VALIDATION AND ANALYSIS OF NUMERICAL METHODS FOR SOLVING FRACTIONAL-ORDER DIFFERENTIAL EQUATIONS

A Thesis

by

CONNOR LAWRENCE MITCHELL

Submitted to the Office of Graduate and Professional Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Chair of Committee,	Alan Freed
Committee Members,	Anastasia Muliana
	Raytcho Lazarov
Head of Department,	Andreas Polycarpu

May 2018

Major Subject: Mechanical Engineering

ABSTRACT

Fractional-order differential equations have been very effective in modeling the behavior of various phenomena that have been otherwise difficult to accurately simulate. A new class of predictor-corrector schemes for solving nonlinear fractional-order differential equations has been proposed with claims of both increased accuracy and decreased computational cost. The purpose of this research is to test and attempt to validate these claims through both an independent replication of the results reported for the new method, as well as a comparison against the performance of an existing standard. Additionally, an operational software package that implements this new method will result as a byproduct of this research.

Independent simulations are run for this new method, and compared against the findings of this new method's author. This research is broken down into three main areas of interest: (1) Independently develop software for the newly proposed and existing methods, (2) Conduct an error analysis on the methods' order of accuracy, and (3) Conduct a computational cost analysis of the two methods in their pure form. This research has successfully created a tool for solving fractional-order differential equations with greater accuracy than the existing method, with replicated findings in regard to order of accuracy. The claims of decreased computational cost were neither confirmed or disproved, and require further study. The results of this research have not only increased the validity and usage-case understanding of the newly proposed method, but also raised new areas for future work with respect to optimization and run speed.

DEDICATION

For my family, who have shaped me into the person I am, support me in every endeavor, and will always be a part of my heart. Thank you Don, Fay, Mark, Susan, and Taylor.

ACKNOWLEDGMENTS

I would like to extend a special thank you to my advisor, Dr. Alan Freed. As a former professor of mine, your passion and enthusiasm for teaching speaks volumes to your character, and your courses inspired my career path. As an advisor, you were always available and eager to help guide me in research or otherwise, and you taught me the value of thought. Getting to work with you as a student, teaching assistant, and researcher has been an absolute pleasure, and I want to thank you for all of your time and support.

I would also like to thank Dr. Muliana and Dr. Lazarov for serving on my thesis committee, dedicating their time to me, and supporting me throughout my degree program.

CONTRIBUTORS AND FUNDING SOURCES

Contributors

This work was supported by a thesis committee consisting of Dr. Alan Freed and Dr. Anastasia Muliana of the Department of Mechanical Engineering and Dr. Raytcho Lazarov of the Department of Mathematics. The data and mathematical models analyzed for this thesis were provided by Dr. Freed. All other work conducted for the thesis was completed by the student independently.

Funding Sources

This thesis was partially funded by the Texas A&M University Engineering Experiment Station (TEES).

NOMENCLATURE

ABM	Adams-Bashforth-Moulton
FDE	Fractional Differential Equation
FFT	Fast Fourier Transform
MF-PCL	(Acronym undefined by [1]) Second-order scheme with linear interpolation
MF-PCQ	(Acronym undefined by [1]) Third-order scheme with quadratic interpolation
ODE	Ordinary Differential Equation
PC	Predictor Corrector
PECE	Predict Evaluate Correct Evaluate
STEM	Science, Technology, Engineering, and Mathematics
TEES	Texas A&M University Engineering Experiment Station

TABLE OF CONTENTS

Page

LIST OF FIGURES

FIGURE Page

LIST OF TABLES

TABLE Page

1. INTRODUCTION AND LITERATURE REVIEW

1.1 Introduction

With any new development in the realm of STEM, a very integral part of the scientific process is the act of replication. Successful replication of the findings of an author helps to validate that the findings are in fact accurate and meaningful, and is a critical step in the advancement of scientific knowledge. Note here that the linguistic choice of replication was not arbitrarily selected; it is chosen and differentiated from the term 'reproduction'. Replication insists that the method is independently tested, without influence from the original author. This eliminates biases on experimentation, and therefore can truly validate/debunk the scientific discovery or advancement that is being claimed. Reproduction, in contrast, would imply that the same data/results are analyzed and found to be in agreement by an independent group. More specifically, reproducibility could suggest that the same software used by [1] was also used in this research, which is why the distinction is made here. One additional distinction that should be made before continuing is the definition of a method and a scheme in the context of this document. A method hierarchically envelops specific schemes of a certain method. For example, this research investigates two methods: the Adams-Bashforth-Moulton method, and the new method proposed by [1]. However, this new method contains two different implementations with regard to the interpolator that is used, which are referred to as separate schemes under this single method.

In this study, the published journal article by Nguyen and Jang [1] will be studied and the method proposed within will be recreated independently. No software by the authors of this method, nor any other third-party distribution is provided or referenced. Using the schemes proposed within [1], this research will produce a software package that is capable of computing solutions using the proposed method. These independently simulated results will then undergo the same analysis and post-processing as presented in [1], and will be compared against each other to

determine if the method has, in fact, been replicated.

In order to attempt to replicate the findings of [1], this research has investigated the three objectives discussed briefly in the abstract, and in further detail later. A software package was developed that is capable of solving fractional differential equations, and post-processing of the simulation data was performed with respect to both error and computational cost of the method. The results of this study have replicated the method to a high degree of accuracy, and have been able to at least partially validate the claims made by [1]. The specific findings and claims will be discussed in more detail later. In addition to these findings, this study has also created the need for further research with respect to order of accuracy for inhomogeneous initial conditions, improved numerical integration methods for a specific class of integrands over diminutive intervals, and optimization techniques for decreased computational cost.

1.2 Motivation

The primary motivation for this study with respect to application is rooted in the simulation of mechanics of material deformation. The behavior and response of certain classes of materials, especially in the classes of viscoelasticity and viscoplasticity, have been exceptionally difficult to accurately model using ordinary differential equations. One strategy to increase the fidelity of these difficult-to-model behaviors is to use a heredity term, or a so-called memory effect relating to the material in question. In other words, the prediction of the state of the material at the current time in question is influenced by its state at all previous times. One effective approach to include this heredity is to model the material behavior using fractional-order differential equations (FDE's), which will be discussed in further detail later.

The inclusion of this material heredity mechanism in the model, however, comes with a greatly increased computational cost. Many previous studies into specific applications using FDE's have failed due to computational expense issues. Therefore, a new method that decreases the computa-

tional expense of the memory effect would be a very useful advancement.

1.3 Fractional-Order Differential Equations

In order to discuss these new methods, let us first give an overview of what is meant by a fractional-order differential equation. An abbreviated explanation of some concepts used in this research are recalled below. For a more detailed and complete background into fractional calculus, see [2]. The following discussion and derivation will follow closely to the introduction of [1], as the methods proposed in this article are the primary focus of the research. The fractional derivative of a function is defined below as

$$D_0^\alpha y(t) = f(t, y(t)), \quad t = [0, T] \tag{1.1}$$

$$y^{(k)}(0) = y_k, \quad k = 0, 1, ..., m-1, \tag{1.2}$$

where $m-1 < \alpha \leq m \in Z^+$. The fractional derivative expressed above may be defined in several different forms; however the form used moving forward will be the Caputo derivative ([3]), which is defined below as

$$D_0^\alpha y(t) = \frac{1}{\Gamma(m-\alpha)} \int_0^t (t-\tau)^{m-1-\alpha} y^m(\tau) d\tau \tag{1.3}$$

This choice is made because of the convenience in establishing the initial conditions with relative ease when using the Caputo derivative. The initial conditions of the Caputo derivative are equivalent to those of integer-order derivatives, which are much easier to establish, and have a physical significance. The Riemann-Liouville derivative, which is not used here, is much more useful for applications in the study of wave-based phenomena. This is due to the initial conditions of the wave not being of particular interest, but rather, the wave at some point along its path. The initial conditions of the Riemann-Liouville derivative are defined in terms of initial values of fractional derivatives, as opposed to the integer-order values as mentioned with the Caputo derivative. A more detailed description of these different forms can be found in [4].

A continuous function $y(t)$ is a solution to Eq. (1.1) if and only if it is a solution to the Volterra integral equation ([5])

$$y(t) = g(t) + \frac{1}{\Gamma(\alpha)} \int_0^t (t-\tau)^{\alpha-1} f(\tau, y(\tau)) d\tau, \tag{1.4}$$

where the initial condition, $g(t)$ is defined as

$$g(t) = \sum_{k=0}^{m-1} y_k \frac{t^k}{k!}. \tag{1.5}$$

A discretized grid over which a solution is sought is constructed as

$$\Phi_N := \{t_j : 0 = t_0 < t_1, ... < t_j < ... < t_n < t_{n+1} < t_N = T\}. \tag{1.6}$$

Assuming this grid is uniform, i.e. $h = t_{j+1} - t_j, \forall j = 0, ..., N-1$, a solution can be expressed as a set of three terms, seen below by rewriting Eq. (1.4) as ([6]),

$$y(t_{n+1}) = g(t_{n+1}) + y^*(t_{n+1}) + Y(t_{n+1}), \tag{1.7}$$

where $y^*(t_{n+1}) = y^*_{n+1}$ quantifies a lag term, and $Y(t_{n+1}) = Y_{n+1}$ is defined as the increment term. The lag term is the source of heredity, or memory effect, in fractional-order differential equations. This lag term, described by

$$y^*(t_{n+1}) = \frac{1}{\Gamma(\alpha)} \int_0^{t_n} (t_{n+1} - \tau)^{\alpha-1} f(\tau, y(\tau)) d\tau \tag{1.8}$$

requires information from all prior time steps, from 0 to the current time t_n, in order to calculate the $y(t_{n+1})$ term. The increment term, described by

$$Y(t_{n+1}) = \frac{1}{\Gamma(\alpha)} \int_{t_n}^{t_{n+1}} (t_{n+1} - \tau)^{\alpha-1} f(\tau, y(\tau)) d\tau, \tag{1.9}$$

is responsible for advancing solution to the next increment, as the name implies.

All of the methods being analyzed in this study involve the same basic "Predict, Evaluate, Correct, Evaluate" (PECE) operating principle. That is, an estimated value y_{n+1}^P is predicted, and used to calculate the increment term to progress. This is followed by calculation of the corrector term, which is the approximation of the solution, using the notation $\tilde{y}_{n+1} \approx y_{n+1}$.

1.4 Adams-Bashforth-Moulton (ABM) Method

The Adams-Bashforth-Moulton method ([7]) is re-derived below in order to highlight the components of the method, and to aid in setting up a comparison between the methods presented in this study. This method is the existing standard that will be used as a comparison and benchmark for the new schemes proposed in [1]. Equation (1.4) can be rewritten as

$$\begin{aligned} y(t_{n+1}) = g(t_{n+1}) + \frac{1}{\Gamma(\alpha)} \sum_{j=0}^{n-1} \int_{t_j}^{t_{j+1}} (t_{n+1} - \tau)^{\alpha-1} f(\tau, y(\tau)) \\ + \frac{1}{\Gamma(\alpha)} \int_{t_n}^{t_{n+1}} (t_{n+1} - \tau)^{\alpha-1} f(\tau, y(\tau)) d\tau. \end{aligned} \tag{1.10}$$

First, the predictor value, $\tilde{y}_{n+1}^P$ must be developed. A low-order approximation is used, obtained using a constant interpolation over each interval, $I_j = [t_j, t_{j+1}]$,

$$f(t) \approx f(t_j)\chi_{I_j}(t), \tag{1.11}$$

where

$$\chi_{I_j}(t) = \begin{cases} 1, & \text{if } t \in I_j, \\ 0, & \text{otherwise.} \end{cases} \tag{1.12}$$

Combining Eq. (1.11) with Eq. (1.10), the predictor equation is constructed:

$$\tilde{y}_{n+1}^{P} = g_{n+1} + \frac{1}{\Gamma(\alpha)} \sum_{j=0}^{n} C_{n+1}^{j} \tilde{f}_j, \tag{1.13}$$

where the coefficient is defined as

$$C_{n+1}^{j} = \int_{t_j}^{t_{j+1}} (t_{n+1} - \tau)^{\alpha-1} d\tau = \frac{h^\alpha}{\alpha}[(n+1-j)^\alpha - (n-j)^\alpha]. \tag{1.14}$$

Next, the corrector, $\tilde{y}_{n+1}$ must be developed. Using a linear Lagrange interpolation function, $f(t)$ is interpolated over each interval, $t \in I_j$, such that

$$f(t) \approx f_j L_j(t) + f_{j+1} L_{j+1}(t), \quad t \in I_j, \tag{1.15}$$

where

$$L_k(t) = \frac{t - t_l}{t_k - t_l}, \ (k,l) = (j, j+1), (j+1, j). \tag{1.16}$$

Combining Eq. (1.10) and Eq. (1.15), the equation for the corrector can be rewritten as

$$\tilde{y}_{n+1} = g_{n+1} + \frac{1}{\Gamma(\alpha)} \sum_{j=0}^{n-1} [B_{n+1}^{0,j} \tilde{f}_j + B_{n+1}^{1,j} \tilde{f}_{j+1}] + \frac{1}{\Gamma(\alpha)} [B_{n+1}^{0,n} \tilde{f}_n + B_{n+1}^{1,n} \tilde{f}_{n+1}^{P}], \tag{1.17}$$

with the coefficients, which can be evaluated explicitly, being defined as

$$\begin{aligned} B_{n+1}^{0,j} &= \int_{t_j}^{t_{j+1}} (t_{n+1} - \tau)^{\alpha-1} L_j(\tau) d\tau, \\ B_{n+1}^{1,j} &= \int_{t_j}^{t_{j+1}} (t_{n+1} - \tau)^{\alpha-1} L_{j+1}(\tau) d\tau. \end{aligned} \tag{1.18}$$

1.5 New Proposed Method by Nguyen and Jang

The following development summarizes the new method proposed by Nguyen and Jang in [1]. Both the second-order scheme, with a linear interpolator (herein referred to as MF-PCL), and the

third-order scheme, with a quadratic interpolator (herein referred to as MF-PCQ), will be displayed below.

1.5.1 Second-Order Scheme with a Linear Interpolator (MF-PCL)

The construction of the newly proposed second-order scheme begins with the following lemma.

Lemma 1.1 *Assume $\phi(t) \in P_1[0,T]$ where P_1 is the space of all polynomials of degree less than or equal to one. Let $\phi_n, n = 0, ...N$, be the restricted value of $\phi(t)$ on grid Φ_N defined in Eq. (1.6). Then, there exist coefficients b^0_{n+1} and b^1_{n+1} such that the following equality*

$$\int_{t_n}^{t_{n+1}} (t_{n+1} - \tau)^{\alpha-1} \phi(\tau) d\tau = \frac{h^\alpha}{\alpha(\alpha+1)} (b^0_{n+1}\phi_{n-1} + b^1_{n+1}\phi_n), \tag{1.19}$$

is exact. More precisely,

$$b^0_{n+1} = -1, \quad b^1_{n+1} = \alpha + 2; \tag{1.20}$$

The proof for this is shown in full detail in [1]. The new scheme is now defined below, where the predictor formula is defined as

$$\tilde{y}^P_{n+1} = g_{n+1} + \tilde{y}^*_{n+1} + \frac{h}{\Gamma(\alpha+2)} [b^0_{n+1}\tilde{f}_{n-1} + b^1_{n+1}\tilde{f}_n], \tag{1.21}$$

and the corrector formula is defined as

$$\tilde{y}_{n+1} = g_{n+1} + \tilde{y}^*_{n+1} + \tilde{Y}_{n+1}. \tag{1.22}$$

These equations consist of the lag term, which appears in both the predictor and corrector equations, defined as

$$\tilde{y}^*_{n+1} = \frac{1}{\Gamma(\alpha)} \sum_{j=0}^{n-1} [B^{0,j}_{n+1}\tilde{f}_j + B^{1,j}_{n+1}\tilde{f}_{j+1}], \tag{1.23}$$

the increment term, defined as

$$\tilde{Y}_{n+1} = \frac{1}{\Gamma(\alpha)}[B_{n+1}^{0,n}\tilde{f}_n + B_{n+1}^{1,n}\tilde{f}_{n+1}^{P}], \tag{1.24}$$

and the initial condition, g_{n+1}, as defined in Eq. (1.5).

1.5.2 Third-Order Scheme with a Quadratic Interpolator (MF-PCQ)

The construction of the newly proposed third-order scheme begins with the following lemma.

