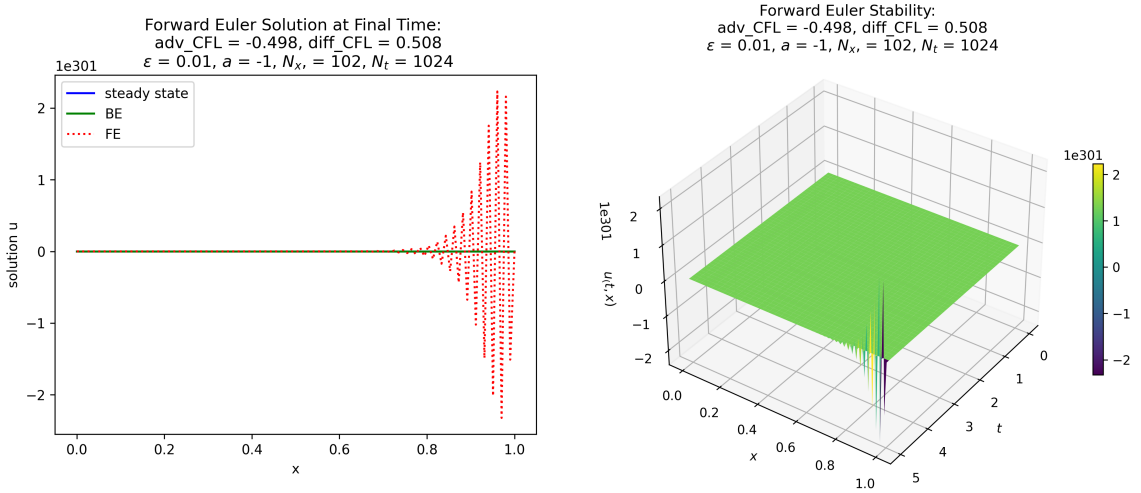
The CFL terms for stability are derived in Appendix B and are given as

$$\left|a\frac{\delta t}{\delta x}\right| \leq 1, \qquad \epsilon\frac{\delta t}{\delta x^2} \leq \frac{1}{2}, \tag{4.4}$$

from page 186 and 211 of [27] and page 133 of [20]. We demonstrate the effects of violating these conditions in the tests below. In the first test, we look at when both conditions are satisfied.

For the first test, we set $N_t = 1024$ and $N_x = 80$. This gives an advection term $|a\frac{\delta t}{\delta x}| = 0.391 < 1$ and diffusion term $\epsilon\frac{\delta t}{\delta x^2} = 0.313 < \frac{1}{2}$. Figure 4.2b shows that the forward Euler solution is computed with no stability issues. Next, we look at when the advection CFL condition is met and the diffusion CFL condition has failed.

Setting $N_t = 1024$ and $N_x = 102$ gives us an advection term $|a\frac{\delta t}{\delta x}| = 0.498 < 1$ and a diffusion term $\epsilon\frac{\delta t}{\delta x^2} = 0.508 > \frac{1}{2}$. Figure 4.3 shows that the solution becomes



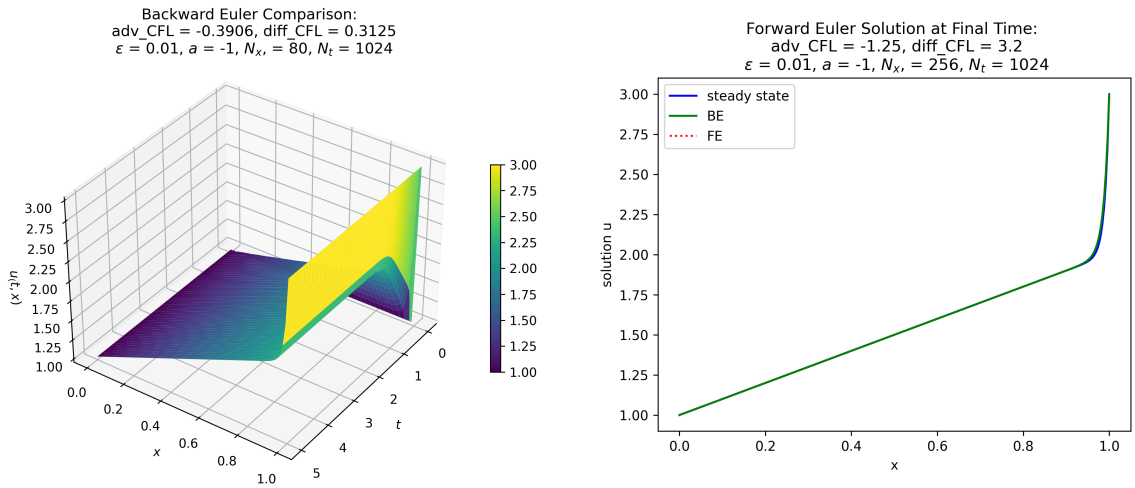
(a) (b)

Figure 4.3: Figure (a) Forward Euler solution at final time when the advection condition is met, but the diffusion condition failed. Figure (b) Space-time plot of the same unstable forward Euler solution.

unstable. Figure 4.3a shows us the instability at the final time. figure 4.3b shows the solution is unstable over the full space-time-domain. We compare the solution using backward Euler with the same parameters. This is seen in figure 4.4a, and shows no stability issues.

Since the solution is unstable when one of the CFL condition fails, it should be obvious that when both CFL conditions fail, the solution is also unstable. Setting $N_t = 1024$ and $N_x = 256$ gives us an advection term $|a\frac{\delta t}{\delta x}| = 1.25 > 1$ and a diffusion term $\epsilon\frac{\delta t}{\delta x^2} = 3.200 > \frac{1}{2}$. We see in figure 4.4b, that forward Euler, indicated by a red dotted line, does not show up in the plot at all. It was too unstable to get proper values to work with resulting in "NaN" or "Not a Number" values. For this reason, we do not give a plot of the full space-time domain.

For all of our tests in this thesis, we are working with an advection velocity of $a = -1$, so we do not need to be concerned with the case where the advection CFL



(a) (b)

Figure 4.4: Figure (a) Space-time plot of backward Euler solution using the same parameters as figure 4.3. Advection condition is met and the diffusion condition failed. Figure (b) Forward Euler solution at final time when both conditions failed. Forward Euler solution is unstable and does not show up on the figure.

condition fails and the diffusion CFL condition is met. As long as the advection velocity is $a = -1$ and $0 < \epsilon \leq 1$ then the advection CFL condition will always be satisfied as long as the diffusion CFL condition is met.

4.3 Training PINNs Network

We discuss the set up of our network, its parameters, and the software/hardware we run the experiments on. Experiments were ran using the Pytorch library (see section 3.3) using an Intel Core i7 Extreme 990X cpu with 6 cores and 12 threads.

4.3.1 Specifying Hyperparameters for Our Model

Recall back in chapter 3, we discussed hyperparameters such as learning rate, number of hidden layers, number of nodes per layer, number of epochs, and optimizers. In this section, we look at the specifications of each hyperparameter.

The decision to choose which hyperparameters to use came through trial and error. We tweaked these hyperparameters to result in the largest decrease in our loss function. Our model was trained using 4 hidden layers and 100 nodes per layer. We trained for $N = 8450$ epochs. We chose Adam [23] for our optimizer with an initial learning rate of $\eta = 0.01$ and restarted it with smaller learning rates after epochs the 1000th ($\eta = 0.001$) and 3000th ($\eta = 0.0005$) epochs. Since Adam is a stochastic method, we set our PyTorch seed to -42 for reproducibility. For our collocation points in our data set, we chose $N_0 = 100, N_b = 200$, and $N_r = 5000$ which are randomly, uniformly distributed on the space-time domain as illustrated in figure (4.5).

After training our model and evaluating it on a space-time domain of $N_t = 1024$ and $N_x = 32$, we get the solution in figure 4.6a.

We can get an idea of the accuracy by looking at the error between our PINNs evaluation and our backward Euler approximation on the same problem size. The error is shown in figure 4.6b. We can see that the PINNs evaluation itself is only accurate to about 1 to 2 digits of accuracy near the boundary. The bulk of the error comes from near the boundary for $x = 1$. As a stand-alone method, this is not very accurate but may still prove to be useful when using our trained network as the coarse-grid time-stepping scheme in the MGRIT algorithm.