Lemma 1.2 *Assume $\phi(t) \in P_2[0,T]$ where P_2 is the space of all polynomials of degree less than or equal to two. Let $\phi_n, n = 0, ...N$, be the restricted value of $\phi(t)$ on grid Φ_N defined in Eq. (1.6). Then, there exist coefficients a_{n+1}^0, a_{n+1}^1, and a_{n+1}^2 such that the following equality*

$$\int_{t_n}^{t_{n+1}} (t_{n+1} - \tau)^{\alpha-1}\phi(\tau)d\tau = A(a_{n+1}^0\phi_{n-2} + a_{n+1}^1\phi_{n-1} + a_{n+1}^2\phi_n) \tag{1.25}$$

is exact, where

$$A = \frac{h^\alpha}{\alpha(\alpha+1)(\alpha+2)}, \tag{1.26}$$

and

$$\begin{aligned} a_{n+1}^0 &= \frac{\alpha+4}{2}, \\ a_{n+1}^1 &= -2(\alpha+3), \\ a_{n+1}^2 &= \frac{2\alpha^2+9\alpha+12}{2}. \end{aligned} \tag{1.27}$$

Using a quadratic Lagrange polynomial, $f(t)$ is interpolated over each interval $[t_{j-1}, t_{j+1}], j \geq 1$, such that

$$f(t) \approx \sum_{k=j-1}^{j+1} f_k Q_k^j(t), \tag{1.28}$$

where

$$Q_k^j(t) = \prod_{\substack{m=j-1 \\ m\neq k}}^{j+1} \frac{t - t_m}{t_k - t_m}. \tag{1.29}$$

On the first interval $[t_0, t_1], j = 0$, so t_{j-1} is not defined. Hence, $f(t)$ can be interpolated here using

$$f(t) \approx f_0 Q_0^0(t) + f_{\frac{1}{2}} Q_{\frac{1}{2}}^0(t) + f_1 Q_1^0(t), \tag{1.30}$$

where [1]

$$\begin{aligned} Q_0^0(t) &= \frac{(t - t_{\frac{1}{2}})(t - t_1)}{(t_0 - t_{\frac{1}{2}})(t_0 - t_1)}, \\ Q_{\frac{1}{2}}^0(t) &= \frac{(t - t_0)(t - t_1)}{(t_{\frac{1}{2}} - t_0)(t_{\frac{1}{2}} - t_1)}, \\ Q_1^0(t) &= \frac{(t - t_0)(t - t_{\frac{1}{2}})}{(t_1 - t_0)(t_1 - t_{\frac{1}{2}})}. \end{aligned} \tag{1.31}$$

Combining Eq. (1.28) and Eq. (1.10), the new scheme is now defined below, where the predictor formula is defined as

$$\tilde{y}_{n+1}^P = g_{n+1} + \tilde{y}_{n+1}^* + \frac{h^\alpha}{\Gamma(\alpha + 3)}[a_{n+1}^0 \tilde{f}_{n-2} + a_{n+1}^1 \tilde{f}_{n-1} + a_{n+1}^2 \tilde{f}_n], \tag{1.32}$$

and the corrector formula is defined as

$$\tilde{y}_{n+1} = g_{n+1} + \tilde{y}_{n+1}^* + \tilde{Y}_{n+1}. \tag{1.33}$$

This corrector is the of the same form as the second-order scheme, and likewise uses the same lag term as in the predictor. The components of these equations consist of the lag term, defined as

$$\begin{aligned} \tilde{y}_{n+1}^* = &\frac{1}{\Gamma(\alpha)}[A_{n+1}^{0,0} \tilde{f}_0 + A_{n+1}^{1,0} \tilde{f}_{\frac{1}{2}} + A_{n+1}^{2,0} \tilde{f}_1] \\ &+ \frac{1}{\Gamma(\alpha)} \sum_{j=1}^{n-1} [A_{n+1}^{0,j} \tilde{f}_{j-1} + A_{n+1}^{1,j} \tilde{f}_j + A_{n+1}^{2,j} \tilde{f}_{j+1}], \end{aligned} \tag{1.34}$$

[1] The equation for $Q_{\frac{1}{2}}^0(t)$ is incorrectly reported in [1]. The correct form is reported in Eq. (1.31).

where

$$
\begin{aligned}
A_{n+1}^{0,0} &= \int_0^{t_1} (t_{n+1} - \tau)^{\alpha-1} Q_0^0(\tau) d\tau, \\
A_{n+1}^{1,0} &= \int_0^{t_1} (t_{n+1} - \tau)^{\alpha-1} Q_{\frac{1}{2}}^0(\tau) d\tau, \\
A_{n+1}^{2,0} &= \int_0^{t_1} (t_{n+1} - \tau)^{\alpha-1} Q_1^0(\tau) d\tau,
\end{aligned}
\tag{1.35}
$$

the increment term, defined as

$$
\tilde{Y}_{n+1} = \frac{1}{\Gamma(\alpha)} [A_{n+1}^{0,n} \tilde{f}_{n-1} + A_{n+1}^{1,n} \tilde{f}_n + A_{n+1}^{2,n} \tilde{f}_{n+1}^P], \tag{1.36}
$$

and the initial condition, g_{n+1}, as defined in Eq. (1.5).

2. RESEARCH OBJECTIVES

To guide this research, the following objectives were identified and explored, broken down into three major areas of interest. Each of these three objectives will be discussed in further detail regarding both the methods of testing and the results.

1. Independently develop software for the proposed method and the existing standard.
2. Conduct an error analysis on the methods' order of accuracy using simulated test cases.
3. Conduct a computational cost analysis of the methods.

2.1 Independent Software Development and Simulation

The first step towards running a meaningful analysis of the methods in question requires creating a functional software tool. In order for this validation to be meaningful, the numerical methods should be independently written and simulated, using only the derived equations present in [1]. This research objective is a prerequisite to the remaining two objectives, as functional numerical methods must be constructed in order to collect meaningful data for error analysis and computational cost comparisons.

The numerical methods under analysis are written using MATLAB®. There also exists a third-party MATLAB® function [8] for the Adams-Bashforth-Moulton method that will be used for increased fidelity in simulation testing. The completion of this research objective allows for future research of applications in the area of fractional-order differential equations, with a well-documented, easy to operate user interface.

2.2 Error Analysis

The second research objective, error analysis, explores the accuracy of each of the numerical methods proposed. This objective is the major focus of this research, and is necessary for the com-

pletion of the first objective. The results from this objective yield important information regarding simulation fidelity. The focus here will be not necessarily on performance of the methods on a case-by-case basis, but rather on the order of error across several classes of equations at different values of α, in both magnitudes of $0 < \alpha \leq 1$ and $\alpha > 1$.

Jang and Nguyen state that the global error E_G for their proposed schemes are on the order of $\mathcal{O}(h^2)$ for the 2^{nd} order scheme with linear interpolation, and $\mathcal{O}(h^3)$ for the 3^{rd} order scheme with quadratic interpolation ([1]), regardless of the order of α. This would be a significant improvement over the Adams-Bashforth-Moulton method for cases where $0 < \alpha < 1$, which has been shown in [9] to be on the order of $\mathcal{O}(h^p)$, where $p = min(1 + \alpha, 2)$.

2.3 Computational Cost Analysis

The third and final major focus of this research will involve computational cost of the numerical methods. While increased accuracy is typically a welcomed advancement, it can oftentimes stymie the adoption of new numerical methods if it comes with an increased computational cost. However, the method presented by Nguyen and Jang claims to lower the computational cost while also decreasing (or maintaining for $\alpha \geq 1$) the order the error.

3. METHODS

The following chapter will discuss a detailed approach to completing each of the previously expressed research objectives.

3.1 Independent Software Development and Simulation

3.1.1 Independent Software Development

The software development portion of this objective will largely require converting mathematical equations into numerical methods in a programming language. As mentioned, MATLAB® was the chosen programming language for developing these methods.

Computation of integral functions will be achieved using the MATLAB® adaptive quadrature function, *integral()*. No independent integrator function will be developed for this research. The functions are of the following form:

$$[T, Y] = MF\_PCL(alpha, fdefun, t0, tfinal, y0, h), \tag{3.1}$$

$$[T, Y] = MF\_PCQ(alpha, fdefun, t0, tfinal, y0, h), \tag{3.2}$$

$$[T, Y] = ABM(alpha, fdefun, t0, tfinal, y0, h) \tag{3.3}$$

where Eq. (3.1) corresponds to the 2^{nd} order scheme with linear interpolation, and Eq. (3.2) corresponds to the 3^{rd} order scheme with quadratic interpolation. Equation (3.3) refers to the existing method being compared against, the Adams-Bashforth-Moulton method.

As mentioned previously, there exists a preexisting third-party MATLAB® function for the

ABM method, titled fde12, which has the following user interface:

$$[T, Y] = fde12(alpha, fdefun, t0, tfinal, y0, h). \tag{3.4}$$

All methods are designed to follow the same interface, allowing the user to easily switch between methods.

Function Inputs

- **alpha:** The order of the differential equation. Alpha is any positive number $\alpha \in R^+$
- **fdefun:** The fractional-order differential equation. fdefun should be a function handle, and should be a function of y and t. The notation for this in MATLAB® is @(y,t) followed by the equation.
- **t0:** The initial time. *t0* should be a scalar value.
- **tfinal:** The final time. *tfinal* should be a scalar number > *t0*.
- **y0:** Initial conditions. *y0* is a matrix with rows of length equal to the size of the problem, and columns equal to m, which is defined above in the discussion following Eq. (1.1) and Eq. (1.2).
- **h:** The step size of the problem. *h* must be a scalar value > 0. The step size remains constant over the entire simulation.

Function Outputs

- **Y:** Solution to the FDE. **Y** should be a matrix matching in column size to the *y0* input, and row size according to the number of steps taken.
- **T:** Time. **T** should be a vector of times ranging from the start time, *t0*, to the final time, *tfinal*, incremented by the step size, *h*.

Supporting Files

The software package consists of a main function file for each numerical method as defined above, as well as all of the supporting functions saved in separate function files. The supporting files in the software package include:

- Linear Lagrange Interpolation Function: $(Linear\_Interp.m)$
- Quadratic Lagrange Interpolation Function: $(Quadratic\_Interp.m)$
- Initial Condition Calculator: $(Calculate\_IC.m)$
- 2^{nd} Order Coefficient Calculator: $(B\_Coefficents.m)$
- 3^{rd} Order Coefficient Calculator: $(A\_Coefficents.m)$
- Start-up Algorithm: $(StartupProcedure.m)$
- Simulation and Usage-Example Test Script: $(MF\_TestScript.m)$

Additionally, while they not part of the methods, the following two files included in this research are used in conducting an error analysis on anomalous behavior in certain specific test cases. This will be discussed in further detail in the results of this thesis.

- 2^{nd} Order Analytical Coefficient Calculator: $(B\_Coefficents\_Analytical.m)$
- 3^{rd} Order Analytical Coefficient Calculator: $(A\_Coefficents\_Analytical.m)$

The MATLAB® source code for these functions are included in Appendix B.

Start-up Algorithm

In order to find a desired accuracy for $\tilde{y}_1$, $\tilde{y}_2$, [1] uses a start-up algorithm that utilizes a combination of constant, linear, and quadratic interpolation to approximate values at $t_{h/4}$, $t_{h/2}$, t_h, and t_{2h}. This start-up grid is shown below in Fig. 3.1. The start-up algorithm is reproduced here from [1]. Note here that the nomenclature used for $t_0, t_{\frac{1}{4}}$, etc. is defined in Fig. 3.1. The subscripts

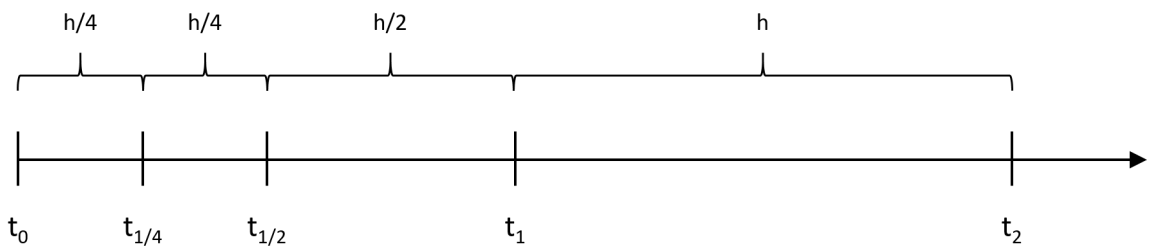

Figure 3.1: Start-up grid for the new proposed method from [1].

are in reference to the starting time, t_0, and the step-size, h. For example, $\tilde{y}_{\frac{1}{4}}$ is described as the approximate solution to the FDE at time $t_0 + \frac{h}{4}$. Similarly, $g_{\frac{1}{4}}$ is the initial condition term evaluated at this same time step, which is defined previously by Eq. (1.5). The following two equations will be used frequently, and are defined below:

$$\hat{A}^{c,d,e}_{n,a,b} = \int_{t_a}^{t_b} (t_n - \tau)^{\alpha-1} \Big(\frac{\tau - t_d}{t_c - t_d}\Big)\Big(\frac{\tau - t_e}{t_c - t_e}\Big) d\tau$$
$$\hat{B}^{c,d}_{n,a,b} = \int_{t_a}^{t_b} (t_n - \tau)^{\alpha-1} \Big(\frac{\tau - t_d}{t_c - t_d}\Big) d\tau$$

1. Approximate $\tilde{y}_{\frac{1}{4}}$:

   - Predict $\tilde{y}^P_{\frac{1}{4}}$: (constant interpolation)

$$\tilde{y}^P_{\frac{1}{4}} = g_{\frac{1}{4}} + \frac{1}{\Gamma(\alpha+1)} \Big(\frac{h}{4}\Big)^{\alpha} \tilde{f}_0$$

2. Correct $\tilde{y}_{\frac{1}{4}}$: (linear interpolation)

$$\tilde{y}_{\frac{1}{4}} = g_{\frac{1}{4}} + \frac{1}{\Gamma(\alpha)} (\hat{B}^{0,\frac{1}{4}}_{\frac{1}{4},0,\frac{1}{4}} \tilde{f}_0 + \hat{B}^{\frac{1}{4},0}_{\frac{1}{4},0,\frac{1}{4}} \tilde{f}^P_{\frac{1}{4}})$$

3. Approximate $\tilde{y}_{\frac{1}{2}}$:

$$\tilde{y}^*_{\frac{1}{2}} = \frac{1}{\Gamma(\alpha)} (\hat{B}^{0,\frac{1}{4}}_{\frac{1}{2},0,\frac{1}{4}} \tilde{f}_0 + \hat{B}^{\frac{1}{4},0}_{\frac{1}{2},0,\frac{1}{4}} \tilde{f}_{\frac{1}{4}})$$

- Predict $\tilde{y}_{\frac{1}{2}}^{P_1}$: (constant interpolation)

$$\tilde{y}_{\frac{1}{2}}^{P_1} = g_{\frac{1}{2}} + \tilde{y}_{\frac{1}{2}}^{*} + \frac{1}{\Gamma(\alpha+1)}\Big(\frac{h}{4}\Big)^{\alpha}\tilde{f}_{\frac{1}{4}}$$

- Predict $\tilde{y}_{\frac{1}{2}}^{P_2}$: (linear interpolation)

$$\tilde{y}_{\frac{1}{2}}^{P_2} = g_{\frac{1}{2}} + \tilde{y}_{\frac{1}{2}}^{*} + \frac{1}{\Gamma(\alpha)}(\hat{B}_{\frac{1}{2},\frac{1}{4},\frac{1}{2}}^{\frac{1}{4},\frac{1}{2}}\tilde{f}_{\frac{1}{4}} + \hat{B}_{\frac{1}{2},\frac{1}{4},\frac{1}{2}}^{\frac{1}{2},\frac{1}{4}}\tilde{f}_{\frac{1}{2}}^{P_1})$$

- Correct $\tilde{y}_{\frac{1}{2}}$: (quadratic interpolation)

$$\tilde{y}_{\frac{1}{2}} = g_{\frac{1}{2}} + \frac{1}{\Gamma(\alpha)}(\hat{A}_{\frac{1}{2},0,\frac{1}{2}}^{0,\frac{1}{4},\frac{1}{2}}\tilde{f}_0 + \hat{A}_{\frac{1}{2},0,\frac{1}{2}}^{\frac{1}{4},0,\frac{1}{2}}\tilde{f}_{\frac{1}{4}} + \hat{A}_{\frac{1}{2},0,\frac{1}{2}}^{\frac{1}{2},0,\frac{1}{4}}\tilde{f}_{\frac{1}{2}}^{P_2})$$

4. Approximate $\tilde{y}_1$:

$$\tilde{y}_1^{*} = \frac{1}{\Gamma(\alpha)}(\hat{B}_{1,0,\frac{1}{2}}^{0,\frac{1}{2}}\tilde{f}_0 + \hat{B}_{1,0,\frac{1}{2}}^{\frac{1}{2},0}\tilde{f}_{\frac{1}{2}})$$

- Predict $\tilde{y}_1^{P_1}$: (constant interpolation)

$$\tilde{y}_1^{P_1} = g_1 + \tilde{y}_1^{*} + \frac{1}{\Gamma(\alpha+1)}\Big(\frac{h}{2}\Big)^{\alpha}\tilde{f}_{\frac{1}{2}}$$

- Predict $\tilde{y}_1^{P_2}$: (linear interpolation)

$$\tilde{y}_1^{P_2} = g_1 + \tilde{y}_1^{*} + \frac{1}{\Gamma(\alpha)}(\hat{B}_{1,\frac{1}{2},1}^{\frac{1}{2},1}\tilde{f}_{\frac{1}{2}} + \hat{B}_{1,\frac{1}{2},1}^{1,\frac{1}{2}}\tilde{f}_1^{P_1})$$

- Correct $\tilde{y}_1$: (quadratic interpolation)

$$\tilde{y}_1 = g_1 + \frac{1}{\Gamma(\alpha)}(\hat{A}_{1,0,1}^{0,\frac{1}{2},1}\tilde{f}_0 + \hat{A}_{1,0,1}^{\frac{1}{2},0,1}\tilde{f}_{\frac{1}{2}} + \hat{A}_{1,0,1}^{1,0,\frac{1}{2}}\tilde{f}_1^{P_2})$$

5. Approximate $\tilde{y}_2$:

$$\tilde{y}_2^{*} = \frac{1}{\Gamma(\alpha)}(\hat{A}_{2,0,1}^{0,\frac{1}{2},1}\tilde{f}_0 + \hat{A}_{2,0,1}^{\frac{1}{2},0,1}\tilde{f}_{\frac{1}{2}} + \hat{A}_{2,0,1}^{1,0,\frac{1}{2}}\tilde{f}_1)$$

- Predict $\tilde{y}_2^{P_1}$: (constant interpolation)

$$\tilde{y}_2^{P_1} = g_2 + \tilde{y}_2^* + \frac{h^\alpha}{\Gamma(\alpha+1)}\tilde{f}_1$$

- Predict $\tilde{y}_2^{P_2}$: (linear interpolation)

$$\tilde{y}_2^{P_2} = g_2 + \tilde{y}_2^* + \frac{1}{\Gamma(\alpha)}(\hat{B}_{2,1,2}^{1,2}\tilde{f}_1 + \hat{B}_{2,1,2}^{2,1}\tilde{f}_2^{P_1})$$

- Correct $\tilde{y}_2$: (quadratic interpolation)

$$\tilde{y}_2 = g_2 + \frac{1}{\Gamma(\alpha)}(\hat{A}_{2,1,2}^{0,1,2}\tilde{f}_0 + \hat{A}_{2,1,2}^{1,0,2}\tilde{f}_1 + \hat{A}_{2,1,2}^{2,0,1}\tilde{f}_2^{P_2})$$

3.1.2 Simulation

Upon completion of construction and validation of the software tools, simulations will be run using an assortment of fractional differential equations. Results from each of the numerical methods will be compared against each other as well as the FDEs' known solutions. The four test cases that were used in [1] include the following:

Sample Equation 1

$$D_0^\alpha y(t) = \frac{40320}{\Gamma(9-\alpha)}t^{8-\alpha} - 3\frac{\Gamma(5+\frac{\alpha}{2})}{\Gamma(5-\frac{\alpha}{2})}t^{4-\frac{\alpha}{2}} + \frac{9}{4}\Gamma(\alpha+1) + \left(\frac{3}{2}t^{\frac{\alpha}{2}} - t^4\right)^3 - y(t)^{\frac{3}{2}} \tag{3.5}$$

$$y(0) = 0, \quad y'(0) = 0$$

whose exact solution is given by:

$$y(t) = t^8 - 3t^{4+\frac{\alpha}{2}} + \frac{9}{4}t^\alpha$$

Sample Equation 2

$$D_0^\alpha y(t) = \frac{\Gamma(4+\alpha)}{6} t^3 + t^{3+\alpha} - y(t) \tag{3.6}$$

$$y(0) = 0, \quad y'(0) = 0$$

whose exact solution is given by:

$$y(t) = t^{3+\alpha}$$

Sample Equation 3

$$D_0^\alpha y(t) = \frac{\Gamma(5+\alpha)}{24} t^4 + t^{8+2\alpha} - y^2(t) \tag{3.7}$$

$$y(0) = 0, \quad y'(0) = 0$$

whose exact solution is given by:

$$y(t) = t^{4+\alpha}$$

Sample Equation 4

$$D_0^\alpha y(t) = \begin{cases} \frac{2}{\Gamma(3-\alpha)} t^{2-\alpha} - y(t) + t^2 - t, & \alpha > 1, \\ \frac{2}{\Gamma(3-\alpha)} t^{2-\alpha} - \frac{1}{\Gamma(2-\alpha)} t^{1-\alpha} - y(t) + t^2 - t, & \alpha \leq 1, \end{cases} \tag{3.8}$$

$$y(0) = 0, \quad y'(0) = -1$$

whose exact solution is given by:

$$y(t) = t^2 - t$$

These equations are simulated using both competing numerical methods, and are also compared against their true solutions. These four test cases also have the added benefit for this analysis that they have also been simulated and reported in [1]. This gives four test cases to compare this study against, not only for the actual solutions, but also against the findings of Nguyen and Jang.

3.2 Error Analysis

Upon completion of the software development and simulations, error analysis can now be conducted. The three major questions that this research attempts to answer with regard to error analysis include:

1. How does the point-wise error at the final time, $t_N = T$ compare between the methods?
2. How does the least squares error, E_{L^2}, over the time interval compare between the methods?
3. Has the method proposed by Nguyen and Jang yielded a constant order of error of $\mathcal{O}(h^2)$ for the 2^{nd} order scheme, and $\mathcal{O}(h^3)$ for the 3^{nd} order scheme, as they claim, regardless of α?

Pointwise Error Calculation

The pointwise error of the simulations will be calculated for the methods in question using the following:

$$E_{pt}(T) = |y_N - \tilde{y}_N| \tag{3.9}$$

Least Squares Error Calculation

The second measure of error used for this analysis will be a least squares error over the full domain of the problem, calculated using the following:

$$E_{L^2} = \left(h \sum_{j=0}^{N} |y_j - \tilde{y}_j|^2 \right)^{\frac{1}{2}} \tag{3.10}$$

Order of Error

One of the hallmark achievements claimed by this new method is a reduction in order of error, specifically whenever $\alpha < 1$. This is because the order of the ABM method is an error of order $\mathcal{O}(h^p)$, where $p = min(\alpha + 1, 2)$, as shown in [9]. It is therefore clear that for values of alpha less than one, especially with $h \ll 1$, the magnitude of error will be significantly larger for the ABM method. In order to assess this claim, the above two measures of error, Eq. (3.9) and Eq. (3.10),

will be used to calculate the order of error for each of the methods.

This order of error is calculated by finding the slope of the error plots for both measures of error described above. The local slope, m, is calculated between two neighboring reported errors and their respective step sizes, where $h = [h_1, h_2, ..., h_i, ..., h_n]$, and E_i represents the measure at that corresponding step-size:

$$m_{i+1} = \frac{\log(E_{i+1}) - \log(E_i)}{\log(h_{i+1}) - \log(h_i)} \tag{3.11}$$

Each simulation may then yield an overall order by averaging each local slope, m_i, as calculated by

$$M = \frac{1}{n-1} \sum_{i=1}^{n-1} m_{i+1}. \tag{3.12}$$

Each local slope will be reported in the tabulated results of the simulations, while the overall order will appear for each method on the plots displaying error vs. step-size.

3.3 Computational Cost Analysis

The second hallmark achievement that the method by Nguyen and Jang claims is that they have reduced the computational cost of their schemes by half over the ABM method. Their claim is that this increase in speed is a result of the removal of recalculating the memory effect in the predictor step. This idea can be seen in the derivations of the methods, where the same lag term, Eq. (1.23), is used in calculating both the predictor, Eq. (1.21), and the corrector, Eq. (1.22) for the MF-PCL method. In contrast, the ABM method uses two different heredity terms for calculation of the predictor and corrector terms, as seen in Eq. (1.13) and Eq. (1.17), respectively.

The final objective of this research is to run a computational cost study of the methods in question. This will be achieved by running a number of simulation test cases in which both α and the step-size are varied. Each trial case is run five times to collect a reasonable sample size and eliminate any obvious outliers. These five times for each test case will then be used to obtain an

average computational time. The computational time will be collected using the MATLAB® tic and toc functions to provide a consistent and reliable measurement for time collection. Using the same approach as in the error analysis, the computational time and step-size are plotted for several different values of α in order to determine the order of the method with respect to computational expense. These plots help to show the differences in the order of the competing methods, as well as the relative speed for each test case.

4. RESULTS

The following chapter will be separated into two major sections. The 'Expected Results' section will present only data and results from [1]. No data from the independent simulations will be reported here. In contrast, the 'Experimental Results' section will display all results from the independent simulations of the method, as-well-as plots and data comparing the simulated data to the independent simulations.

4.1 Expected Results

4.1.1 Simulation

The findings of [1] include two simulations comparing the performance of ABM against their newly proposed method, referred to as MF-PCL. Both methods are evaluated against the true solution, using a grid size of $N = 20, T = 1$ second, and values of $\alpha = 0.25, 1.25$. These simulations use Eq. (3.5) and its associated initial conditions/true solution. These results are displayed in [1, Fig. 1].

4.1.2 Error Analysis

There are several tables of error calculations reported in [1]. The tables displaying results from Eq. (3.5) will be reproduced below for the purpose of comparison to the independent simulation results. To avoid redundant commentary, the results from the remaining equations are included in Appendix A. Note in the tabulated data to follow that, as indicated, roughly the same order of accuracy is present for $0 < \alpha < 1$ as well as for $\alpha \geq 1$ for the MF-PCL method. Conversely, there is a clear change in order for varying values of α when looking at the ABM method.

Table 4.1 displays the data reported by [1] using Eq. (3.5), and varies simulation parameters of $\alpha = [0.25, 0.5, 1.25]$, and $N = \frac{1}{h} = [10, 20, 40, 80, 160, 320]$. It is acknowledged in [1] that the

Table 4.1: Expected results using Example Equation 1, Eq. (3.5), as reported in [1].

α = 0.25								
	ABM				MF-PCL			
N	E_{pt}	Order	E_{L^2}	Order	E_{pt}	Order	E_{L^2}	Order
10	2.50E-01	-	3.14E-01	-	1.74E-01	-	2.10E-01	-
20	1.81E-02	3.79	8.69E-02	1.85	1.46E-02	3.58	1.44E-02	3.87
40	3.61E-03	2.33	2.48E-02	1.81	2.64E-03	2.46	1.64E-03	3.14
80	1.45E-03	1.31	8.05E-03	1.63	5.07E-04	2.38	2.47E-04	2.73
160	6.58E-04	1.14	2.82E-03	1.52	9.95E-05	2.35	4.13E-05	2.58
320	2.97E-04	1.15	1.03E-03	1.45	2.02E-05	2.3	7.46E-06	2.47
α = 0.5								
10	1.79E-02	-	4.94E-02	-	2.66E-02	-	1.41E-02	-
20	1.81E-03	3.3	1.38E-02	1.84	5.30E-03	2.33	2.09E-03	2.75
40	4.17E-04	2.12	4.15E-03	1.73	1.07E-03	2.3	3.60E-04	2.54
80	1.76E-04	1.24	1.32E-03	1.65	2.31E-04	2.22	7.10E-05	2.35
160	7.98E-05	1.15	4.33E-04	1.61	5.31E-05	2.12	1.56E-05	2.19
320	3.39E-05	1.24	1.46E-04	1.57	1.27E-05	2.06	3.65E-06	2.1
α = 1.25								
10	5.53E-03	-	8.14E-03	-	1.02E-02	-	6.04E-03	-
20	1.59E-03	1.8	1.88E-03	2.11	2.34E-03	2.12	1.38E-03	2.13
40	4.33E-04	1.88	4.43E-04	2.09	5.70E-04	2.04	3.34E-04	2.04
80	1.14E-04	1.92	1.06E-04	2.07	1.41E-04	2.01	8.26E-05	2.02
160	2.97E-05	1.94	2.54E-05	2.06	3.52E-05	2	2.05E-05	2.01
320	7.66E-06	1.96	6.13E-06	2.05	8.80E-06	2	5.12E-06	2

order of accuracy for small step sizes is even higher than anticipated. This behavior is attributed to the competence between the errors in the lag and increment term of the truncation errors when α is small. It should also be noted that the start-up algorithm utilizes a combination of linear and quadratic interpolation, as well as decreased step-size to initialize the second-order linear scheme, and therefore may also result in a higher than anticipated order of accuracy, especially where N is small and the start-up algorithm carries a larger influence.

4.1.3 Computational Cost

Finally, [1] has set a benchmark on computer run-time measurements. While it is not expected that the times themselves will match, this does give a benchmark trend between the two methods,

in terms of relative speeds. There is also no specification on how these values were obtained, how many samples of each run were recorded, nor if this is an average run-time or the fastest run for each test case. Table 4.2 below displays a table of computation time values, with the graphical representation of these times plotted on a log-log scale in Fig. 4.1.

Table 4.2: Expected results from computational cost analysis, as reported in [1].

$\alpha = 0.25$						
N	100	200	400	600	800	1000
ABM (seconds)	0.002	0.01	0.04	0.09	0.16	0.262
MF-PCL (seconds)	0.001	0.007	0.025	0.05	0.096	0.15
$\alpha = 0.5$						
N	100	200	400	600	800	1000
MF-PCQ (seconds)	0.123	0.231	0.622	0.981	1.49	1.471

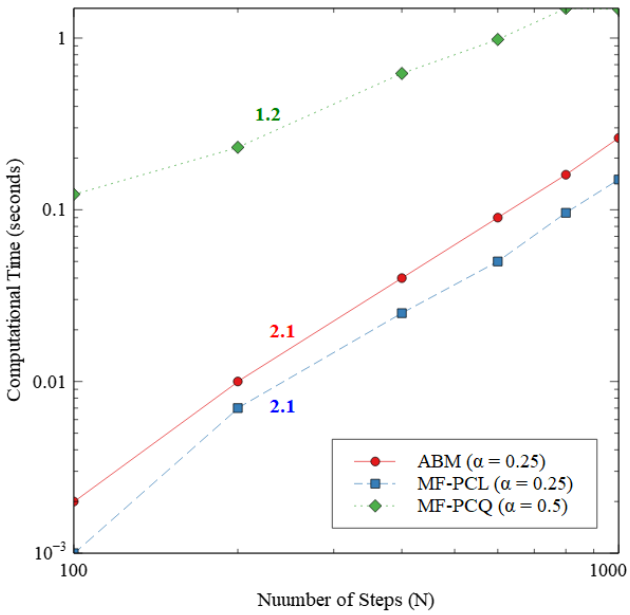

Figure 4.1: Expected results for computational time (in seconds) vs. number of steps (N).

4.2 Experimental Results

4.2.1 Simulation

The following results will display the independent simulations and findings from this research, and compare these findings against those of [1]. Figure 4.2 below is a recreation of the solutions displayed in [1, Fig. 1], in order to provide further evidence of successful reproduction of the methods. This figure displays results using Eq. (3.5) for $N = 20$, and $\alpha = 0.25$ (left) and $\alpha = 1.25$ (right).

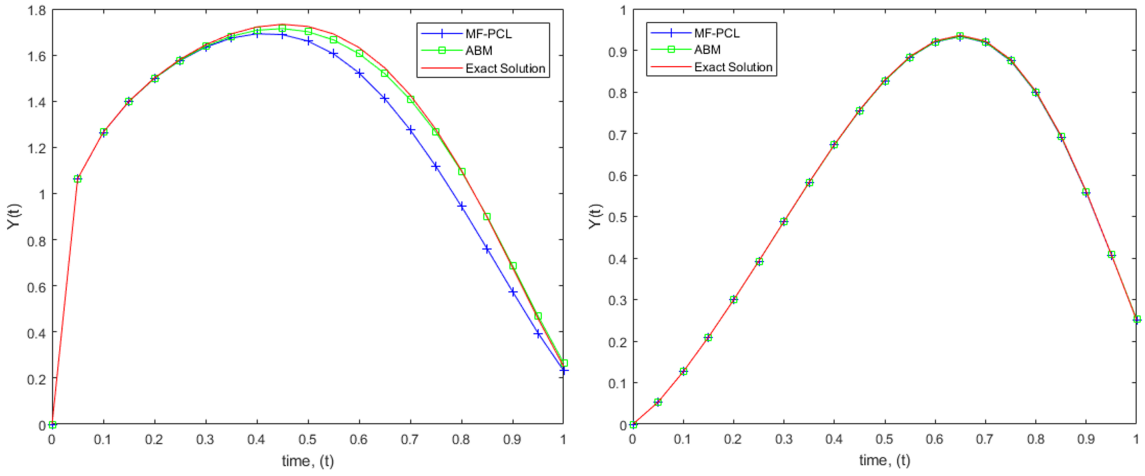

Figure 4.2: Independent simulation results of competing methods. Left: $\alpha = 0.25$, Right: $\alpha = 1.25$.

4.2.2 Error Analysis

Table 4.3 below displays the results from independent simulation of the MF-PCL scheme. Again, the ABM values displayed are generated from independent simulations using the fde12 MATLAB® function provided in [8]. The tabulated data here matches very closely with Table 4.1, with the exception of the results from $\alpha = 0.25$ with high values of N. This claim will be discussed in more detail and proved later.

Table 4.3: Independent simulation results of ABM vs MF-PCL using Example Equation 1, Eq. (3.5)

α = 0.25								
	ABM				MF-PCL			
N	E_{pt}	Order	E_{L^2}	Order	E_{pt}	Order	E_{L^2}	Order
10	2.50E-01	–	3.14E-01	–	1.65E-01	–	1.99E-01	–
20	1.81E-02	3.79	8.69E-02	1.85	1.46E-02	3.49	1.45E-02	3.78
40	3.61E-03	2.33	2.48E-02	1.81	2.69E-03	2.45	1.69E-03	3.10
80	1.45E-03	1.31	8.05E-03	1.63	5.42E-04	2.31	2.80E-04	2.59
160	6.58E-04	1.14	2.82E-03	1.51	1.26E-04	2.10	6.79E-05	2.04
320	2.97E-04	1.15	1.03E-03	1.45	4.13E-05	1.62	3.06E-05	1.15
α = 0.5								
10	1.79E-02	–	4.94E-02	–	2.65E-02	–	1.42E-02	–
20	1.81E-03	3.30	1.38E-02	1.84	5.30E-03	2.32	2.10E-03	2.76
40	4.16E-04	2.12	4.15E-03	1.73	1.07E-03	2.30	3.60E-04	2.54
80	1.77E-04	1.24	1.32E-03	1.65	2.31E-04	2.22	7.10E-05	2.34
160	7.98E-05	1.15	4.33E-04	1.61	5.31E-05	2.12	1.56E-05	2.19
320	3.39E-05	1.24	1.46E-04	1.57	1.27E-05	2.06	3.65E-06	2.09
α = 1.25								
10	5.53E-03	–	8.14E-03	–	1.01E-02	–	6.06E-03	–
20	1.59E-03	1.80	1.88E-03	2.11	2.34E-03	2.11	1.38E-03	2.14
40	4.33E-04	1.88	4.43E-04	2.09	5.70E-04	2.04	3.34E-04	2.04
80	1.14E-04	1.92	1.06E-04	2.07	1.41E-04	2.01	8.26E-05	2.02
160	2.97E-05	1.94	2.54E-05	2.06	3.52E-05	2.00	2.05E-05	2.01
320	7.66E-06	1.96	6.13E-06	2.05	8.80E-06	2.00	5.12E-06	2.00

Recall from prior discussion that the expected order of error of the methods are given as $\mathcal{O}(h^2)$ for the 2^{nd} order scheme with linear interpolation, and $\mathcal{O}(h^3)$ for the 3^{rd} order scheme with quadratic interpolation. Meanwhile, the ABM method is expected to show error on the order of $\mathcal{O}(h^p)$, where $p = min(1 + \alpha, 2)$. Seeing that all of these methods should have an order of error of the form h^M, it is convenient to display the error against the step-size on a log-log plot, where the this data should be a line with approximate slope M ([10]). Recalling from Eq. (3.11), the local slopes can be calculated and then averaged by Eq. (3.12) to obtain an overall order of the method for that specific simulation configuration. This value, M, is displayed on each of the plots to follow, with the exception of the simulations using $\alpha = 0.25$, due to the non-linearity of results.