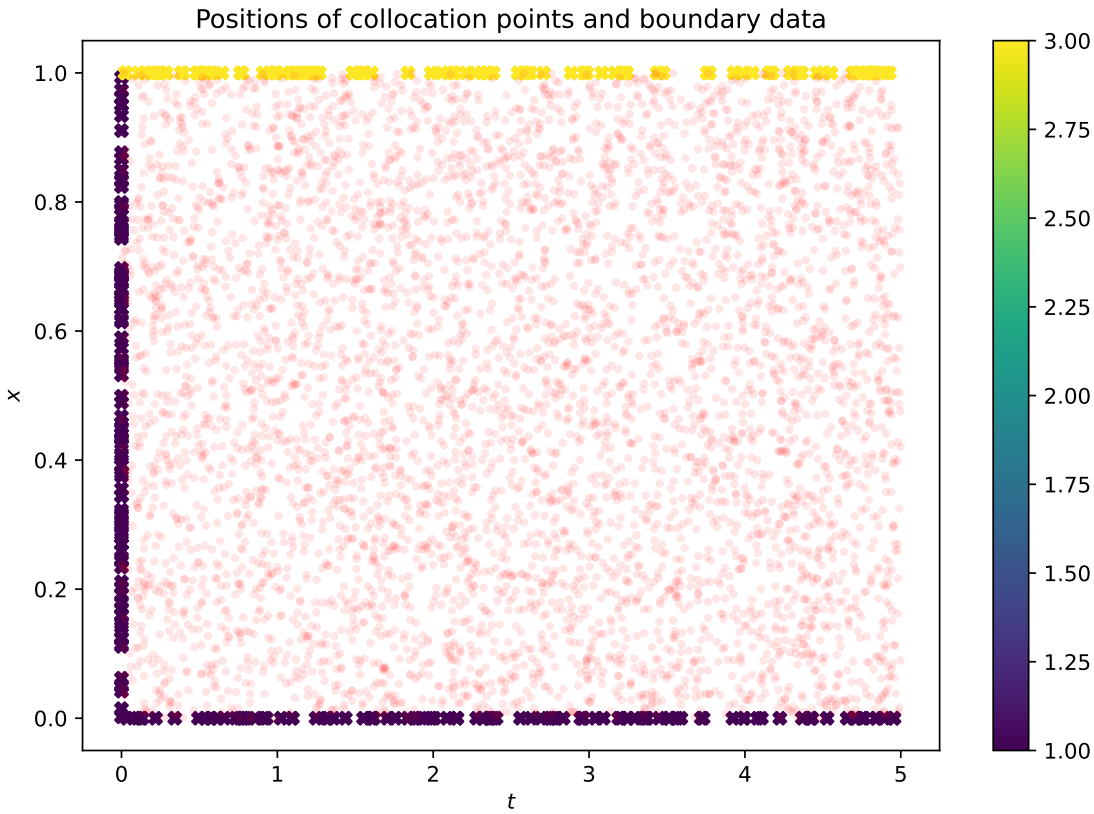
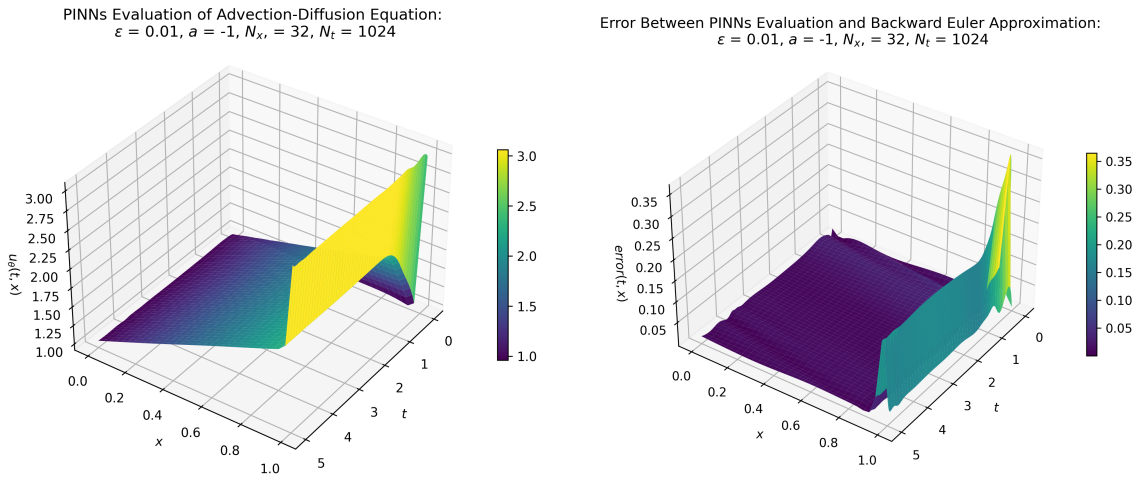


Figure 4.5: Collocation points visualized in our PINNs data set. The colored points correspond to the initial and boundary conditions.

4.3.2 PINNs on Coarse Grid

Before we implement PINNs in MGRIT, it is important to note that the PINNs method will typically get 4 to 5 digits of accuracy from the initial residual as typically witnessed in [10, 11, 12, 13]. And as our error demonstrated, we only got 1 to 2 digits of accuracy near the boundary in our model. Knowing this, it would not make sense to use it on the fine grid as the accuracy required for the fine grid is typically high. Thus we will only use PINNs on the coarser grids. For this thesis, we focus evaluating a trained model on the coarse grid of the 2-level MGRIT algorithm, but we also do a series of tests for the multilevel case. For the multilevel case, we look at what



(a) (b)

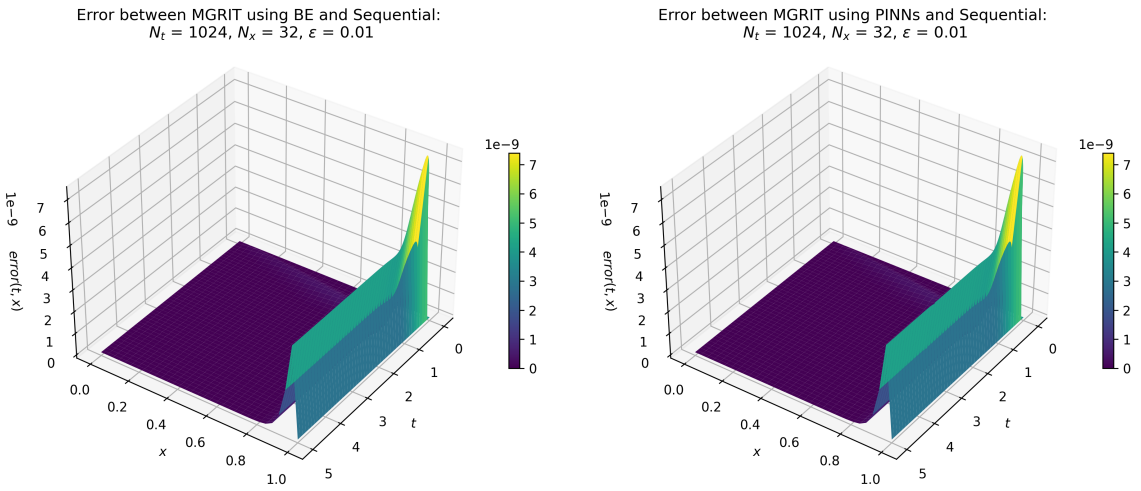
Figure 4.6: Figure (a) PINNs solution of advection diffusion equation trained using 8400 epochs, 50 initial collocation points, 200 boundary collocation points, and 5000 residual collocation points evaluated on a $N_t = 1024$ and $N_x = 32$ problem size. Figure (b) Shows the error between a backward Euler approximation and the PINNs evaluation on the same problem size.

happens when we only use PINNs on the coarsest grid versus using PINNs on all coarse levels.