Table 4.4 displays the results of the MF-PCQ simulation using the same simulation configuration as in Table 4.3. The results from the ABM simulation are reproduced here for the sake of comparison, but are identical to those values presented in Table 4.3. Again, note that there are unexpected results for $\alpha = 0.25$ which will be discussed later.

Table 4.4: Independent simulation results of ABM vs MF-PCQ using Example Equation 1, Eq. (3.5)

α = 0.25								
	ABM				MF-PCQ			
N	E_{pt}	Order	E_{L^2}	Order	E_{pt}	Order	E_{L^2}	Order
10	2.50E-01	–	3.14E-01	–	6.77E-02	–	7.66E-02	–
20	1.81E-02	3.79	8.69E-02	1.85	4.28E-03	3.98	2.48E-03	4.95
40	3.61E-03	2.33	2.48E-02	1.81	4.70E-04	3.18	2.41E-04	3.37
80	1.45E-03	1.31	8.05E-03	1.63	7.67E-05	2.62	5.52E-05	2.12
160	6.58E-04	1.14	2.82E-03	1.51	3.05E-05	1.33	3.33E-05	0.73
320	2.97E-04	1.15	1.03E-03	1.45	2.11E-05	0.53	2.48E-05	0.43
α = 0.5								
10	1.79E-02	–	4.94E-02	–	1.33E-02	–	5.82E-03	–
20	1.81E-03	3.30	1.38E-02	1.84	1.49E-03	3.16	5.60E-04	3.38
40	4.16E-04	2.12	4.15E-03	1.73	1.58E-04	3.23	5.25E-05	3.41
80	1.77E-04	1.24	1.32E-03	1.65	1.75E-05	3.18	5.24E-06	3.32
160	7.98E-05	1.15	4.33E-04	1.61	2.03E-06	3.11	5.68E-07	3.21
320	3.39E-05	1.24	1.46E-04	1.57	2.44E-07	3.05	6.57E-08	3.11
α = 1.25								
10	5.53E-03	–	8.14E-03	–	7.08E-03	–	2.53E-03	–
20	1.59E-03	1.80	1.88E-03	2.11	9.57E-04	2.89	2.95E-04	3.10
40	4.33E-04	1.88	4.43E-04	2.09	1.26E-04	2.93	3.57E-05	3.04
80	1.14E-04	1.92	1.06E-04	2.07	1.62E-05	2.96	4.40E-06	3.02
160	2.97E-05	1.94	2.54E-05	2.06	2.05E-06	2.98	5.47E-07	3.01
320	7.66E-06	1.96	6.13E-06	2.05	2.58E-07	2.99	6.82E-08	3.00

Figure 4.3 and Figure 4.4 display that the ABM method is behaving as expected, with error on the order of $\mathcal{O}(h^p)$. Likewise, the MF-PCL and MF-PCQ methods are maintaining constant error orders of approximately $\mathcal{O}(h^2)$ and $\mathcal{O}(h^3)$, respectively.

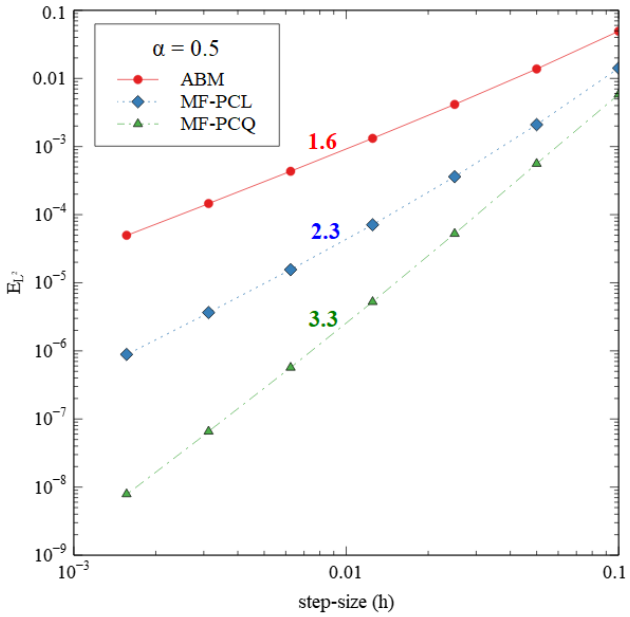

Figure 4.3: Error analysis from independent simulation of competing methods ($\alpha = 0.5$).

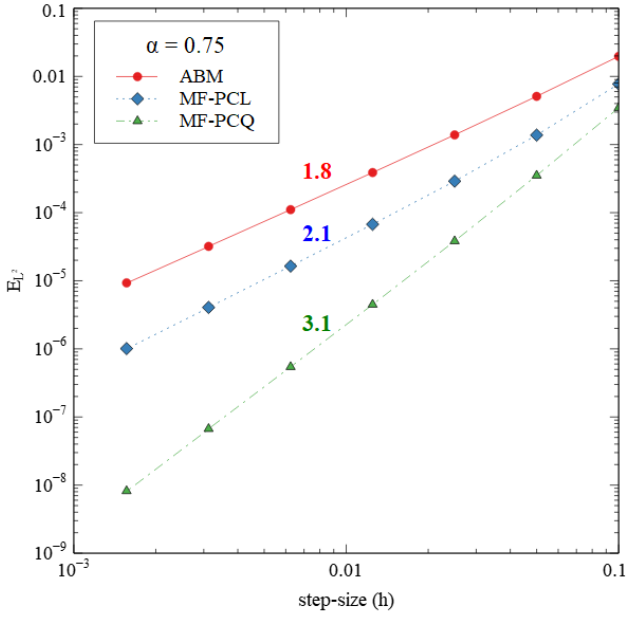

Figure 4.4: Error analysis from independent simulation of competing methods ($\alpha = 0.75$).

With the transition from values of $\alpha < 1$ to $\alpha > 1$, the ABM method should now show a constant error of order $\mathcal{O}(h^2)$. This is shown by the reported order of the method being equal to

$\mathcal{O}(h^p)$, where $p = min(1 + \alpha, 2)$. For all values of $\alpha \geq 1, p = 2$. This means that the MF-PCL and ABM methods should perform equally with respect to accuracy, which is evident in Fig. 4.5 and Fig. 4.6.

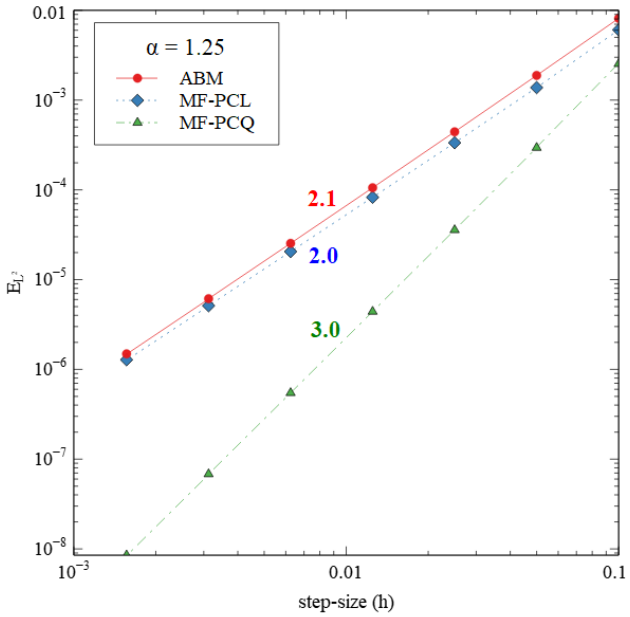

Figure 4.5: Error analysis from independent simulation of competing methods ($\alpha = 1.25$).

As mentioned previously, there exists anomalous behavior for the simulations conducted using a value of $\alpha = 0.25$. Figure 4.7 is produced to illustrate this unexpected result. It can be seen that both the MF-PCL and MF-PCQ schemes appear to asymptotically approach a limited order of accuracy. This behavior is investigated further in the following section.

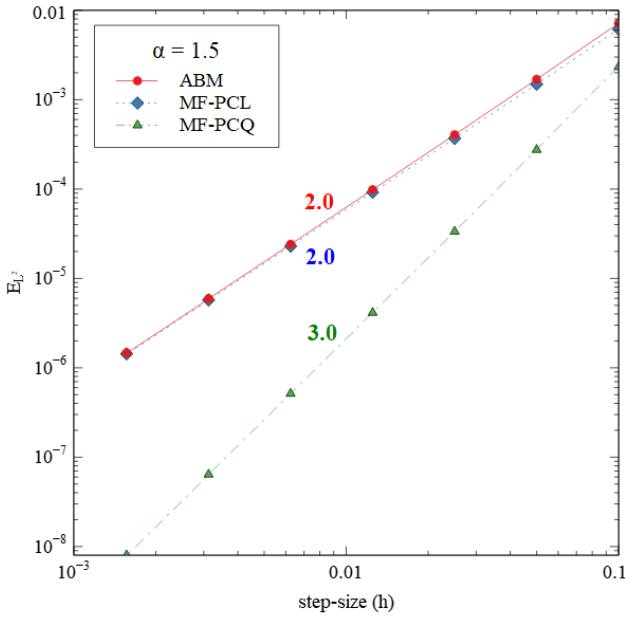

Figure 4.6: Error analysis from independent simulation of competing methods ($\alpha = 1.5$).

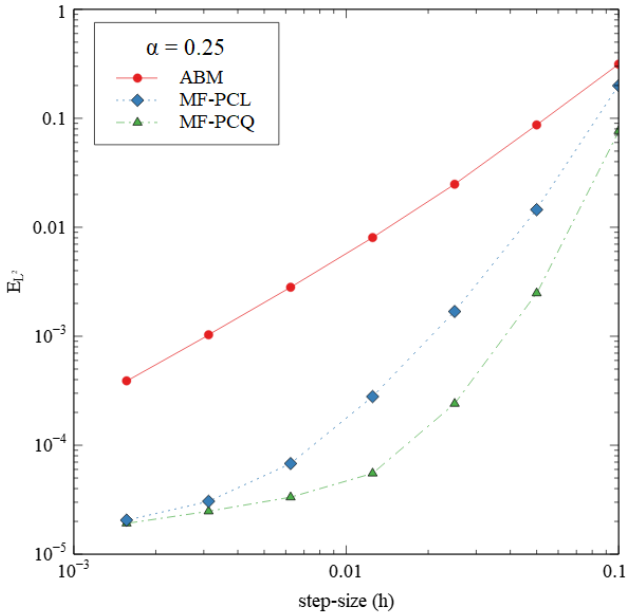

Figure 4.7: Error analysis from independent simulation of competing methods ($\alpha = 0.25$).

Figure 4.8 and Figure 4.9 display the values reported from [1] overlaid with results from the independent simulations conducted in this research. This set of graphics is presented in order to more clearly compare their performance. The independent simulations for these values of α perform nearly identically to the expected results.

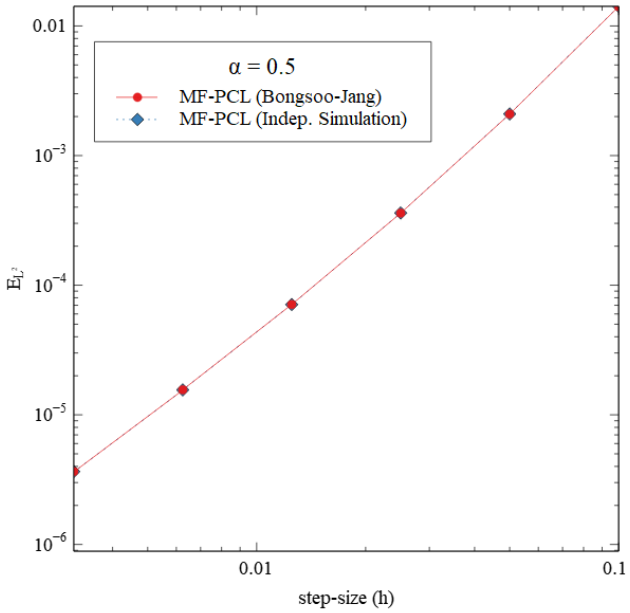

Figure 4.8: Comparison of reported data from [1] with independent simulation, $\alpha = 0.5$.

In contrast, Fig. 4.10 illustrates the magnitude of deviation in performance as N increases. This further illustrates the anomalous behavior that has been previously mentioned, and motivates the need to determine whether this behavior is a characteristic of the method, or rather a misrepresentation of the method's true performance that was brought on by implementation.

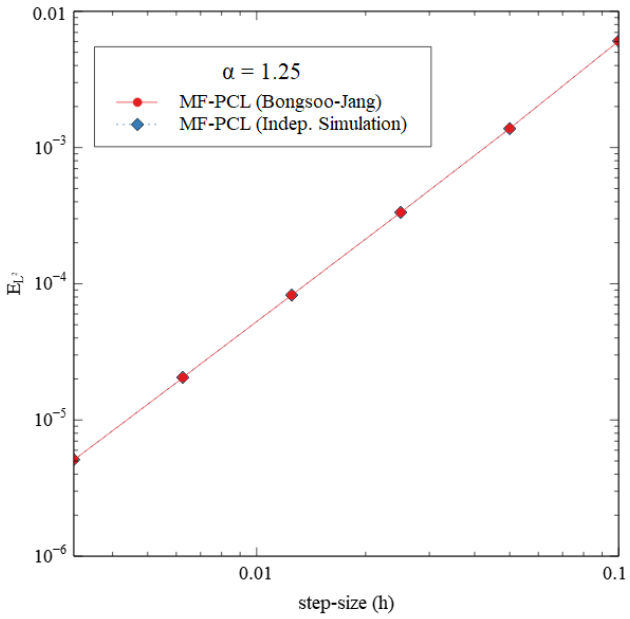

Figure 4.9: Comparison of reported data from [1] with independent simulation, $\alpha = 1.25$.

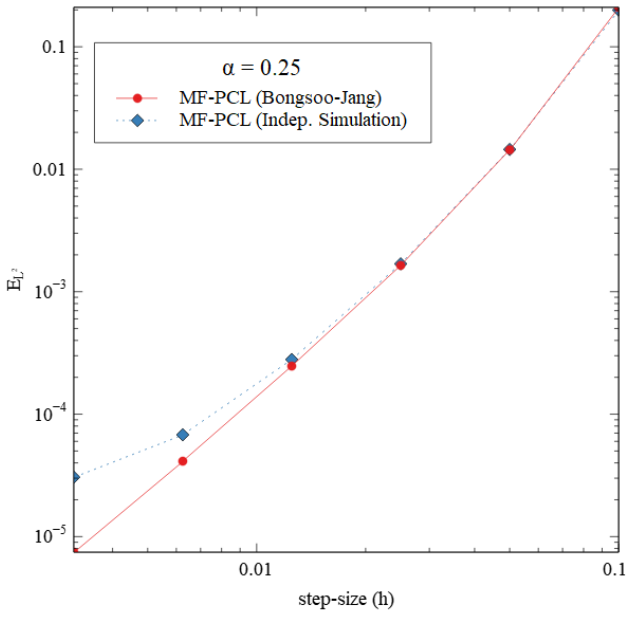

Figure 4.10: Comparison of reported data from [1] with independent simulation, $\alpha = 0.25$.

As an additional means of comparing performance of the two methods with respect to error, Table 4.5 is produced below. This table calculates the percent difference for each measure of error, and its respective order. Percent difference, calculated between findings by [1] and the independent simulations of this research, is computed by

$$\%Difference = \frac{(E_{Ind.Sim.} - E_{[1]})}{\frac{1}{2}(E_{Ind.Sim.} + E_{[1]})} X 100\%, \tag{4.1}$$

where $E_{Ind.Sim.}$ is any value from the independent simulations of this research, and $E_{[1]}$ is any corresponding value from [1]. It should be noted that the results published in [1] are only reported to two decimal places, so some round-off error will be present when computing the percent differences.

Table 4.5: Percent difference of values between those reported in [1], and inpependent simulation values.