4.4 MGRIT Experiments Using Fixed Diffusion Coefficient ϵ

4.4.1 Initial Convergence Tests

Before we begin our tests with naive MGRIT vs MGRIT with PINNs, we first verify that our MGRIT algorithm is converging to the sequential solution within tolerance. For our tests, we choose a residual tolerance of 10^{-10} on a problem size of $N_t = 1024$ and $N_x = 32$. This problem size was chosen for two reasons. First, we wanted



(a) (b)

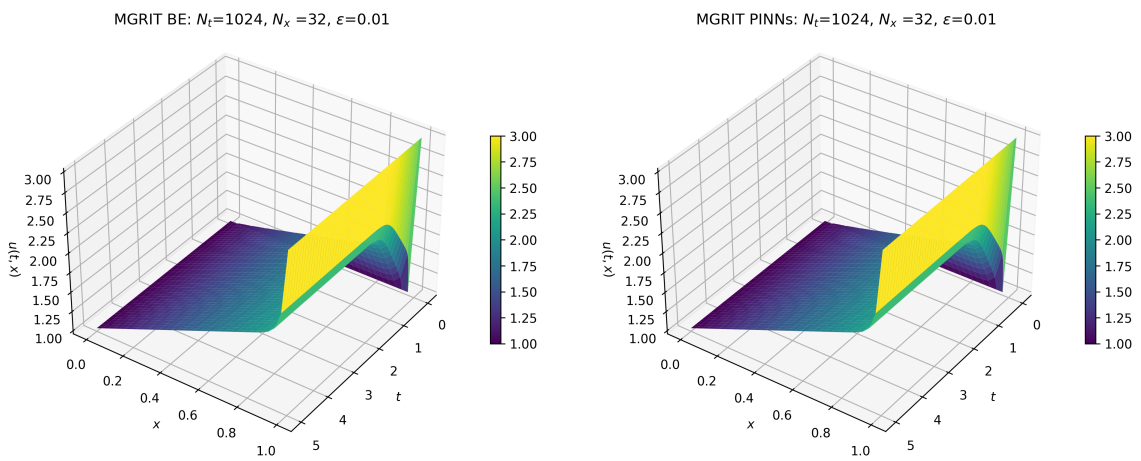
Figure 4.7: Figure (a) Error between naive MGRIT and our sequential solution using backward Euler. Figure (b) Error between MGRIT using PINNs and our sequential solution using backward Euler.

to meet the CFL stability conditions for when we later look at the forward Euler experiments. Second, most of the error comes from the temporal domain, so we need a smaller time step size to get an accurate solution, rather than a smaller space step size.

First, we compare MGRIT using backward Euler on two levels to our sequential solution. This is shown in figure 4.7a. Next, we look at the error between MGRIT using PINNs on the coarse grid, and using backward Euler on the fine grid. This is shown in figure 4.7b. Both errors appear to be the same within tolerance. With this being confirmed, we can now move onto the actual tests.

4.4.2 Naive MGRIT vs MGRIT with PINNs

In this section, we look at some tests for naive MGRIT vs MGRIT using PINNs on the coarse grid. We focus on 2-level MGRIT, but we also cover multi-level MGRIT



(a) (b)

Figure 4.8: Figure (a) 2-level MGRIT using backward Euler's method on both levels. Figure (b) 2-level MGRIT using backward Euler's method on the fine grid and PINNs on the coarse grid.

with a variety of ways implementing PINNs on different levels.

First, we compare naive MGRIT using backward Euler on both levels to MGRIT using PINNs on the coarse grid and using backward Euler on the fine grid. The numerical solutions for each is shown in figures 4.8a and 4.8b respectively. Both figures look nearly identical, but to be sure they are the same within tolerance, we look at the 2-norm of the error between the two solutions. The 2-norm between the two is 7.25×10^{-10}. A visualization of the error is given in figure 4.9.

While it is clear that the two solutions are the same within tolerance (10^{-10}), there are also clear signs that most of the error is coming from the corner from $x = 0.6$ to $x = 1.0$ and $t = 3$ to $t = 5$. This is something we will be looking out for later to see if it causes problems for our solver.

Another thing to note is the number of MGRIT cycles it took to converge. Naive MGRIT took 8 cycles to converge whereas MGRIT with PINNs took 169 cycles. One thing to note is that for the above tests, naive MGRIT's residuals were decreasing

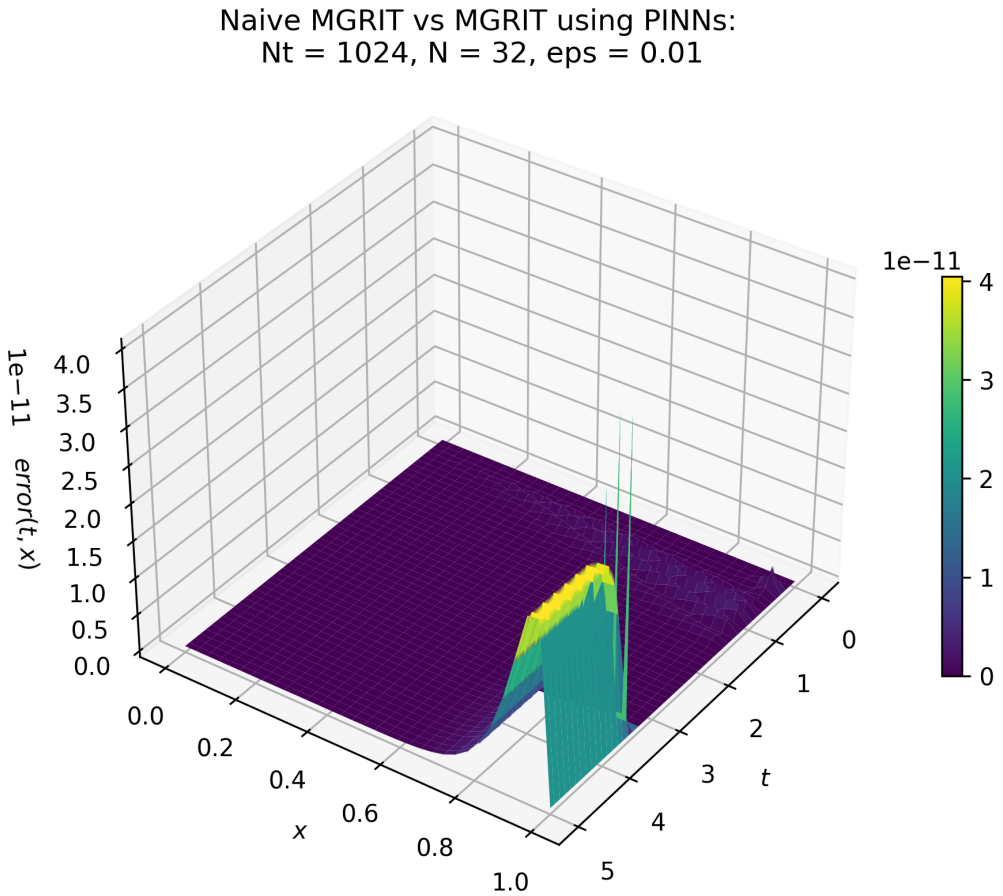
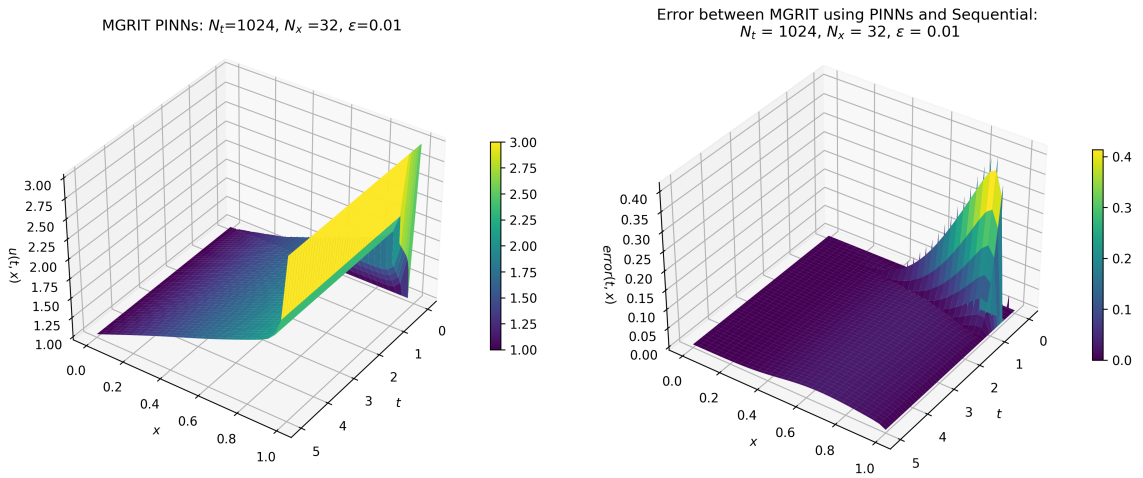


Figure 4.9: Error between naive MGRIT and MGRIT using PINNs on the coarse grid. The 2-norm of the error is 7.25×10^{-10}.

at around 10^{-1} per cycle (or gaining a digit of accuracy per cycle) whereas with PINNs, the residual decreased very slowly. This tells us that while stable, our PINNs evaluation is not very accurate. The underlying issue for this inaccuracy is currently unknown. However, one thing we can do is look at the error between the sequential solution and MGRIT using PINNs after forcibly being cut off after some number of cycles. The idea is that by ending the algorithm prematurely, we can see where most of the error is coming from. We run MGRIT using PINNs on the coarse grid, and end the algorithm after 25 iterations. The solution is shown in figure 4.10a.



(a) (b)

Figure 4.10: Figure (a) MGRIT using PINNs on coarse grid and forward Euler's method on fine grid for two levels. The problem size is $N_t = 1024$, $N_x = 32$. In this example, MGRIT was forcibly cut off prematurely after 25 iterations to get an idea of where the error was most prevalent. Figure (b) Error for MGRIT using PINNs on the coarse level after prematurely ending the algorithm in 25 cycles.

The error illustrated in figure 4.10b gives us an idea of where the bulk of the error is coming from. Unlike figure 4.9, the bulk of the error appears to be around $t = \delta t$ and $x = 0.4$ to $x = 1$. Both the solution and the error appear relatively accurate when it comes to the long-term behavior of the solution, but the algorithm struggles near the initial conditions. The root cause of this accumulation of error is unclear. It may be the case that whatever is causing the error to accumulate near the initial conditions is what is causing the slow MGRIT convergence in our algorithm using PINNs.

Next, we run tests for several different problem sizes with forward Euler's method and then later backward Euler's method. We begin with a problem size of $N_t = 500, N_x = 40$, and $\epsilon = 0.01$. For these tests, we look at using varying number of levels (2, 3 and 4). We compare naive MGRIT (which we refer to as "No PINNs" on the table), PINNs on the coarsest level (referred to as "Only Coarsest Level" on

the table) to PINNs used on every level aside from the fine level (referred to as "Full Coarse Grid" on the table). The results are listed in Table 4.1.

Wall Time	Cycles	Number Levels	Using PINNs?
43.442	118	2	No PINNs
64.017	73	2	Full Coarse Grid
91.485	115	3	No PINNs
92.826	115	3	Only Coarsest Level
126.119	73	3	Full Coarse Grid
71.580	117	4	No PINNs
84.100	116	4	Only Coarsest Level
151.342	73	4	Full Coarse Grid

Table 4.1: Naive MGRIT vs MGRIT using PINNs for a problem size $N_t = 500, N_x = 40, \epsilon = 0.01$ with a coarsening factor of 2. For the fine grid and every other level not using PINNs, we use forward Euler.

For this problem size, MGRIT using PINNs actually performs quite well versus naive MGRIT. In terms of cycles, we get a decrease of 45 cycles on the 2-level algorithm. For the multi-level algorithm, we mostly benefit when using PINNs on the full coarse grid, rather than just the coarsest level. For the wall-clock time, we see that naive MGRIT still outperforms MGRIT with PINNs. This is due to the fact that naive MGRIT is computationally cheaper in terms of FLOPS. It should also be mentioned that the implementation of PINNs has not been optimized for the fastest possible evaluation on each level. For this reason, we see that using PINNs on the full coarse grid is computationally the most expensive (in terms of wall-clock times) in every case. The more levels there are, the more computationally expensive using PINNs on the full coarse grid gets.

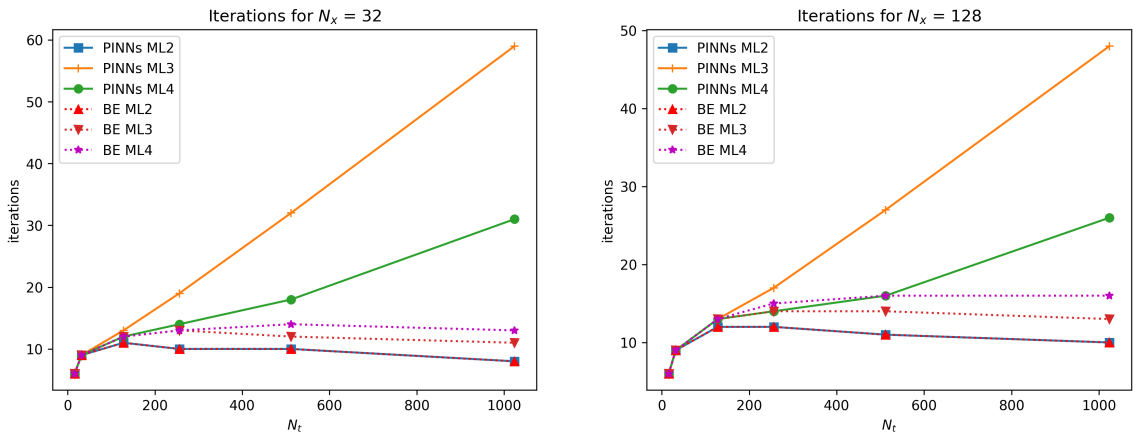
For this particular problem size, the exactness property should get the exact solution in $\frac{1024}{4} = 256$ iterations. Thus for the problems that took $115-118$ iterations to converge, they heavily relied on this property. This tells us that our coarse-grid method was only partially aiding in the convergence of these solutions.

Overall for the forward Euler case, it appears that adding PINNs to more levels decreases the number of cycles it takes to converge, but also increases the wall-clock time. Aside from the 2-level algorithm (in which PINNs performs quite well), using PINNs only on the coarsest level appears comparable to naive MGRIT in which both methods performed poorly. Now let's redo these tests, but this time replacing the forward Euler method with the backward Euler method.

Wall Time	Cycles	Number Levels	Using PINNs?
5.702	10	2	No PINNs
91.089	82	2	Full Coarse Grid
10.897	13	3	No PINNs
33.883	30	3	Only Coarsest Level
158.604	82	3	Full Coarse Grid
13.465	14	4	No PINNs
19.992	17	4	Only Coarsest Level
192.307	82	4	Full Coarse Grid

Table 4.2: Naive MGRIT vs MGRIT using PINNs for a problem size $N_t = 500, N_x = 40, \epsilon = 0.01$ with a coarsening factor of 2. For the fine grid and every other level not using PINNs, we use backward Euler.

Table 4.2 shows us almost completely different results from table 4.1. For every case when using backward Euler's method, naive MGRIT takes less cycles to converge than when using PINNs on any part of the coarse grid. In the case of forward Euler, using PINNs on the full coarse grid caused a decrease in the number of cycles needed to converge. But in the case of backward Euler, using PINNs on the full coarse grid caused an increase in the number of cycles needed to converge (the opposite). Out of all of the cases, the only instance where PINNs performed similarly to "No PINNs" was the 4-level case for "No PINNs" versus "Only Coarsest Level." It is also worth noting that it is much more computationally expensive using PINNs on the coarse grid for the backward Euler case. Overall, it appears when using backward Euler's method, it is better to stick with naive MGRIT than to using PINNs on the



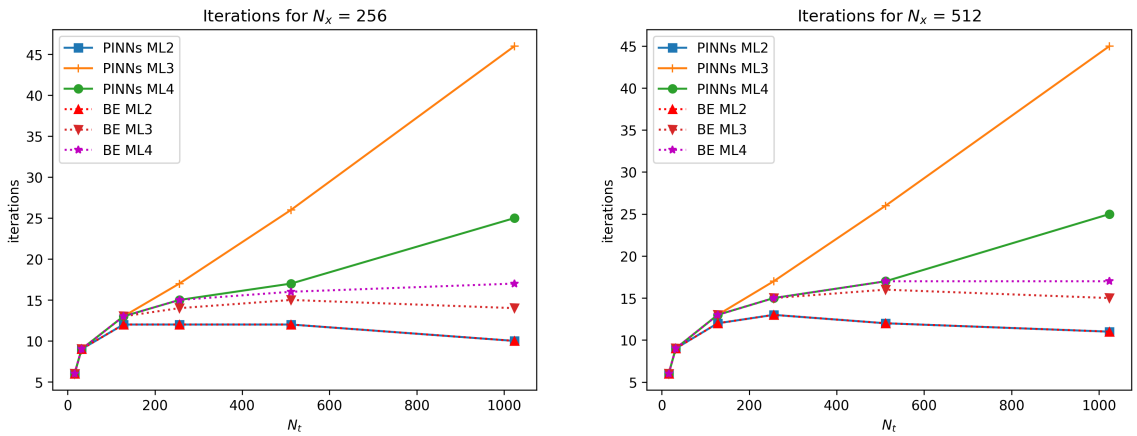
(a) (b)