α = 0.25								
	ABM				MF-PCL			
N	E_{pt} (%)	Order (%)	E_{L^2} (%)	Order (%)	E_{pt} (%)	Order (%)	E_{L^2} (%)	Order (%)
10	0.03	–	-0.10	–	-5.6	–	-5.3	–
20	-0.03	-0.04	0.05	0.07	0.26	-2.52	0.73	-2.36
40	-0.13	-0.11	0.13	-0.12	1.79	-0.60	2.88	-1.18
80	0.15	0.15	-0.05	-0.26	6.68	-2.99	12.35	-5.10
160	0.01	0.17	-0.17	-0.33	23.85	-11.24	48.66	-23.26
320	-0.04	-0.15	0.18	-0.13	68.55	-34.96	121.56	-72.94
α = 0.5								
10	-0.23	–	-0.07	–	-0.3	–	0.5	–
20	0.12	0.03	-0.23	0.12	-0.08	-0.27	0.24	0.29
40	-0.19	0.12	0.04	-0.02	0.45	0.03	0.13	-0.04
80	0.31	-0.23	-0.09	0.27	0.18	-0.20	-0.07	-0.21
160	-0.01	-0.37	0.10	-0.28	0.00	0.18	-0.21	-0.08
320	0.00	-0.40	-0.21	0.18	0.22	0.03	0.03	-0.38
α = 1.25								
10	0.05	–	-0.05	–	-0.6	–	0.3	–
20	0.20	-0.22	0.11	0.09	0.12	-0.31	-0.18	0.29
40	-0.04	0.00	0.03	-0.16	0.02	-0.05	0.03	0.18
80	0.30	0.03	-0.42	-0.02	0.23	0.11	-0.06	-0.15
160	0.14	0.14	-0.18	-0.11	0.09	0.20	0.15	-0.12
320	0.04	-0.18	-0.02	-0.07	-0.01	0.07	0.00	0.18

Corrected Results Using Analytically-Solved Coefficient Integrals

The above results of the independent simulations for MF-PCL and MF-PCQ contain an area of concern with regard to the combination of low values of α and high values of N. It would appear that there is a subroutine or calculation in the process of carrying out these schemes under certain conditions that is adversely affecting their performance. It can be seen in Fig. 4.7 that both the linear and quadratic interpolation schemes converge to the same order of error, pointing suspicions to the kernel of the integrand found in Eq. (1.18) and Eq. (1.35), given as $(t_{n+1} - \tau)^{\alpha-1}$.

These inconsistencies were discovered to be the result of computational error of a lower order

than the method when B coefficients (or A coefficients in the case of MF-PCQ) in the increment term, specifically in the B_1^n term when $t_{n+1} = t_{j+1}$. This is brought on by the MATLAB® integral function performing poorly for low values of alpha when evaluating this integral explicitly, when compared against the analytical solutions. With increasing number of steps, N, this integration error becomes more apparent, and it can be seen to converge to an order of this integral evaluation error. This is because the method at low step size is of lower order of error than the integration error. However, as the step-size decreases and the method reaches the crossover point of exhibiting a higher order of error than this integration error, the integration error overshadows the method. Once this error is larger than the method's error, it dictates the apparent order of the method.

However, for cases where α is larger, the MATLAB® integral function performs much more accurately, and the expected order of the method for each respective scheme is clearly visible. The findings for these cases match very closely to both the theoretical order of error and the findings reported in [1].

In order to prove the above claim, a short analysis of the calculation of the B coefficients is presented. Note that the analysis is not repeated for the A coefficients, as the kernel of the integrand, which is the same for both, is the source of this error, and not the interpolation functions that multiply onto the kernel.

Using the MATLAB® symbolic math package, definite integrals can be solved analytically with the *int()* function, and then compared against the numerical integration function, *integral()*. Figure 4.11 displays this analysis on the $B_{n+1}^{0,n}$ coefficient, while Fig. 4.12 presents the same analysis on the $B_{n+1}^{1,n}$ coefficent, with these coefficients being defined in Eq. (1.18), specifically when applied in Eq. (1.24).

This analysis was carried out by taking the average point-wise error at a single time-step when

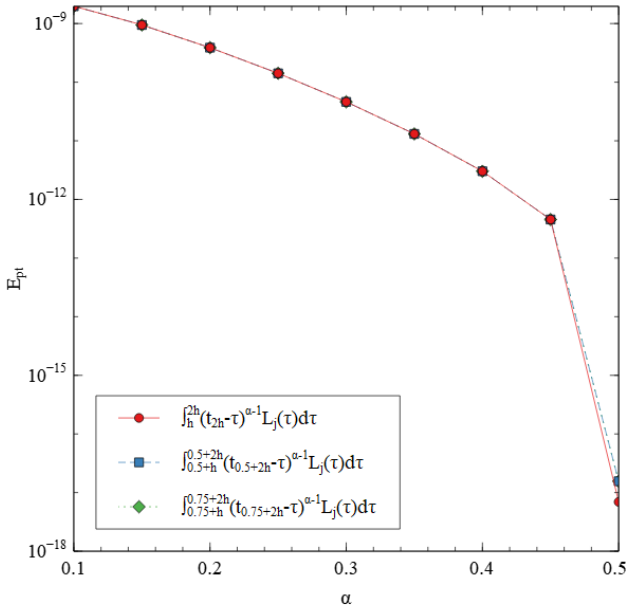

Figure 4.11: $B_{n+1}^{0,n}$ Coefficient Error Analysis

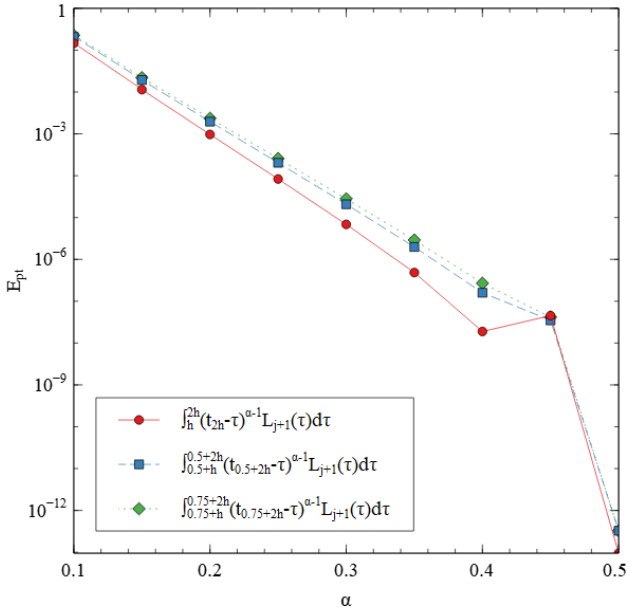

Figure 4.12: $B_{n+1}^{1,n}$ Coefficient Error Analysis

calculated with several different step-sizes, $h = [\frac{1}{10}, \frac{1}{20}, \frac{1}{40}, \frac{1}{80}, \frac{1}{160}, \frac{1}{320}, \frac{1}{640}, \frac{1}{1280}]$. The point-wise error was then calculated across a range of low α values. This analysis was repeated for three total

time-steps. Multiple step-sizes and time-steps were used to eliminate confounding results, and verify that this error is in fact a function of α.

Figure 4.11 shows that there is negligible error in this coefficient calculation for values of α as low as 0.1. However, the error seen in Fig. 4.12 is exceptionally large up to values of roughly $\alpha = 0.45$. This error present in the numerical integration of the $B_{n+1}^{1,n}$ coefficient has the potential to be the driving factor in the order of the method for simulations with $\alpha < 0.5$, particularly where N is large.

Figure 4.13 displays that, with the substitution of the analytical calculation of the B coefficients in the increment term, this previously-shown anomaly ceases to persist. With this new finding, this research is fully successful in replicating the findings presented in [1] with respect to the proposed method. Table 4.6 now matches very closely with the findings of [1], as reported in Table 4.1.

The decision was made that the results should be reported using purely the numerical integration, rather than using the analytical integration as a fix. This choice was made for two major reasons. First, this research is a study of pure numerical methods, and it is desired to replicate the method using only techniques of numerical methods. The use of symbolic math equation solvers would violate this goal. Second, replicating the results of [1] using the analytical integral solver would make the software package much more difficult to reproduce using other programming languages. By reporting these results, including the shortcomings of the MATLAB® numerical integral solver, this research draws attention to this concern, and serves as a point of warning and encouragement to study more effective numerical methods of solving integrals of this type.

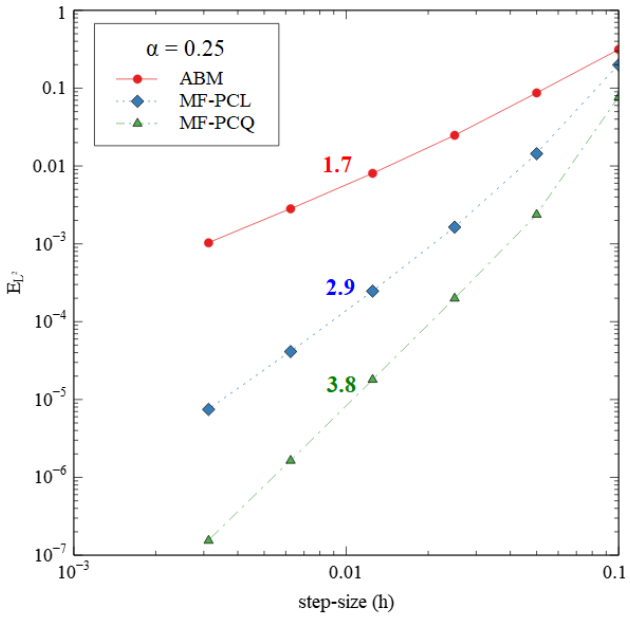

Figure 4.13: Error analysis from independent simulation of competing methods ($\alpha = 0.25$), using E_{L^2}, with analytically-solved coefficient integrals.

Table 4.6: Independent simulation results of MF-PCL and MF-PCQ using Example Equation 1, Eq. (3.5), using analytical solutions to B coefficients and A coefficients

$\alpha = 0.25$								
	MF-PCL				MF-PCQ			
N	E_{pt}	Order	E_{L^2}	Order	E_{pt}	Order	E_{L^2}	Order
10	1.64E-01	–	1.99E-01	–	6.76E-02	–	7.60E-02	–
20	1.46E-02	3.49	1.44E-02	3.79	4.19E-03	4.01	2.37E-03	5.00
40	2.64E-03	2.46	1.64E-03	3.14	4.24E-04	3.30	1.99E-04	3.57
80	5.07E-04	2.38	2.47E-04	2.73	4.26E-05	3.32	1.80E-05	3.47
160	9.95E-05	2.35	4.13E-05	2.58	4.26E-06	3.32	1.64E-06	3.45
320	2.02E-05	2.30	7.47E-06	2.47	4.33E-07	3.30	1.54E-07	3.41

4.2.3 Computational Cost

In order to test the competing numerical methods with respect to computational time, it was important to ensure that the subroutines used to calculate terms were the same. This means that the same integration method should be used across all methods. By this standard, the *fde12()* function should not be used as the standard to compare these methods against in regard to computation time. This is because the coefficients are evaluated using the FFT algorithm described in [11], which increases performance of the computational cost to $N\log(N)^2$ instead of N^2 in the traditional ABM implementation. Therefore, a traditional implementation of the ABM method, as re-derived in [1], is required. By creating an independent ABM method, this research is able to compare the actual methods in question, with both the MF-PCL/MF-PCQ and ABM using the same set of subroutines.

Table 4.7: Results from computational cost analysis (time for each method is reported in seconds).

$\alpha = 0.25$							
N	10	20	40	50	100	200	400
ABM	0.07	0.20	0.65	1.02	3.86	15.35	64.78
MF-PCL	0.11	0.23	0.70	1.06	3.92	15.57	62.84
MF-PCQ	0.27	0.44	1.24	1.77	6.15	23.16	95.69
fde12	0.03	0.03	0.04	0.04	0.05	0.05	0.07
$\alpha = 0.75$							
N	10	20	40	50	100	200	400
ABM	0.05	0.17	0.65	1.00	3.91	15.42	62.02
MF-PCL	0.07	0.22	0.67	0.99	3.96	15.60	62.38
MF-PCQ	0.24	0.42	1.20	1.77	6.01	23.14	92.35
fde12	0.03	0.04	0.04	0.04	0.05	0.05	0.06
$\alpha = 1.25$							
N	10	20	40	50	100	200	400
ABM	0.05	0.17	0.67	1.07	3.89	15.42	61.92
MF-PCL	0.07	0.20	0.71	1.02	3.88	15.43	62.14
MF-PCQ	0.22	0.40	1.19	1.74	5.98	23.10	91.68
fde12	0.03	0.04	0.04	0.04	0.05	0.05	0.06

The findings of this research do not show any conclusive results that would suggest that the MF-PCL scheme performs at the one-half computational cost claimed by [1]. However, this also does not disprove their claim. The completion of the computational cost analysis points again to the MATLAB® numerical integration function, *integral()*, which appears to have a very high computational cost. This claim is made due to the fact that computation of the heredity term in the corrector is the most computationally expensive portion of the method. This term contains a call to the MATLAB® *integral()* function in both methods. The major difference between the methods stems from the difference in calculation of the heredity term in the predictor. In the ABM method, Eq. (1.14) shows the heredity calculation is merely a sum with only simple mathematical operations being performed for each term being the summed. The computational time for a single calculation of this term is roughly between the order of 10^{-5} and 10^{-6} seconds. Comparatively, the computational time of calculating the B coefficients one time is $10^{-}3$ seconds. Therefore, the benefits of eliminating a small-cost term in the predictor will be hidden when both methods are running a routine that is much more computationally cumbersome on each step, especially as the heredity term grows.

It can be seen in Table 4.7 that while the order of computational time appears to follow the same order with respect to number of steps, the actual computational time is several orders of magnitude higher than those reported by [1]. Therefore, it is believed that any improvements in speed are much too small to be detectable using the current integration technique. This finding provides further encouragement for future research into alternative or new integration techniques that are both fast and capable of accurately calculating integrals of this type.

Additionally, this analysis includes the startup procedure in the cost for the MF-PCL/MF-PCQ schemes. The ABM method does not have a startup procedure in its traditional application, so this helps to explain the initial gap in computation time for low step-sizes, as seen in Fig. 4.14, Fig. 4.16, and 4.16. It is unclear if the startup-procedure is included in the results published in [1].

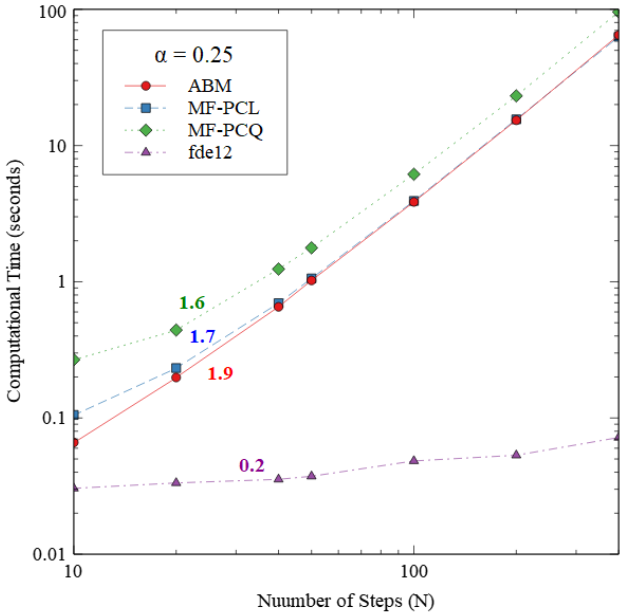

Figure 4.14: Computational cost results for $\alpha = 0.25$.

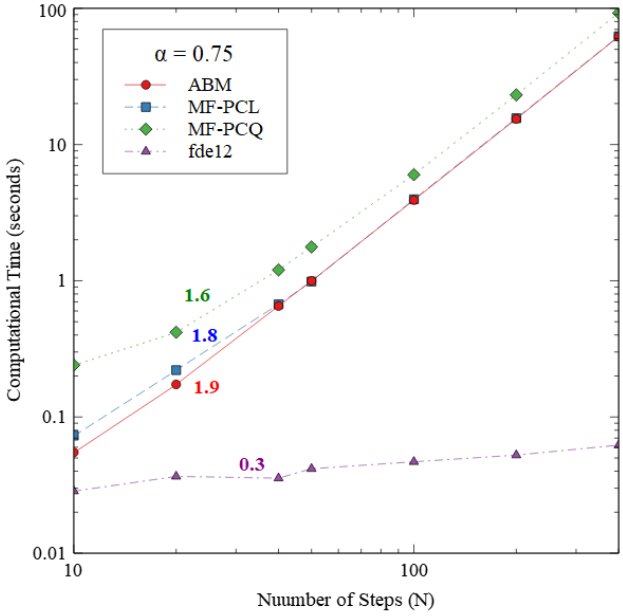

Figure 4.15: Computational cost results for $\alpha = 0.75$.

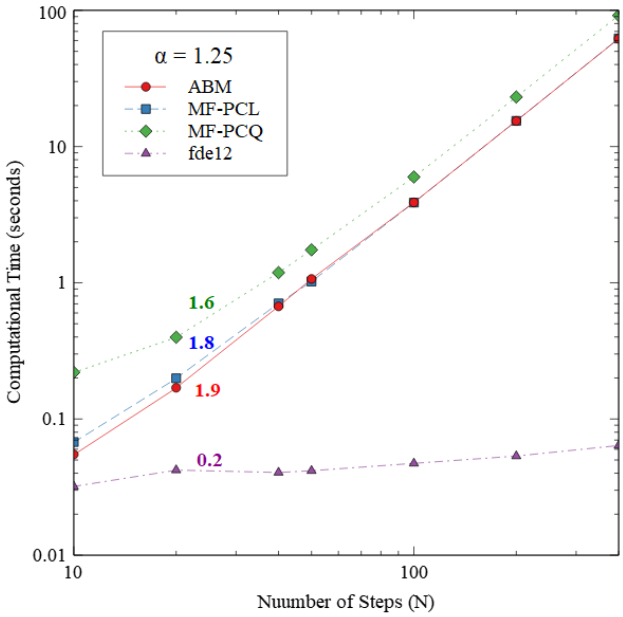

Figure 4.16: Computational cost results for $\alpha = 1.25$.