Figure 4.11: Figure (a) Fixed $N_x = 32$. Number of iterations for different methods. ML means maximum number of levels used. BE means using backward Euler's method on all levels. PINNs means using PINNs on the coarsest grid only. Figure (b) Fixed $N_x = 128$. Number of iterations for different methods. ML means maximum number of levels used. BE means using backward Euler's method on all levels. PINNs means using PINNs on the coarsest grid only.

coarse/coarsest grid.

Since the above examined only one problem size, we should next look at how MGRIT using PINNs performs on many different problem sizes. For each test, we will have a fixed N_x and varying N_t. We look at $N_x = 32, 128, 256$ and 512. N_t values will tested at $16, 32, 128, 256, 512$, and 1024. For these tests, we will be using PINNs only on the coarsest grid to save computational time. The coarsening factor will be $m = 2$.

Figure 4.11a shows how MGRIT using PINNs performs when using the 2-level, 3-level, and 4-level algorithms and backward Euler. Quite surprisingly, the 2-level algorithm using PINNs performs quite well, directly comparable to 2-level, naive MGRIT ("BE ML2"). This is not the case for the 3-level and 4-level cases. Out of all the cases, the 3-level algorithm using PINNs on the coarsest grid performs the

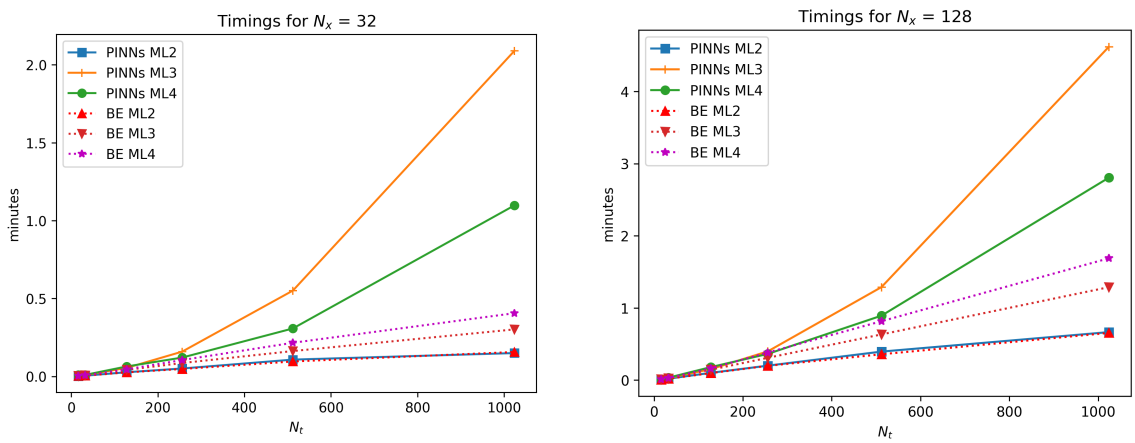


(a) (b)

Figure 4.12: Figure (a) Fixed $N_x = 256$. Number of iterations for different methods. ML means maximum number of levels used. BE means using backward Euler's method on all levels. PINNs means using PINNs on the coarsest grid only. Figure (b) Fixed $N_x = 512$. Number of iterations for different methods. ML means maximum number of levels used. BE means using backward Euler's method on all levels. PINNs means using PINNs on the coarsest grid only.

worst. The number of iterations continues to grow relatively large, as N_t grows. The 4-level algorithm performs decently up to $N_t = 256$, but it continues to grow after that (non-optimal). This trend continues for $N_x = 128$, $N_x = 256$, and $N_x = 512$ as shown in figures 4.11b, 4.12a and 4.12b.

Overall, it appears that the 2-level algorithm using PINNs is the only consistently comparable method to naive MGRIT. It performs nearly identical to the 2-level, naive MGRIT algorithm. For smaller problem sizes, 4-level MGRIT using PINNs performs well, but the method is not optimal. This is because after $N_t = 256$, as N_t increases, the number of iterations increase. In every case, 3-level MGRIT using PINNs performed the worst. The method is not optimal because of the steadily increasing iteration counts for even smaller problem sizes. Note, these results are in agreement with our earlier observations from 4.2. But before we discuss the cases using forward Euler, let's first look at the computational time it takes to complete

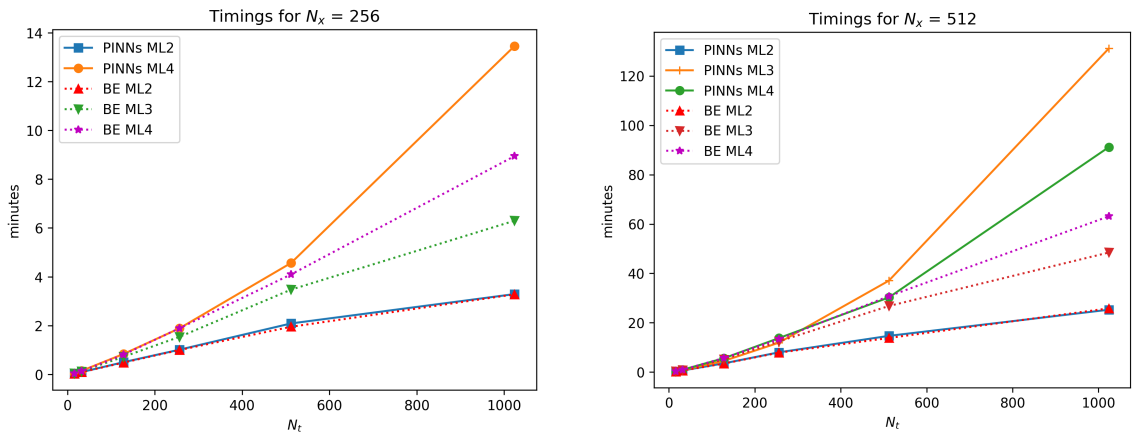


(a) (b)

Figure 4.13: Figure (a) Fixed $N_x = 32$. Wall clock times for different methods. ML means maximum number of levels used. BE means using backward Euler's method on all levels. PINNs means using PINNs on the coarsest grid only. Figure (b) Fixed $N_x = 128$. Wall clock times for different methods. ML means maximum number of levels used. BE means using backward Euler's method on all levels. PINNs means using PINNs on the coarsest grid only.

each MGRIT solve. Figures 4.13 and 4.14, show the time (in minutes) for MGRIT to converge for different problem sizes. For all of the tests conducted, it appears that using PINNs with a maximum number of levels set to 3 took the longest to complete. Note these tests are only looking at MGRIT using PINNs on the coarsest level rather than the full coarse grid (to save computation time). However, these results may be misleading. Since MGRIT using PINNs can be be solved fully in parallel whereas naive MGRIT must be solved sequentially on the coarsest level.

When looking at the results from Tables 4.1 and 4.2, we observe that MGRIT appears to benefit more from PINNs when using forward Euler than backward Euler. With backward Euler, it took more iterations to converge and adding PINNs to more levels didn't improve convergence (as it did in the forward Euler case). Thus in the next section, we look at how MGRIT using PINNs on the coarse grid performs when using forward Euler.



(a) (b)

Figure 4.14: Figure (a) Fixed $N_x = 256$. Wall clock times for different methods. ML means maximum number of levels used. BE means using backward Euler's method on all levels. PINNs means using PINNs on the coarsest grid only. Figure (b) Fixed $N_x = 512$. Wall clock times for different methods. ML means maximum number of levels used. BE means using backward Euler's method on all levels. PINNs means using PINNs on the coarsest grid only.

4.4.3 Stability Tests

For the forward Euler tests on varying problem sizes we have to consider the CFL conditions in (4.4). We list two tables of the conditions for advection and diffusion, where the CFC conditions fail and when they are met.

Figure 4.15 shows us that both of the CFL conditions are met for problem sizes $N = 32$ with $N_t = 256, 512$, and 1024. For this reason, we focus our comparisons between naive MGRIT and MGRIT using PINNs on these particular problem sizes. Before we list our results, let's look at the CFL terms on the coarse grid with $m = 2$.

Advection CFL Condition Tests

N	N_t	16	32	128	256	512	1024
32		10	5	1.25	0.625	0.3125	0.15625
128		40	20	5	2.5	1.25	0.625
256		80	40	10	5	2.5	1.25
512		160	80	20	10	5	2.5

Diffusion CFL Condition Tests

N	N_t	16	32	128	256	512	1024
32		3.2	1.6	0.4	0.2	0.1	0.05
128		51.2	25.6	6.4	3.2	1.6	0.8
256		204.8	102.4	25.6	12.8	6.4	3.2
512		819.2	409.6	102.4	51.2	25.6	12.8

Figure 4.15: Two tables of the CFL terms for advection $|a\frac{\delta t}{\delta x}|$ and diffusion $\epsilon\frac{\delta t}{\delta x^2}$. Each entry looks at these terms evaluated at different problem sizes for N and N_t and compares them to the CFL conditions (4.4). The green indicates the conditions are met and the red indicates the conditions are not met.

We see that for $N_t = 128$, the fine grid is partially unstable. The advection CFL condition is not met, but the diffusion condition is met. For $N_t = 256, 512$, and 1024, both conditions are met on the fine grid. For the coarse-grid in figure 4.16, we see the $N_t = 128$ becomes completely unstable and $N_t = 256$ becomes partially unstable. This is important to note for our tests shown next.

Advection CFL Condition Tests for Coarse Grid

N	N_t	16	32	128	256	512	1024
32		20	10	2.5	1.25	0.625	0.3125

Diffusion CFL Condition Tests Coarse Grid

N	N_t	16	32	128	256	512	1024
32		6.4	3.2	0.8	0.4	0.2	0.1

Figure 4.16: Two tables of the CFL terms for advection $|a\frac{\delta T}{\delta x}|$ and diffusion $\epsilon\frac{\delta T}{\delta x^2}$ on the coarse grid. Each entry looks at these terms evaluated at different problem sizes for N and N_t and compares them to the CFL conditions 4.4. The green indicates the conditions are met and the red indicates the conditions are not met.

2-Level Naive MGRIT using Forward Euler on Both Levels

N \ N_t	16	32	128	256	512	1024
32	div	div	div	67	13	10

Table 4.3: MGRIT tests for forward Euler running on both levels for various problem sizes. The number indicates how many iterations it took to converge whereas "div" indicates that the test diverged.

2-Level MGRIT using PINNs on coarse grid and Forward Euler on Fine Grid

N \ N_t	16	32	128	256	512	1024
32	div	div	div	48	80	161

Table 4.4: MGRIT tests using PINNs on coarse grid and forward Euler on fine grid for various problem sizes. The number indicates how many iterations it took to converge whereas "div" indicates that the test diverged.

Tables 4.3 and 4.4 then list the MGRIT convergence results for naive MGRIT and MGRIT with PINNs, respectively. Notice that while figure 4.15 shows that for $N = 32, N_t = 256$, the CFL conditions are met on the fine grid, but table 4.16 shows that the CFL condition for advection is not met on the coarse grid. This is also precisely the problem size that MGRIT using PINNs outperformed naive MGRIT. This tells us that MGRIT using PINNs is slightly more stable than naive MGRIT. But we see when the CFL conditions are met on both the fine and coarse grid, naive MGRIT outperforms MGRIT using PINNs (much better). Let's look at one more test to confirm this hypothesis. We look at the coarse grid with a coarsening factor of $m = 4$ making the coarse grid twice as unstable. Similar to above, figure 4.17, lists the values for advection $|a\frac{\delta T}{\delta x}|$ and diffusion $\epsilon\frac{\delta T}{\delta x^2}$ to indicate whether or not the CFL conditions are met.

Advection CFL Condition Tests for Coarse Grid m = 4

N	N_t	16	32	128	256	512	1024
32		40	20	5	2.5	1.25	0.625

Diffusion CFL Condition Tests Coarse Grid

N	N_t	16	32	128	256	512	1024
32		12	6.4	1.6	0.8	0.4	0.2

Figure 4.17: Two tables of the CFL terms for advection $|a\frac{\delta T}{\delta x}|$ and diffusion $\epsilon\frac{\delta T}{\delta x^2}$ on the coarse grid with coarsening factor $m = 4$. Each entry looks at these terms evaluated at different problem sizes for N and N_t and compares them to the CFL conditions 4.4. The green indicates the conditions are met and the red indicates the conditions are not met.

2-Level Naive MGRIT using Forward Euler on Both Levels $m = 4$

N \ N_t	256	512	1024
32	34	52	40

Table 4.5: MGRIT tests for forward Euler running on both levels for various problem sizes with coarsening factor $m = 4$. The number indicates how many iterations it took to converge whereas "div" indicates that the test diverged.

2-Level MGRIT using PINNs on coarse grid and Forward Euler on Fine Grid $m = 4$

N \ N_t	256	512	1024
32	25	40	81

Table 4.6: MGRIT tests using PINNs on coarse grid with coarsening factor $m = 4$ and forward Euler on fine grid for various problem sizes. The number indicates how many iterations it took to converge whereas "div" indicates that the test diverged.

When you compare how naive MGRIT does versus MGRIT using PINNs on the coarse grid, it is clear that when the CFL condition is met on the fine grid, but

fails on the coarse grid, MGRIT using PINNs actually performs better than naive MGRIT! For the problem size, $N = 32, N_t = 256$, figure 4.17 shows that both the advection and diffusion CFL condition fails on the coarse grid (but is met on the fine grid). Likewise, tables 4.5 and 4.6 show that MGRIT using PINNs takes 9 less iterations to converge than naive MGRIT. For the problem size $N = 32, N_t = 512$, figure 4.17 shows that while the diffusion CFL condition is met, the advection CFL condition is not met. For this problem size, we again see that MGRIT using PINNs does better. It takes 12 less iterations to converge to the solution than naive MGRIT.

However, once the CFL conditions are met, naive MGRIT outperforms MGRIT using PINNs on the coarse grid. Figure 4.17 shows that both CFL conditions are met, and tables 4.5 and 4.6 show that naive MGRIT takes about half the number of iterations to converge. Thus we can say that when the CFL conditions are met, it is better to use naive MGRIT. However, when the coarse grid becomes unstable, it is actually preferable to use MGRIT using PINNs on the coarse grid.

4.5 Discussion of MGRIT Convergence when using PINNs

Since we have shown benefit to using PINNs on the coarse grid in the cases with an unstable coarse grid, future research should look at these tests with a more accurate PINNs model since our model obtains only 1-2 digits on accuracy near the boundary. Currently, naive MGRIT performs a lot better when the CFL conditions are met on all levels, but would this still be the case for when using a highly accurate PINNs model?

In all of the tests conducted, regardless of problem size, XBraid's residual began on the order of 10^0 or 10^{-1} when using MGRIT with PINNs on the coarse grid. On

the other hand, the residual would vary greatly depending on the stability of the method when using naive MGRIT. For example, when forward Euler was used and a CFL condition(s) fails, then the residual would grow without bound and eventually diverge. When the advection and diffusion terms approached the CFL conditions, the residual would grow, then eventually decrease resulting in a large number of iterations needed to converge.

If we could replicate the accuracy of naive MGRIT (getting a digit of accuracy per MGRIT cycle) with the stability of MGRIT using PINNs on the coarse grid, we should be able to see convergence in 9 to 10 XBraid cycles, regardless of problem size (resulting in an optimal method). Future work should prioritize this line of thinking and target increased accuracy of the PINNs model used on coarse MGRIT levels. If better PINNs accuracy could be achieved, the residual should in turn decrease much faster than has been observed during these experiments. The work [16] discusses some shortcomings of PINNs and also techniques on how to improve accuracy. Another possible approach would be to use techniques in [28]. These papers could provide guidance for improving the PINNs model for coarse MGRIT levels.