5. CONCLUSIONS AND FUTURE WORK

5.1 Conclusions

Using only the methods presented by Nguyen and Jang in [1], this research was able to successfully replicate their reported results, and confirm the claimed orders of $\mathcal{O}(h^2)$ for the 2^{nd} order scheme with linear interpolation, MF-PCL, and $\mathcal{O}(h^3)$ for the 3^{rd} order scheme with quadratic interpolation (MF-PCQ), regardless of the order of α. This finding, satisfying the second objective of this research, comes with one caveat, which is also displayed and briefly mentioned in [1]. The solutions to simulations of Eq. (3.8), which are included in Appendix A of this paper and match very closely with the results of Nguyen and Jang, show that the expected order of error does not hold here. They justify this deviation with the fact that Eq. (3.8) violates a theorem presented in their error analysis that requires the function be Lipschitz continuous.

With this acknowledged, the successful replication both of these schemes under Nguyen and Jang's new method provides validation of the software package that was developed to implement this method, satisfying the first objective as well. This success is also met with one caveat, which is the limitation of using the MATLAB® numerical integration function, *integral()*. This research has displayed the shortcomings of this integration function, and its profound effect on the accuracy of the method under certain simulation configurations. The range of acceptable usage of this built-in MATLAB® function with respect to this software package has been laid out in the discussion in the results chapter.

Unfortunately, another short-coming of the MATLAB® integration function is its large computational cost. This function caused difficulty to be able to see any statistically significant speed improvements of MF-PCL over the ABM method, and also greatly increased overall computational times. However, the MF-PCL scheme also did not perform more slowly than the existing method,

while boasting improved accuracy for $\alpha < 1$.

5.2 Future Work

As mentioned above, there are multiple justifications for the replacement of the built-in MATLAB® numerical integration function. It is suggested that a new/different method of calculating numerical integrals of this class be used in its place that would perform much more accurately at low values of α. This replacement integration method should also greatly improve the software package's computational speed. One proposed option would be use of the technique found in fde12 ([8]), which uses a fast Fourier transform (FFT) algorithm to solve the convolution integrals.

Additionally, this software package was developed using as few built-in MATLAB® functions as possible. This was done in effort to minimize the language and environment dependence of the method's implementation. Writing software in this manner makes it much easier to transition this method into other programming languages. It is suggested that this software be transitioned over to other languages, such as Python, in order to broaden its usage.

Finally, it is suggested that this method be tested using additional FDE's not reported in this thesis or [1]. More specifically, additional test cases should be sought with non-homogeneous initial conditions, as only one case was presented in Nguyen and Jang's article, and it was shown to deviate from the expected order of accuracy. Further investigation should be performed to see if this is in fact a factor in the deviation from expected performance.

REFERENCES

[1] T. B. Nugyen and B. Jang, "A high-order predictor-corrector method for solving nonlinear differential equations of fractional order," *Fractional Calculus and Applied Analysis*, vol. 20, No. 2, pp. 447–476, 2017.

[2] I. Podlubny, *Fractional Differential Equations*. Academic Press, 1999.

[3] M. Caputo, "Linear models of dissipation whose q is almost frquency independent," *The Geophysical Journal of the Royal Astronomical Society*, vol. 13, pp. 529–539, 1967.

[4] M. Rahimy, "Applications of fractional differential equations," *Applied Mathematical Sciences*, vol. 4, pp. 2453–2461, 2010.

[5] K. Diethelm, *The Analysis of Fractional Differential Equations*. Springer, 2010.

[6] D. Conte and I. D. Prete, "Fast collocation methods for volterra integral equations of convolution type," *Journal of Computational and Applied Mathematics*, vol. 196, pp. 652–663, 2006.

[7] K. Diethelm, N. J. Ford, and A. D. Freed, "A predictor-corrector approach for the numerical solution of fractional differential equations," *Nonlinear Dynamics*, vol. 29, pp. 3–22, 2002.

[8] R. Garrappa, "Predictor-corrector pece method for fractional differential equations." `https://www.mathworks.com/matlabcentral/fileexchange/32918-predictor-corrector-pece-method-for-fractional-differential-equations`, 2012.

[9] K. Diethelm, N. J. Ford, and A. D. Freed, "Detailed error analysis for a fractional adams method," *Numerical Algorithms*, vol. 36, pp. 31–52, 2004.

[10] J. Butcher, *Numerical Methods for Ordinary Differential Equations*. Wiley, 2008.

[11] E. Harrier, C. Lubich, and M. Schlichte, "Fast numerical solution of nonlinear volterra convolution equations," *SIAM J. Sci. Statist. Comput.*, vol. 6, pp. 532–541, 1985.

APPENDIX A

ADDITIONAL RESULTS, EXPECTED AND INDEPENDENTLY SIMULATED

A.1 Example Equation 2 Results

A.1.1 Expected Results, from [1]

Table A.1: Expected results using Example Equation 2, Eq. (3.6), as reported in [1].

α = 0.25								
	ABM				MF-PCL			
N	E_{pt}	Order	E_{L^2}	Order	E_{pt}	Order	E_{L^2}	Order
10	1.20E-01	-	5.84E-02	-	2.17E-02	-	1.32E-02	-
20	4.46E-02	1.43	2.07E-02	1.49	3.71E-03	2.54	2.22E-03	2.57
40	1.65E-02	1.43	7.48E-03	1.47	6.64E-04	2.48	3.92E-04	2.5
80	6.20E-03	1.41	2.77E-03	1.44	1.25E-04	2.41	7.26E-05	2.43
160	2.36E-03	1.39	1.05E-03	1.4	2.46E-05	2.34	1.42E-05	2.36
320	9.15E-04	1.37	4.03E-04	1.38	5.09E-06	2.27	2.91E-06	2.29
α = 0.5								
10	4.39E-02	-	2.14E-02	-	8.33E-03	-	4.98E-03	-
20	1.38E-02	1.67	6.32E-03	1.76	1.53E-03	2.44	8.76E-04	2.51
40	4.41E-03	1.64	1.96E-03	1.69	3.20E-04	2.26	1.76E-04	2.31
80	1.44E-03	1.61	6.28E-04	1.64	7.28E-05	2.14	3.92E-05	2.17
160	4.82E-04	1.58	2.07E-04	1.6	1.74E-05	2.07	9.24E-06	2.08
320	1.63E-04	1.56	6.97E-05	1.57	4.25E-06	2.03	2.25E-06	2.04
α = 1.25								
10	1.33E-02	-	6.41E-03	-	9.46E-03	-	4.75E-03	-
20	3.17E-03	2.07	1.44E-03	2.16	2.34E-03	2.02	1.10E-03	2.1
40	7.60E-04	2.06	3.35E-04	2.1	5.83E-04	2	2.67E-04	2.05
80	1.83E-04	2.05	7.97E-05	2.07	1.46E-04	2	6.56E-05	2.02
160	4.43E-05	2.05	1.92E-05	2.05	3.64E-05	2	1.63E-05	2.01
320	1.08E-05	2.04	4.66E-06	2.04	9.10E-06	2	4.05E-06	2.01

Table A.2: Expected results using Example Equation 2, Eq. (3.6), as reported in [1].

$\alpha = 0.2$				
MF-PCQ				
N	E_{pt}	Order	E_{L^2}	Order
10	3.25E-03	-	2.93E-03	-
20	2.13E-04	3.93	2.32E-04	3.66
40	1.89E-05	3.49	2.02E-05	3.52
80	1.72E-06	3.46	1.83E-06	3.46
160	1.63E-07	3.4	1.71E-07	3.42
320	1.59E-08	3.36	1.65E-08	3.38
$\alpha = 0.5$				
10	5.81E-04	-	5.63E-04	-
20	5.09E-05	3.51	4.69E-05	3.58
40	5.24E-06	3.28	4.56E-06	3.36
80	5.93E-07	3.14	4.96E-07	3.2
160	7.06E-08	3.07	5.77E-08	3.1
320	8.63E-09	3.03	6.97E-09	3.05
$\alpha = 1.5$				
10	5.81E-04	-	5.63E-04	-
20	5.09E-05	3.51	4.69E-05	3.58
40	5.24E-06	3.28	4.56E-06	3.36
80	5.93E-07	3.14	4.96E-07	3.2
160	7.06E-08	3.07	5.77E-08	3.1
320	8.63E-09	3.03	6.97E-09	3.05

A.1.2 Independent Simulation Results

Table A.3: Independent simulation results of ABM vs MF-PCL using Example Equation 2, Eq. (3.6).

α = 0.25								
	ABM				MF-PCL			
N	E_{pt}	Order	E_{L^2}	Order	E_{pt}	Order	E_{L^2}	Order
10	1.20E-01	-	5.84E-02	-	2.16E-02	-	1.32E-02	-
20	4.46E-02	1.43	2.07E-02	1.49	3.65E-03	2.57	2.20E-03	2.59
40	1.65E-02	1.43	7.48E-03	1.47	6.13E-04	2.57	3.73E-04	2.56
80	6.20E-03	1.41	2.77E-03	1.44	8.55E-05	2.84	5.89E-05	2.67
160	2.36E-03	1.39	1.05E-03	1.40	6.16E-06	3.79	5.55E-06	3.41
320	9.15E-04	1.37	4.03E-04	1.38	1.95E-05	-1.66	6.87E-06	-0.31
α = 0.5								
10	4.39E-02	-	2.14E-02	-	8.36E-03	-	5.02E-03	-
20	1.38E-02	1.67	6.32E-03	1.76	1.54E-03	2.44	8.77E-04	2.52
40	4.41E-03	1.64	1.96E-03	1.69	3.20E-04	2.26	1.76E-04	2.31
80	1.44E-03	1.61	6.28E-04	1.64	7.28E-05	2.14	3.92E-05	2.17
160	4.82E-04	1.58	2.07E-04	1.60	1.74E-05	2.07	9.24E-06	2.08
320	1.63E-04	1.56	6.97E-05	1.57	4.25E-06	2.03	2.25E-06	2.04
α = 1.25								
10	1.33E-02	-	6.41E-03	-	9.48E-03	-	4.76E-03	-
20	3.17E-03	2.07	1.44E-03	2.16	2.34E-03	2.02	1.10E-03	2.11
40	7.60E-04	2.06	3.35E-04	2.10	5.83E-04	2.00	2.67E-04	2.05
80	1.83E-04	2.05	7.97E-05	2.07	1.46E-04	2.00	6.56E-05	2.02
160	4.43E-05	2.05	1.92E-05	2.05	3.64E-05	2.00	1.63E-05	2.01
320	1.08E-05	2.04	4.66E-06	2.04	9.10E-06	2.00	4.05E-06	2.01

Table A.4: Independent simulation results of ABM vs MF-PCQ using Example Equation 2, Eq. (3.6).

α = 0.2								
	ABM				MF-PCQ			
N	E_{pt}	Order	E_{L^2}	Order	E_{pt}	Order	E_{L^2}	Order
10	1.58E-01	-	7.66E-02	-	2.57E-03	-	2.72E-03	-
20	6.17E-02	1.35	2.87E-02	1.42	2.96E-04	3.12	1.87E-04	3.86
40	2.37E-02	1.38	1.07E-02	1.42	3.78E-04	-0.35	1.46E-04	0.36
80	9.14E-03	1.37	4.10E-03	1.39	3.14E-04	0.26	1.22E-04	0.25
160	3.58E-03	1.35	1.59E-03	1.36	2.56E-04	0.29	9.91E-05	0.30
320	1.42E-03	1.33	6.30E-04	1.34	2.11E-04	0.28	8.11E-05	0.29
α = 0.5								
10	4.39E-02	-	2.14E-02	-	5.81E-04	-	5.63E-04	-
20	1.38E-02	1.67	6.32E-03	1.76	5.09E-05	3.51	4.69E-05	3.58
40	4.41E-03	1.64	1.96E-03	1.69	5.24E-06	3.28	4.56E-06	3.36
80	1.44E-03	1.61	6.28E-04	1.64	5.93E-07	3.14	4.96E-07	3.20
160	4.82E-04	1.58	2.07E-04	1.60	7.06E-08	3.07	5.77E-08	3.10
320	1.63E-04	1.56	6.97E-05	1.57	8.63E-09	3.03	6.97E-09	3.05
α = 1.5								
10	1.34E-02	-	6.31E-03	-	1.16E-03	-	6.37E-04	-
20	3.20E-03	2.06	1.42E-03	2.15	1.55E-04	2.91	8.35E-05	2.93
40	7.76E-04	2.04	3.33E-04	2.09	1.99E-05	2.95	1.07E-05	2.97
80	1.90E-04	2.03	8.03E-05	2.05	2.53E-06	2.98	1.35E-06	2.98
160	4.66E-05	2.02	1.96E-05	2.03	3.19E-07	2.99	1.70E-07	2.99
320	1.15E-05	2.02	4.83E-06	2.02	4.00E-08	2.99	2.13E-08	3.00

A.2 Example Equation 3 Results

A.2.1 Expected Results, from [1]

Table A.5: Expected results using Example Equation3, Eq. (3.7), as reported in [1].

α = 0.25								
	ABM				MF-PCL			
N	E_{pt}	Order	E_{L^2}	Order	E_{pt}	Order	E_{L^2}	Order
10	2.85E-01	-	9.86E-02	-	1.10E-01	-	3.80E-02	-
20	1.41E-01	1.02	4.00E-02	1.3	2.49E-02	2.15	6.99E-03	2.44
40	5.87E-02	1.26	1.49E-02	1.43	4.10E-03	2.6	1.07E-03	2.7
80	2.17E-02	1.43	5.32E-03	1.48	6.10E-04	2.75	1.68E-04	2.67
160	7.75E-03	1.49	1.91E-03	1.48	9.58E-05	2.67	2.92E-05	2.52
320	2.79E-03	1.47	7.03E-04	1.44	1.63E-05	2.56	5.56E-06	2.4
α = 0.5								
10	1.10E-01	-	3.92E-02	-	3.05E-02	-	1.18E-02	-
20	3.79E-02	1.54	1.15E-02	1.77	4.73E-03	2.69	1.80E-03	2.7
40	1.23E-02	1.62	3.47E-03	1.73	7.84E-04	2.59	3.29E-04	2.45
80	3.97E-03	1.63	1.08E-03	1.68	1.50E-04	2.38	6.97E-05	2.24
160	1.30E-03	1.61	3.51E-04	1.63	3.24E-05	2.21	1.61E-05	2.11
320	4.34E-04	1.58	1.16E-04	1.59	7.52E-06	2.11	3.89E-06	2.05

Table A.6: Expected results using Example Equation 3, Eq. (3.7), as reported in [1].

$\alpha = 0.2$				
MF-PCQ				
N	E_{pt}	Order	E_{L^2}	Order
10	3.88E-02	-	1.31E-02	-
20	4.80E-03	3.01	1.26E-03	3.37
40	3.97E-04	3.6	9.26E-05	3.77
80	2.74E-05	3.86	6.68E-06	3.79
160	1.99E-06	3.78	5.41E-07	3.63
320	1.59E-07	3.65	4.79E-08	3.5
$\alpha = 0.5$				
10	4.92E-03	-	1.98E-03	-
20	3.45E-04	3.84	1.45E-04	3.77
40	2.69E-05	3.68	1.32E-05	3.46
80	2.46E-06	3.45	1.41E-06	3.23
160	2.56E-07	3.27	1.63E-07	3.11
320	2.90E-08	3.14	1.96E-08	3.05
$\alpha = 1.5$				
10	3.06E-03	-	1.45E-03	-
20	4.00E-04	2.93	1.80E-04	3.01
40	5.12E-05	2.97	2.24E-05	3.01
80	6.48E-06	2.98	2.79E-06	3
160	8.15E-07	2.99	3.49E-07	3
320	1.02E-07	3	4.36E-08	3

A.2.2 Independent Simulation Results

Table A.7: Independent simulation results of ABM vs MF-PCL using Example Equation 3, Eq. (3.7).

$\alpha = 0.25$								
	ABM				MF-PCL			
N	E_{pt}	Order	E_{L^2}	Order	E_{pt}	Order	E_{L^2}	Order
10	2.85E-01	-	9.86E-02	-	1.10E-01	-	3.80E-02	-
20	1.41E-01	1.02	4.00E-02	1.30	2.48E-02	2.15	6.95E-03	2.45
40	5.87E-02	1.26	1.49E-02	1.43	4.00E-03	2.63	1.05E-03	2.73
80	2.17E-02	1.43	5.32E-03	1.48	5.52E-04	2.86	1.51E-04	2.79
160	7.75E-03	1.49	1.91E-03	1.48	5.63E-05	3.29	1.67E-05	3.18
320	2.79E-03	1.47	7.03E-04	1.44	1.22E-05	2.21	4.11E-06	2.02
$\alpha = 0.5$								
10	1.10E-01	-	3.92E-02	-	3.05E-02	-	1.18E-02	-
20	3.79E-02	1.54	1.15E-02	1.77	4.73E-03	2.69	1.80E-03	2.71
40	1.23E-02	1.62	3.47E-03	1.73	7.84E-04	2.59	3.29E-04	2.45
80	3.97E-03	1.63	1.08E-03	1.68	1.50E-04	2.38	6.97E-05	2.24
160	1.30E-03	1.61	3.51E-04	1.63	3.24E-05	2.21	1.61E-05	2.11
320	4.34E-04	1.58	1.16E-04	1.59	7.52E-06	2.11	3.89E-06	2.05

Table A.8: Independent simulation results of ABM vs MF-PCQ using Example Equation 3, Eq. (3.7).