4.6 Conclusions

This thesis looked at comparisons between using naive (standard) MGRIT and MGRIT using PINNs on the coarse grid for a problem with hyperbolic behavior (the advection-diffusion equation with small diffusion coefficient). The goal was to see whether or not adding PINNs would improve convergence and stability of MGRIT for hyperbolic problems. For the backward Euler (implicit method) case, naive MGRIT outperforms MGRIT using PINNs on the coarse grid. This is shown in terms of the number of iterations it takes to converge to a solution. For backward Euler, this number was much lower. Another note for backward Euler is that adding

PINNs to more coarse-grid levels did not appear to improve convergence whereas it did for the forward Euler case. However, the two-level method appeared to be comparable for naive MGRIT and MGRIT using PINNs. For the forward Euler (explicit method) case, we look at when the CFL conditions for advection and diffusion were met and when they were not met. We looked at different problem sizes with that introduce different stabilities and instabilities within the method. We found that when the CFL conditions were not met on any of the levels (fine grid or coarse grid), MGRIT using PINNs outperformed naive MGRIT. Furthermore, adding PINNs to each level (Full coarse grid) in the coarse grid caused an increase in convergence. But when the CFL conditions were met on every level, naive MGRIT outperformed MGRIT using PINNs. The overall take away has been that MGRIT using PINNs is more stable which is good for explicit methods with varying coarse-grid levels, but is not as accurate as naive MGRIT. Since implicit methods deal with stability, there doesn't seem to be reason to use PINNs with these methods. For explicit methods involving multiple coarse-grid equations, MGRIT using PINNs is generally preferable as varying levels increasing instability on each level.

Chapter 5

Future Work

Since it appears that the increased convergence of MGRIT is highly dependent upon the accuracy for the PINNs model, this should be a focus for future research. One way to go about this is to implement techniques discussed in [16] when training the model. It could be beneficial to use higher order discretizations for the numerical schemes for temporal methods (such as Runge-Kutta 4) and higher order up-winding methods for spatial discretizations. One way to use these methods, would be to get the error between a highly accurate method (higher-order method with small space and time step sizes) and lower-order method to be on the same magnitude as the error between the fine grid solver and PINNs. Once we get the error to be of the same magnitude, we could use this lower-order method as our coarse grid for the naive MGRIT tests. This way, the methods used to solve the coarse grid can be more comparable in terms of accuracy.

Another area of research could involve optimizing the evaluation of PINNs within Xbraid. Once optimized (and assuming we achieved better accuracy for PINNs evaluations), we can begin doing parallel studies using PINNs on the coarse grid. Since the coarse grid is solved sequentially using naive MGRIT, it is theoretically

possible to improve solving the coarse grid using PINNs. This is because we are able to parallelize the evaluation of PINNs on the coarse grid. This means that we could solve both the fine and coarse grids in parallel when using PINNs. If we can achieve a more accurate model as well as implement PINNs optimally, we can then work to obtain parallel speed up (overarching goal).

A separate area we can look at is the case when training a PINNs network on varying diffusion and advection coefficients rather than fixed coefficients. With this newly trained model, we could evaluate multiple different PDEs given any coefficients within the range trained on. This could possibly lead to speed up for solving many different problems as opposed to training a network for each individual problem. Several attempts were made to train such a network (ϵ varied from 0.01 to 1), but the added parameters and hyperparameters resulted in a low accuracy solution leading the task to be halted for future work. Perhaps future work can look into using techniques discussed in [28] to achieve more accurate results.

Finally, our PINNs network was trained to get a continuous solution of the PDE. But since our convergence-error bound is based on the coarse-grid eigenvalues approximating the eigenvalues of m applications of the fine-grid propagator, we could instead train a PINNs network to explicitly target these eigenvalues to better approximate the fine-grid propagator's eigenvalues. This could lead to faster convergence of our method.

References

[1] M. J. Gander, "50 years of time parallel time integration."

[2] R. D. Falgout, S. Friedhoff, T. V. Kolev, S. P. MacLachlan, and J. B. Schroder, "Parallel time integration with multigrid," *SIAM Journal on Scientific Computing*, vol. 36, no. 6, pp. C635–C661, 2014.

[3] V. Dobrev, T. Kolev, N. A. Petersson, and J. B. Schroder, "Two-level convergence theory for multigrid reduction in time (mgrit)," *SIAM Journal on Scientific Computing*, vol. 39, no. 5, pp. S501–S527, 2017.

[4] F. Danieli and S. P. MacLachlan, "Multigrid reduction in time for non-linear hyperbolic equations," *ArXiv*, vol. abs/2104.09404, 2021.

[5] A. Katz, V. Kolev, and J. B. Schroder, "Parallel time integration with multigrid reduction for a compressible fluid dynamics application," 2014.

[6] J. Christopher, X. Gao, S. M. Guzik, D. Robert, Falgout, and J. B. Schroder, "Fully parallelized space-time adaptive meshes for the compressible navier-stokes equations using multigrid reduction in time," 2019.

[7] R. D. Falgout, T. A. Manteuffel, B. O'Neill, and J. B. Schroder, "Multigrid reduction in time for nonlinear parabolic problems: A case study," *SIAM J. Sci. Comput.*, vol. 39, 2017.

[8] X. W. Gao, S. M. Guzik, R. D. Falgout, and J. B. Schroder, "Space-time adaptivity with multigrid reduction in time for the compressible navier-stokes equations," 2019.

[9] A. Hessenthaler, R. D. Falgout, J. B. Schroder, A. de Vecchi, D. A. Nordsletten, and O. Röhrle, "Time-periodic steady-state solution of fluid-structure interaction and cardiac flow problems through multigrid-reduction-in-time," *ArXiv*, vol. abs/2105.00305, 2022.

[10] M. Raissi, P. Perdikaris, and G. E. Karniadakis, "Physics informed deep learning (part i): Data-driven solutions of nonlinear partial differential equations," 2017.

[11] M. Raissi, P. Perdikaris, and G. E. Karniadakis, "Physics informed deep learning (part ii): Data-driven discovery of nonlinear partial differential equations," 2017.

[12] M. Raissi, P. Perdikaris, and G. Karniadakis, "Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations," *Journal of Computational Physics*, vol. 378, pp. 686–707, 2019.

[13] G. E. Karniadakis, I. G. Kevrekidis, L. Lu, P. Perdikaris, S. Wang, and L. Yang, "Physics-informed machine learning," *Nature Reviews Physics*, vol. 3, no. 6, pp. 422–440, 2021.

[14] J. Blechschmidt and O. G. Ernst, "Three ways to solve partial differential equations with neural networks – a review," 2021.

[15] M. Raissi, "Deep hidden physics models: Deep learning of nonlinear partial differential equations," 2018.

[16] A. S. Krishnapriyan, A. Gholami, S. Zhe, R. M. Kirby, and M. W. Mahoney, "Characterizing possible failure modes in physics-informed neural networks," 2021.

[17] "Xbraid: Parallel multigrid in time." http://llnl.gov/casc/xbraid.

[18] A. Brandt, "Multi–level adaptive solutions to boundary–value problems," *Math. Comp.*, vol. 31, no. 138, pp. 333–390, 1977.

[19] W. Briggs, V. Henson, and S. McCormick, *A Multigrid Tutorial: Second Edition*. Other Titles in Applied Mathematics, Society for Industrial and Applied Mathematics, 2000.

[20] K. E. Atkinson, W. Han, and D. E. Stewart, "Numerical solution of ordinary differential equations: Atkinson/numerical," 2009.

[21] K. Hornik, M. Stinchcombe, and H. White, "Multilayer feedforward networks are universal approximators," *Neural Networks*, vol. 2, no. 5, pp. 359–366, 1989.

[22] C. F. Higham and D. J. Higham, "Deep learning: An introduction for applied mathematicians," 2018.

[23] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," 2014.

[24] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems 32*, pp. 8024–8035, Curran Associates, Inc., 2019.

[25] A. G. Baydin, B. A. Pearlmutter, A. A. Radul, and J. M. Siskind, "Automatic differentiation in machine learning: a survey," 2015.