$\alpha = 0.25$								
	ABM				MF-PCQ			
N	E_{pt}	Order	E_{L^2}	Order	E_{pt}	Order	E_{L^2}	Order
10	3.60E-01	-	1.24E-01	-	3.72E-02	-	1.26E-02	-
20	2.05E-01	0.81	5.66E-02	1.13	3.36E-03	3.47	8.76E-04	3.84
40	9.79E-02	1.07	2.34E-02	1.28	6.13E-04	2.45	1.51E-04	2.53
80	3.92E-02	1.32	8.81E-03	1.41	5.86E-04	0.07	1.49E-04	0.02
160	1.43E-02	1.46	3.22E-03	1.45	3.94E-04	0.57	1.11E-04	0.43
320	5.10E-03	1.48	1.19E-03	1.44	2.79E-04	0.50	8.52E-05	0.38
$\alpha = 0.5$								
10	1.10E-01	-	3.92E-02	-	4.92E-03	-	1.98E-03	-
20	3.79E-02	1.54	1.15E-02	1.77	3.45E-04	3.84	1.45E-04	3.77
40	1.23E-02	1.62	3.47E-03	1.73	2.69E-05	3.68	1.32E-05	3.46
80	3.97E-03	1.63	1.08E-03	1.68	2.46E-06	3.45	1.40E-06	3.23
160	1.30E-03	1.61	3.51E-04	1.63	2.56E-07	3.27	1.63E-07	3.11
320	4.34E-04	1.58	1.16E-04	1.59	2.90E-08	3.14	1.96E-08	3.05
$\alpha = 1.25$								
10	2.38E-02	-	9.77E-03	-	3.06E-03	-	1.45E-03	-
20	5.68E-03	2.07	2.14E-03	2.19	4.00E-04	2.93	1.80E-04	3.01
40	1.36E-03	2.06	4.96E-04	2.11	5.12E-05	2.97	2.24E-05	3.01
80	3.31E-04	2.05	1.19E-04	2.06	6.48E-06	2.98	2.79E-06	3.00
160	8.07E-05	2.03	2.89E-05	2.04	8.15E-07	2.99	3.49E-07	3.00
320	1.98E-05	2.03	7.11E-06	2.02	1.02E-07	3.00	4.36E-08	3.00

A.3 Example Equation 4 Results

A.3.1 Expected Results, from [1]

Table A.9: Expected results using Example Equation 4, Eq. (3.8), as reported in [1].

$\alpha = 0.3$				
MF-PCQ				
N	E_{pt}	Order	E_{L^2}	Order
10	8.44E-04	-	4.77E-03	-
20	4.34E-06	7.6	1.21E-03	1.97
40	4.71E-07	3.2	3.23E-04	1.91
80	6.40E-08	2.88	8.88E-05	1.87
160	7.05E-08	0.14	2.51E-05	1.82
320	3.38E-08	1.06	7.30E-06	1.78
$\alpha = 0.5$				
10	1.03E-04	-	3.25E-03	-
20	5.78E-06	4.15	8.29E-04	1.97
40	3.05E-06	0.92	2.30E-04	1.85
80	1.88E-06	0.7	6.84E-05	1.75
160	8.12E-07	1.21	2.16E-05	1.67
320	3.16E-07	1.36	7.10E-06	1.6
$\alpha = 1.25$				
10	3.02E-04	-	2.81E-04	-
20	8.62E-05	1.81	8.72E-05	1.69
40	2.48E-05	1.8	2.68E-05	1.7
80	7.21E-06	1.78	8.11E-06	1.72
160	2.12E-06	1.77	2.44E-06	1.74
320	6.25E-07	1.76	7.28E-07	1.74

A.3.2 Independent Simulation Results

Table A.10: Independent simulation results of ABM vs MF-PCL using Example Equation 4, Eq. (3.8).

$\alpha = 0.3$								
	ABM				MF-PCL			
N	E_{pt}	Order	E_{L^2}	Order	E_{pt}	Order	E_{L^2}	Order
10	3.14E-02	-	2.01E-02	-	3.56E-03	-	5.99E-03	-
20	1.10E-02	1.52	6.88E-03	1.55	6.04E-04	2.56	1.38E-03	2.12
40	3.91E-03	1.49	2.40E-03	1.52	1.20E-04	2.33	3.54E-04	1.96
80	1.42E-03	1.46	8.59E-04	1.48	2.64E-05	2.19	1.01E-04	1.80
160	5.26E-04	1.43	3.16E-04	1.44	5.96E-06	2.14	3.19E-05	1.67
320	1.98E-04	1.41	1.19E-04	1.41	1.19E-06	2.33	1.07E-05	1.57
$\alpha = 0.5$								
10	1.44E-02	-	8.58E-03	-	2.74E-03	-	5.30E-03	-
20	4.52E-03	1.68	2.55E-03	1.75	7.15E-04	1.94	1.80E-03	1.56
40	1.46E-03	1.63	8.20E-04	1.64	2.10E-04	1.77	6.58E-04	1.45
80	4.81E-04	1.60	2.85E-04	1.52	6.51E-05	1.69	2.49E-04	1.40
160	1.62E-04	1.57	1.05E-04	1.44	2.09E-05	1.64	9.54E-05	1.38
320	5.52E-05	1.55	3.96E-05	1.41	6.86E-06	1.61	3.66E-05	1.38
$\alpha = 1.25$								
10	6.74E-04	-	1.32E-03	-	2.41E-03	-	2.32E-03	-
20	3.63E-04	0.89	5.21E-04	1.34	7.05E-04	1.77	7.25E-04	1.68
40	1.43E-04	1.35	1.84E-04	1.50	2.12E-04	1.73	2.25E-04	1.69
80	5.00E-05	1.51	6.12E-05	1.59	6.43E-05	1.72	6.98E-05	1.69
160	1.65E-05	1.60	1.97E-05	1.63	1.95E-05	1.72	2.15E-05	1.70
320	5.28E-06	1.65	6.22E-06	1.67	5.90E-06	1.72	6.59E-06	1.71

Table A.11: Independent simulation results of MF-PCQ using Example Equation 4, Eq. (3.8).

$\alpha = 0.3$				
MF-PCQ				
N	E_{pt}	Order	E_{L^2}	Order
10	8.41E-04	-	4.77E-03	-
20	2.87E-06	8.20	1.21E-03	1.97
40	6.29E-07	2.19	3.23E-04	1.91
80	9.02E-07	-0.52	8.87E-05	1.87
160	7.18E-07	0.33	2.51E-05	1.82
320	5.40E-07	0.41	7.29E-06	1.78
$\alpha = 0.5$				
10	1.03E-04	-	3.25E-03	-
20	5.78E-06	4.15	8.29E-04	1.97
40	3.05E-06	0.92	2.30E-04	1.85
80	1.88E-06	0.70	6.84E-05	1.75
160	8.12E-07	1.21	2.16E-05	1.67
320	3.16E-07	1.36	7.10E-06	1.60
$\alpha = 1.25$				
10	3.02E-04	-	2.81E-04	-
20	8.62E-05	1.81	8.72E-05	1.69
40	2.48E-05	1.80	2.68E-05	1.70
80	7.21E-06	1.78	8.11E-06	1.72
160	2.12E-06	1.77	2.44E-06	1.74
320	6.25E-07	1.76	7.28E-07	1.74