[26] R. Courant, K. Friedrichs, and H. Lewy, "On the partial difference equations of mathematical physics," *IBM Journal of Research and Development*, vol. 11, no. 2, pp. 215–234, 1967.

[27] R. LeVeque, *Finite Difference Methods for Ordinary and Partial Differential Equations: Steady-State and Time-Dependent Problems (Classics in Applied Mathematics Classics in Applied Mathemat)*. USA: Society for Industrial and Applied Mathematics, 2007.

[28] L. Lu, P. Jin, G. Pang, Z. Zhang, and G. E. Karniadakis, "Learning nonlinear operators via DeepONet based on the universal approximation theorem of operators," *Nature Machine Intelligence*, vol. 3, pp. 218–229, mar 2021.

Appendices

Appendix A

Backward Euler Discretization

$$\begin{cases} u_t - au_x = \epsilon u_{xx} + F, \\ u(0,t) = d_0(t), \quad u(1,t) = d_1(t), \\ u(x,0) = c(x). \end{cases} \tag{A.1}$$

We begin by discretizing u_{xx} using centered finite differencing. Let $F_i = F(t, x_i)$.

$u_{xx} \approx \frac{u_{i+1} - 2u_i + u_{i-1}}{\delta x^2}$ with spatial step size $\delta x = 1/m$ for nodes $x_j = j\delta x, \quad j = 0, 1, ..., m$.

For the advection diffusion equation, we can discretize the u_x term using forward, backward, or centered finite differencing. Here we will be using the "Upwind method" or "Upstream differencing method." Since the solution of the advection diffusion equation can be thought of as a wave moving to the left or right (depending on whether a, the advection velocity, is positive or negative). We can take advantage of this asymmetry by using a one sided approximation method, i.e. the forward or backward finite difference method.

Appendix A. Backward Euler Discretization

If $a > 0$, then the wave is moving to the left. If $a < 0$, then the wave is moving to the right. Depending on the direction, we will need information from the left or right of the points in order to get an accurate result. For example, if $a < 0$, then the wave is moving to the right. We will need information to the left of x_j (i.e. need x_{j-1}). Thus we would use backward finite differencing in this case

$u_x = \frac{u_j - u_{j-1}}{\delta x}$

Then we can approximate the advection diffusion equation as

$u_i'(t) - a u_x = \epsilon \frac{u_{i+1} - 2u_i + u_{i-1}}{\delta x^2} + F_i$

$u_i'(t) - a \frac{u_i - u_{i-1}}{\delta x} = \epsilon \frac{u_{i+1} - 2u_i + u_{i-1}}{\delta x^2} + F_i$

$u_i'(t) = \frac{a}{\delta x}(u_i - u_{i-1}) + \frac{\epsilon}{\delta x^2}\big(u_{i+1} - 2u_i + u_{i-1}\big) + F_i \quad$ for $i = 1, 2, ..., m-1$. Where $u_0 = d_0(t)$ and $u_m = d_1(t)$ are the boundary conditions

Let's look at some cases, $i = 1, i = 2, ..., m-1$
$u_1'(t) = \frac{a}{\delta x}(u_1 - u_0) + \frac{\epsilon}{\delta x^2}\big(u_2 - 2u_1 + u_0\big) + F_1 = \frac{a}{\delta x}(u_1 - d_0(t)) + \frac{\epsilon}{\delta x^2}\big(u_2 - 2u_1 + d_0(t)\big) + F_1$

$u_2'(t) = \frac{a}{\delta x}(u_2 - u_1) + \frac{\epsilon}{\delta x^2}\big(u_3 - 2u_2 + u_1\big) + F_2$

$\vdots$

$u_{m-2}'(t) = \frac{a}{\delta x}(u_{m-2} - u_{m-3}) + \frac{\epsilon}{\delta x^2}\big(u_{m-1} - 2u_{m-2} + u_{m-3}\big) + F_{m-2}$

$u_{m-1}'(t) = \frac{a}{\delta x}(u_{m-1} - u_{m-2}) + \frac{\epsilon}{\delta x^2}\big(u_m - 2u_{m-1} + u_{m-2}\big) + F_{m-1} = \frac{a}{\delta x}(u_{m-1} - u_{m-2}) + \frac{\epsilon}{\delta x^2}\big(d_1(t) - 2u_{m-1} + u_{m-2}\big) + F_{m-1}$

Notice: For $i = 1, ..., m-1$, we can define

$$A = \frac{a}{\delta x}\begin{pmatrix} 1 & & & & \\ -1 & 1 & & & \\ & \ddots & \ddots & & \\ & & -1 & 1 & \\ & & & -1 & 1 \end{pmatrix}, \quad B = \frac{\epsilon}{\delta x^2}\begin{pmatrix} -2 & 1 & & & \\ 1 & -2 & 1 & & \\ & \ddots & \ddots & \ddots & \\ & & 1 & -2 & 1 \\ & & & 1 & -2 \end{pmatrix}.$$

Setting $\Lambda = A + B$ then let's us express the above system of equations as

$u'(t) = \Lambda u(t) + g(t) \quad$ where $\quad g(t) = \left[(\frac{\epsilon}{\delta x^2} - \frac{a}{\delta x})d_0(t) + F_1, F_2, ..., F_{m-2}, \frac{\epsilon}{\delta x^2}d_1(t) + F_{m-1}\right]^T$

$u(0) = u_0 \quad$ with $\quad u_0 = [c(x_1), ..., c(x_{m-1})]^T$

$u(t) = [u_1(t), ..., u_{m-1}(t)]^T$

Our system is of the form $\dot{u} = Au + g$. Now applying backward Euler's method with time-step size δt gives us

$V_{n+1} = V_n + \delta t[\Lambda V_{n+1} + g_{n+1}]$

$V_{n+1} = V_n + \delta t \Lambda V_{n+1} + hg_{n+1}$

$(I - \delta t \Lambda)V_{n+1} = V_n + hg_{n+1}$

$V_{n+1} = (I - \delta t \Lambda)^{-1}(V_n + hg_{n+1})$

When using backward Euler's method, we must invert the matrix $A = (I - \delta t \Lambda)$.

Since the matrix is tridiagonal and the inverse is dense, we would want to use an iterative method to save time. For example, a tridiagonal LU bidiagonal factorization could be used which is $\mathcal{O}(n)$.

Appendix B

Forward Euler Discretization

$$\begin{cases} u_t - au_x = \epsilon u_{xx} + F, \\ u(0,t) = d_0(t), \quad u(1,t) = d_1(t), \\ u(x,0) = c(x). \end{cases} \tag{B.1}$$

Our discretization process is the same as in (APPENDIX A) up until we get $u'(t) = \Lambda u(t) + g(t)$ where

$$A = \frac{a}{\delta x}\begin{pmatrix} 1 & & & & & \\ -1 & 1 & & & & \\ & \ddots & \ddots & & & \\ & & & -1 & 1 & \\ & & & & -1 & 1 \end{pmatrix}, \quad B = \frac{\epsilon}{\delta x^2}\begin{pmatrix} -2 & 1 & & & & \\ 1 & -2 & 1 & & & \\ & \ddots & \ddots & \ddots & & \\ & & 1 & -2 & 1 & \\ & & & 1 & -2 \end{pmatrix},$$

$\Lambda = A + B,$

$g(t) = \left[(\frac{\epsilon}{\delta x^2} - \frac{a}{\delta x})d_0(t) + F_1, F_2, ..., F_{m-2}, \frac{\epsilon}{\delta x^2}d_1(t) + F_{m-1}\right]^T,$

$u(0) = u_0 \quad \text{with} \quad u_0 = [c(x_1), ..., c(x_{m-1})]^T,$

$u(t) = [u_1(t), ..., u_{m-1}(t)]^T$.

Our system is of the form $\dot{u} = Au + g$. Now applying forward Euler's method with timestep size δt gives us

$V_{n+1} = V_n + \delta t[\Lambda V_n + g_n]$

This system can be solved straight forwardly without the need to invert a system. We do, however, have to consider the stability of this method as it is an explicit method. Notice that we have $\delta t\Lambda$. If we expand these terms out a little, we get the terms used in the CFL conditions for advection and diffusion. The advection term is $a\frac{\delta t}{\delta x}$ and the diffusion term is $\epsilon\frac{\delta t}{\delta x^2}$. The CFL conditions are given in equation (4.4).