APPENDIX B

SOFTWARE SOURCE CODE

B.1 Method Source Code

```
function [T,Y] = MF_PCL(alpha,fdefun,t0,tfinal,y0,h)
%MF_PCL   Solves an initial value problem for a non-linear
 ↪  differential
%         equation of fractional order (FDE). The code
 ↪  implements the
%         2nd-order linear interpolation predictor-corrector
 ↪  PECE method of [1].
%
%   [T,Y] = MF_PCL(ALPHA,FDEFUN,T0,TFINAL,Y0,h) integrates the
 ↪  initial value
%   problem for the FDE, or the system of FDEs, of order ALPHA >
 ↪  0
%       D^ALPHA Y(t) = FDEFUN(T,Y(T))
%       Y^(k)(0) = y0, k=0,...,m-1
%   where m is the smallest integer greater than ALPHA and
 ↪  D^ALPHA is the
%   fractional derivative according to the Caputo's definition.
%INPUTS:
%   (1) alpha: The order of the differential equation. Alpha
 ↪  must be >0.
%   (2) fdefun: Fractional Differential Equation. Must be a
 ↪  function handle
```

```
%   in the form of fdefun = @(T,Y) ....
%   (3) t0: Initial time. t0 must be a scalar
%   (4) tfinal: Final time. tfinal must be a scalar
%   (5) y0: Initial conditions. See below for info on y0*
%   (6) h: step-size. h must be a scalar value > 0.
%OUTPUTS:
%   (1) T: Time. T should be a vector ranging from t0:h:tfinal
%   (2) Y: Solution to the FDE. Y should be the same length as
 ↪  T, as
%   defined above.
%
% *The set of initial conditions y0 is a matrix with a number of
 ↪  rows equal to
%   the size of the problem (hence equal to the number of rows
 ↪  of the
%   output of FDEFUN) and a number of columns depending on ALPHA
 ↪  and given
%   by m.
%
%[1] T. B. Nugyen and B. Jang, ŞA high-order predictor-corrector
 ↪  method for
%    solving nonlinear differential equations of fractional
 ↪  order,Ť
%    Fractional Calculus and Applied Analysis, vol. 20, No. 2,
 ↪  pp. 447Ű476, 2017.
```

```
%% Error Catching
%Verify that the final time is greater than initial time.
if tfinal <= t0
    error('The initial time, t0, must be less than the final
     ↪  time, tfinal.')
end

%Verify alpha is a positive value.
if alpha <= 0
    error('Alpha must be a positive value')
end

%Check for required number of initial conditions
num_init_conds_req = ceil(alpha);
if size(y0,2) <= num_init_conds_req
    error('y0 does not contain sufficient initial condition
     ↪  information to solve the problem. y0 should be a row
     ↪  vector of length equal to ceil(alpha). (E.G. if alpha =
     ↪  1.25, y0 = [y(0), y_prime(0)])')
end

% Check that fdefun is a function handle.
if isa(fdefun, 'function_handle') == 0
    error('fdefun input is not a function handle')
end
```

```
% Verify that the grid is compatible with the method.
tfinal_on_grid_check = (tfinal-t0)/h;
if mod(tfinal_on_grid_check, 1) ~= 0
    error('grid does not end on tfinal. This method requires a
     ↪  constant step size. Please reconfigure your t0, tfinal,
     ↪  and h values such that (tfinal-t0)/h is an integer
     ↪  value.')
end



%% Internal Variable Initialization
b0 = -1;
b1 = alpha + 2;

%Preallocate Variable Arrays for Speed
num_terms = (tfinal-t0)/h;

y_corrector = zeros(1, num_terms);
y_predictor = zeros(1, num_terms);
f_predictor = zeros(1, num_terms);
f_corrector = zeros(1, num_terms);
g = zeros(1, num_terms);
y_lag_final = zeros(1, num_terms);
y_increment = zeros(1, num_terms);

%% Initialize Counters:
```

```
% NOTE: Since the step can be a non-integer value, (i.e. n =
% t0:h:tfinal), we need an integer variable corresponding to
 ↪  this step for
% array indexing purposes. These are defined below:
% 1) jj: This is the indexing variable for lag calculations. It
 ↪  corresponds
%        with the step j, where j is summed from 0 to n-1
% 2) nn: This is the indexing variable for the method. It
 ↪  corresponds with
% the step n, which is the current timestep.

%Again, since the startup takes us through 2*h, we have the
 ↪  following:
%nn = 1: y(0)
%nn = 2: y(h)
%nn = 3: y(2h), with y(nn+1) being the first unknown term of the
 ↪  algorithm.
nn = 3;
starting_step = (2*h + t0);

% Startup Export contains the values needed from the
 ↪  StartupProcedure. As
% defined below, StartupProcedure contains:
% 1) startup_export(1) = f_corrector evaluated at t = h
% 2) startup_export(2) = f_corrector evaluated at t = 2h
% 3) startup_export(3) = y_corrector evaluated at t = h
% 4) startup_export(4) = y_corrector evaluated at t = 2h
```

```
scheme = 'linear';
startup_export = StartupProcedure(fdefun, scheme, t0, y0, alpha,
 ↪  h);
f_corrector(1) = fdefun(t0, y0(1));
f_corrector(2) = startup_export(1);
f_corrector(3) = startup_export(2);

y_corrector(1) = y0(1);
y_corrector(2) = startup_export(3);
y_corrector(3) = startup_export(4);

%Calculate Coefficients for Linear Interp method. Will remain a
 ↪  constant
%value over the full method, so can be calculated prior to the
 ↪  loop.
[B_0n_Coeff, B_1n_Coeff] =
 ↪  B_Coefficients(starting_step,starting_step+h,
 ↪  starting_step+h, alpha);

g(1) = calculate_IC(y0,alpha,0);

%% 2nd Order, Linear Interpolation

% n is the current time step. Starts at 2*h, and increases by h
 ↪  each time.
% This is because the startup procedure calculates up through
 ↪  2*h, so the
```

```
% next y_(n+1) term desired is y_(2*h + h). % This will
 ↪  calculate values up
% to final time tfinal, since each loop solves the y_(n+1) term.
for n = starting_step:h:tfinal-h

    %lag term incrementor and lag summer reset for each pass of
     ↪  n.
    jj = 1;
    y_lag_sum = 0;

    for j = 0:h:n-h
        %Calculate Coefficients for lag terms:
        [B_0j_Coeff, B_1j_Coeff] = B_Coefficients(j,j+h, n+h,
         ↪  alpha);

        % Summation Portion of the lag term.
        y_lag_sum = y_lag_sum + B_0j_Coeff*f_corrector(jj) +
         ↪  B_1j_Coeff*f_corrector(jj+1);

        %Increment the interior counter
        jj = jj + 1;
    end

    %Calculate the initial condition term
    g(nn+1) = calculate_IC(y0, alpha, n+h);

    % Lag term for this time step is now complete!
```

```
y_lag_final(nn+1) = 1/gamma(alpha)*y_lag_sum;


%Now we are able to calculate the predictor:

% Predictor term at the new step:
y_predictor(nn+1) = g(nn+1) + y_lag_final(nn+1) +
 ↪  h^alpha/gamma(alpha+2)*(b0*f_corrector(nn-1) +
 ↪  b1*f_corrector(nn));

%Evaluate the FDEfun using this y_predicted value, at the
 ↪  current time
%step that we are looking to predict at.
f_predictor(nn+1) = fdefun(n+h, y_predictor(nn+1));

%Increment term at the new step
y_increment(nn+1) =
 ↪  1/gamma(alpha)*(B_0n_Coeff*f_corrector(nn) +
 ↪  B_1n_Coeff*f_predictor(nn+1));

%Corrector term at the new step, using same lag term, and
 ↪  above
%increment term
y_corrector(nn+1) = g(nn+1) + y_lag_final(nn+1) +
 ↪  y_increment(nn+1);
```

```
    %Evaluate fdefun using corrector y_value guess instead of
     ↪  predicted y_value. To
    %be used in the next prediction stage.
    f_corrector(nn+1) = fdefun(n+h, y_corrector(nn+1));

    %increment n indexing variable.
    nn = nn + 1;

end

%return the final output variables
Y = y_corrector;
T = t0:h:tfinal;
%% END of Method
end

function [T,Y] = MF_PCL(alpha,fdefun,t0,tfinal,y0,h)
%MF_PCL   Solves an initial value problem for a non-linear
 ↪  differential
%        equation of fractional order (FDE). The code
 ↪  implements the
%        2nd-order linear interpolation predictor-corrector
 ↪  PECE method of [1].
%
%   [T,Y] = MF_PCL(ALPHA,FDEFUN,T0,TFINAL,Y0,h) integrates the
 ↪  initial value
%   problem for the FDE, or the system of FDEs, of order ALPHA >
 ↪  0
```

```
%       D^ALPHA Y(t) = FDEFUN(T,Y(T))
%       Y^(k)(0) = y0, k=0,...,m-1
%   where m is the smallest integer greater than ALPHA and
↪  D^ALPHA is the
%   fractional derivative according to the Caputo's definition.
%INPUTS:
%   (1) alpha: The order of the differential equation. Alpha
↪  must be >0.
%   (2) fdefun: Fractional Differential Equation. Must be a
↪  function handle
%   in the form of fdefun = @(T,Y) ....
%   (3) t0: Initial time. t0 must be a scalar
%   (4) tfinal: Final time. tfinal must be a scalar
%   (5) y0: Initial conditions. See below for info on y0*
%   (6) h: step-size. h must be a scalar value > 0.
%OUTPUTS:
%   (1) T: Time. T should be a vector ranging from t0:h:tfinal
%   (2) Y: Solution to the FDE. Y should be the same length as
↪  T, as
%   defined above.
%
% *The set of initial conditions y0 is a matrix with a number of
↪  rows equal to
%   the size of the problem (hence equal to the number of rows
↪  of the
%   output of FDEFUN) and a number of columns depending on ALPHA
↪  and given
```

```
%   by m.
%
%[1] T. B. Nugyen and B. Jang, ŞA high-order predictor-corrector
 ↪  method for
%    solving nonlinear differential equations of fractional
 ↪  order,Ť
%    Fractional Calculus and Applied Analysis, vol. 20, No. 2,
 ↪  pp. 447Ű476, 2017.



%% Error Catching
%Verify that the final time is greater than initial time.
if tfinal <= t0
    error('The initial time, t0, must be less than the final
     ↪  time, tfinal.')
end

%Verify alpha is a positive value.
if alpha <= 0
    error('Alpha must be a positive value')
end

%Check for required number of initial conditions
num_init_conds_req = ceil(alpha);
if size(y0,2) <= num_init_conds_req
```

```
    error('y0 does not contain sufficient initial condition
     ↪ information to solve the problem. y0 should be a row
     ↪ vector of length equal to ceil(alpha). (E.G. if alpha =
     ↪ 1.25, y0 = [y(0), y_prime(0)])')
end

% Check that fdefun is a function handle.
if isa(fdefun, 'function_handle') == 0
    error('fdefun input is not a function handle')
end

% Verify that the grid is compatible with the method.
tfinal_on_grid_check = (tfinal-t0)/h;
if mod(tfinal_on_grid_check, 1) ~= 0
    error('grid does not end on tfinal. This method requires a
     ↪ constant step size. Please reconfigure your t0, tfinal,
     ↪ and h values such that (tfinal-t0)/h is an integer
     ↪ value.')
end



%% Internal Variable Initialization
b0 = -1;
b1 = alpha + 2;

%Preallocate Variable Arrays for Speed
```

```
num_terms = (tfinal-t0)/h;

y_corrector = zeros(1, num_terms);
y_predictor = zeros(1, num_terms);
f_predictor = zeros(1, num_terms);
f_corrector = zeros(1, num_terms);
g = zeros(1, num_terms);
y_lag_final = zeros(1, num_terms);
y_increment = zeros(1, num_terms);

%% Initialize Counters:
% NOTE: Since the step can be a non-integer value, (i.e. n =
% t0:h:tfinal), we need an integer variable corresponding to
 ↪  this step for
% array indexing purposes. These are defined below:
% 1) jj: This is the indexing variable for lag calculations. It
 ↪  corresponds
%        with the step j, where j is summed from 0 to n-1
% 2) nn: This is the indexing variable for the method. It
 ↪  corresponds with
% the step n, which is the current timestep.

%Again, since the startup takes us through 2*h, we have the
 ↪  following:
%nn = 1: y(0)
%nn = 2: y(h)
```

```
%nn = 3: y(2h), with y(nn+1) being the first unknown term of the
 ↪  algorithm.
nn = 3;
starting_step = (2*h + t0);

% Startup Export contains the values needed from the
 ↪  StartupProcedure. As
% defined below, StartupProcedure contains:
% 1) startup_export(1) = f_corrector evaluated at t = h
% 2) startup_export(2) = f_corrector evaluated at t = 2h
% 3) startup_export(3) = y_corrector evaluated at t = h
% 4) startup_export(4) = y_corrector evaluated at t = 2h
scheme = 'linear';
startup_export = StartupProcedure(fdefun, scheme, t0, y0, alpha,
 ↪  h);
f_corrector(1) = fdefun(t0, y0(1));
f_corrector(2) = startup_export(1);
f_corrector(3) = startup_export(2);

y_corrector(1) = y0(1);
y_corrector(2) = startup_export(3);
y_corrector(3) = startup_export(4);

%Calculate Coefficients for Linear Interp method. Will remain a
 ↪  constant
%value over the full method, so can be calculated prior to the
 ↪  loop.
```

```
[B_0n_Coeff, B_1n_Coeff] =
 ↪  B_Coefficients(starting_step,starting_step+h,
 ↪  starting_step+h, alpha);

g(1) = calculate_IC(y0,alpha,0);

%% 2nd Order, Linear Interpolation

% n is the current time step. Starts at 2*h, and increases by h
 ↪  each time.
% This is because the startup procedure calculates up through
 ↪  2*h, so the
% next y_(n+1) term desired is y_(2*h + h). % This will
 ↪  calculate values up
% to final time tfinal, since each loop solves the y_(n+1) term.
for n = starting_step:h:tfinal-h

    %lag term incrementor and lag summer reset for each pass of
     ↪  n.
    jj = 1;
    y_lag_sum = 0;

    for j = 0:h:n-h
        %Calculate Coefficients for lag terms:
        [B_0j_Coeff, B_1j_Coeff] = B_Coefficients(j,j+h, n+h,
         ↪  alpha);
```

```
    % Summation Portion of the lag term.
    y_lag_sum = y_lag_sum + B_0j_Coeff*f_corrector(jj) +
     ↪  B_1j_Coeff*f_corrector(jj+1);

    %Increment the interior counter
    jj = jj + 1;
end

%Calculate the initial condition term
g(nn+1) = calculate_IC(y0, alpha, n+h);

% Lag term for this time step is now complete!
y_lag_final(nn+1) = 1/gamma(alpha)*y_lag_sum;


%Now we are able to calculate the predictor:

% Predictor term at the new step:
y_predictor(nn+1) = g(nn+1) + y_lag_final(nn+1) +
 ↪  h^alpha/gamma(alpha+2)*(b0*f_corrector(nn-1) +
 ↪  b1*f_corrector(nn));

%Evaluate the FDEfun using this y_predicted value, at the
 ↪  current time
%step that we are looking to predict at.
f_predictor(nn+1) = fdefun(n+h, y_predictor(nn+1));
```

```
    %Increment term at the new step
    y_increment(nn+1) =
     ↪  1/gamma(alpha)*(B_0n_Coeff*f_corrector(nn) +
     ↪  B_1n_Coeff*f_predictor(nn+1));

    %Corrector term at the new step, using same lag term, and
     ↪  above
    %increment term
    y_corrector(nn+1) = g(nn+1) + y_lag_final(nn+1) +
     ↪  y_increment(nn+1);

    %Evaluate fdefun using corrector y_value guess instead of
     ↪  predicted y_value. To
    %be used in the next prediction stage.
    f_corrector(nn+1) = fdefun(n+h, y_corrector(nn+1));

    %increment n indexing variable.
    nn = nn + 1;

end

%return the final output variables
Y = y_corrector;
T = t0:h:tfinal;
%% END of Method
end

function [T,Y] = MF_PCQ(alpha,fdefun,t0,tfinal,y0,h)
```

```
%MF_PCQ   Solves an initial value problem for a non-linear
 ↪  differential
%         equation of fractional order (FDE). The code
 ↪  implements the
%         3rd-order quadratic interpolation predictor-corrector
 ↪  PECE method of [1].
%
%   [T,Y] = MF_PCQ(ALPHA,FDEFUN,T0,TFINAL,Y0,h) integrates the
 ↪  initial value
%   problem for the FDE, or the system of FDEs, of order ALPHA >
 ↪  0
%      D^ALPHA Y(t) = FDEFUN(T,Y(T))
%      Y^(k)(0) = y0, k=0,...,m-1
%   where m is the smallest integer greater than ALPHA and
 ↪  D^ALPHA is the
%   fractional derivative according to the Caputo's definition.
%INPUTS:
%   (1) alpha: The order of the differential equation. Alpha
 ↪  must be >0.
%   (2) fdefun: Fractional Differential Equation. Must be a
 ↪  function handle
%   in the form of fdefun = @(T,Y) ....
%   (3) t0: Initial time. t0 must be a scalar
%   (4) tfinal: Final time. tfinal must be a scalar
%   (5) y0: Initial conditions. See below for info on y0*
%   (6) h: step-size. h must be a scalar value > 0.
%OUTPUTS:
```

```
%   (1) T: Time. T should be a vector ranging from t0:h:tfinal
%   (2) Y: Solution to the FDE. Y should be the same length as
 ↪  T, as
%   defined above.
%
% *The set of initial conditions y0 is a matrix with a number of
 ↪  rows equal to
%   the size of the problem (hence equal to the number of rows
 ↪  of the
%   output of FDEFUN) and a number of columns depending on ALPHA
 ↪  and given
%   by m.
%
%[1] T. B. Nugyen and B. Jang, ŞA high-order predictor-corrector
 ↪  method for
%    solving nonlinear differential equations of fractional
 ↪  order,Ť
%    Fractional Calculus and Applied Analysis, vol. 20, No. 2,
 ↪  pp. 447Ű476, 2017.


%% Error Catching
%Verify that the final time is greater than initial time.
if tfinal <= t0
    error('The initial time, t0, must be less than the final
     ↪  time, tfinal.')
end
```

```
%Verify alpha is a positive value.
if alpha <= 0
    error('Alpha must be a positive value')
end

%Check for required number of initial conditions
num_init_conds_req = ceil(alpha);
if size(y0,2) <= num_init_conds_req
    error('y0 does not contain sufficient initial condition
     ↪  information to solve the problem. y0 should be a row
     ↪  vector of length equal to ceil(alpha). (E.G. if alpha =
     ↪  1.25, y0 = [y(0), y_prime(0)])')
end

% Check that fdefun is a function handle.
if isa(fdefun, 'function_handle') == 0
    error('fdefun input is not a function handle')
end

% Verify that the grid is compatible with the method.
tfinal_on_grid_check = (tfinal-t0)/h;
if mod(tfinal_on_grid_check, 1) ~= 0
    error('grid does not end on tfinal. This method requires a
     ↪  constant step size. Please reconfigure your t0, tfinal,
     ↪  and h values such that (tfinal-t0)/h is an integer
     ↪  value.')
```

```
end

%% Internal Variable Initialization
a0 = (alpha + 4)/2;
a1 = -2*(alpha+3);
a2 = (2*alpha^2 + 9*alpha + 12)/2;
m = 1; % TODO: should be calculated later with IC

%% Initialize Counters:
% NOTE: Since the step can be a non-integer value, (i.e. n =
% t0:h:tfinal), we need an integer variable corresponding to
 ↪  this step for
% array indexing purposes. These are defined below:
% 1) jj: This is the indexing variable for lag calculations. It
 ↪  corresponds
%        with the step j, where j is summed from 0 to n-1
% 2) nn: This is the indexing variable for the method. It
 ↪  corresponds with
% the step n, which is the current timestep.

%Again, since the startup takes us through 2*h, we have the
 ↪  following:
%nn = 1: y(0)
%nn = 2: y(h)
%nn = 3: y(2h), with y(nn+1) being the first unknown term of the
 ↪  algorithm.
nn = 3;
```

```
starting_step = 2*h + t0;

% Startup Export contains the values needed from the
 ↪  startupProcedure. As
% defined below, StartupProcedure contains:
% 1) startup_export(1) = f_corrector evaluated at t = h/2
% 2) startup_export(2) = f_corrector evaluated at t = h
% 3) startup_export(3) = f_corrector evaluated at t = 2h
% 4) startup_export(4) = y_corrector evaluated at t = h/2
% 5) startup_export(5) = y_corrector evaluated at t = h
% 6) startup_export(5) = y_corrector evaluated at t = 2h
scheme = 'quadratic';
startup_export = StartupProcedure(fdefun, scheme, t0, y0, alpha,
 ↪  h);
f_corrector(1) = fdefun(t0, y0(1));
f_corrector(2) = startup_export(2);
f_corrector_one_half = startup_export(1);
f_corrector(3) = startup_export(3);

y_corrector(1) = y0(1);
y_corrector(2) = startup_export(5);
y_corrector_one_half = startup_export(4);
y_corrector(3) = startup_export(6);

%Calculate Initial Condition
g(1) =  calculate_IC(y0, alpha, 0);
```

```
[A_0n_Coeff, A_1n_Coeff, A_2n_Coeff] =
 ↪  A_Coefficients(starting_step, starting_step+h,
 ↪  starting_step-h, starting_step, starting_step+h,
 ↪  starting_step+h, alpha);


%% 3rd Order, Quadratic Interp

% n is the current time step. Starts at 2*h, and increases by h
 ↪  each time.
% This is because the startup procedure calculates up through
 ↪  2*h, so the
% next y_(n+1) term desired is y_(2*h + h). % This will
 ↪  calculate values up
% to final time tfinal, since each loop solves the y_(n+1) term.
for n = starting_step:h:tfinal-h

      %lag term incrementor and lag summer reset for each pass
       ↪  of n.
    jj = 2;
    y_lag_sum = 0;

    for j = h:h:n-h
        %Calculate Coefficients for lag terms:
        [A_0j_Coeff, A_1j_Coeff, A_2j_Coeff] = A_Coefficients(j,
         ↪  j+h,j-h, j, j+h, n+h, alpha);
```

```
    % Summation Portion of the lag term.
    y_lag_sum = y_lag_sum + A_0j_Coeff*f_corrector(jj-1) +
     ↪  A_1j_Coeff*f_corrector(jj) +
     ↪  A_2j_Coeff*f_corrector(jj+1);

    %Increment the interior counter
    jj = jj + 1;
end

%Calculate the initial condition term
g(nn+1) = calculate_IC(y0, alpha, n+h);


%Compute the initial set of coefficients, evaluated at n+h.
[A_00_Coeff, A_10_Coeff, A_20_Coeff] =
 ↪  A_Coefficients(0,h,0,h/2,h,n+h, alpha);

% Apply Coefficient weights to the fde evaluated at
 ↪  respective time
% steps
A_initial = A_00_Coeff*f_corrector(1) +
 ↪  A_10_Coeff*f_corrector_one_half +
 ↪  A_20_Coeff*f_corrector(2);

% Lag term for this time step is now complete!
y_lag_final(nn+1) = 1/gamma(alpha)*(A_initial + y_lag_sum);
```

```
%Now we are able to calculate the predictor:

% Predictor term at the new step.
y_predictor(nn+1) = g(nn+1) + y_lag_final(nn+1) +
 ↪  h^alpha/gamma(alpha+3)*(a0*f_corrector(nn-2) +
 ↪  a1*f_corrector(nn-1) + a2*f_corrector(nn));

%Evaluate the FDEfun using this y_predicted value, at the
 ↪  current time
%step we are looking to predict at.
f_predictor(nn+1) = fdefun(n+h, y_predictor(nn+1));

%Increment term at the new step
y_increment(nn+1) =
 ↪  1/gamma(alpha)*(A_0n_Coeff*f_corrector(nn-1) +
 ↪  A_1n_Coeff*f_corrector(nn) +
 ↪  A_2n_Coeff*f_predictor(nn+1));

%Corrector term at the new step, using the same lag term,
 ↪  and  above
%increment term
y_corrector(nn+1) = g(nn+1) + y_lag_final(nn+1) +
 ↪  y_increment(nn+1);

%Evaluate using corrector y_value guess instead of predicted
 ↪  y_value. To
```

```
    %be used in the next prediction stage.
    f_corrector(nn+1) = feval(fdefun, n+h, y_corrector(nn+1));

    %increment n indexing variable
    nn = nn + 1;

end

%return the final output variables
Y = y_corrector;
T = t0:h:tfinal;
%% END of Method
end

function [B_0n_Coeff, B_1n_Coeff] = B_Coefficients(t_j, t_jj,
 ↪  t_nn, alpha)
%B_Coefficients is a supporting function in the FDE solver
 ↪  packages MF_PCL/MF_PCQ.
%This function computes the coefficients as defined in [1].

%INPUTS:
% (1) t_j:   (scalar) lower bound of integration
% (2) t_jj:  (scalar) upper bound of integration, t_jj>t_j
% (3) t_nn:  (scalar) current time-step the coefficient is to be
 ↪  evaluated at
% (4) alpha: (scalar) order of the FDE. alpha > 0.
%OUTPUTS:
% (1) B_0n_Coeff: (scalar) Coefficient calculated using L_j
```

```
% (2) B_1n_Coeff: (scalar) Coefficient calculated using L_jj
%
%%[1] T. B. Nugyen and B. Jang, ŞA high-order
 ↪  predictor-corrector method for
%    solving nonlinear differential equations of fractional
 ↪  order,Ť
%    Fractional Calculus and Applied Analysis, vol. 20, No. 2,
 ↪  pp. 447Ű476, 2017.

% Call the linear Lagrangian interpolation function
[L_j, L_jj] = Linear_Interp(t_j,t_jj);

%The core of the integral, this portion is the same for both B_0
 ↪  and B_1.
B_kernel = @(tau) ((t_nn-tau).^(alpha-1));

%These lines combine the kernel with each appropriate
 ↪  interpolation
%function, combining both function handles into one:
B_0n_funct = @(tau) B_kernel(tau).*L_j(tau);
B_1n_funct = @(tau) B_kernel(tau).*L_jj(tau);

%These lines compute the integral of the overall function, using
 ↪  the bounds
%input:
B_0n_Coeff = integral(B_0n_funct, t_j,t_jj);
B_1n_Coeff = integral(B_1n_funct, t_j,t_jj);
```

```
end

function [A_0n_Coeff, A_1n_Coeff, A_2n_Coeff] = A_Coefficients(i
↪  ntegrand_low,integrand_high,t_low,t_mid,t_up,t_eval,alpha)
%A_Coefficients is a supporting function in the FDE solver
↪  packages MF_PCL/MF_PCQ.
%This function computes the coefficients as defined in [1].
%
%This function has additional inputs as compared to
↪  B_Coefficients due to
%additional flexibility required by the startup scheme,
↪  StartupProcedure.
%
%INPUTS:
% (1) integrand_low:   (scalar) lower bound of integration
% (2) integrand_high:  (scalar) upper bound of integration,
↪  t_high>t_low
% (3) t_low:  (scalar) low value used by the quadratic
↪  interpolator
% (4) t_mid: (scalar) middle value used by the quadratic
↪  interpolator
% (5) t_up: (scalar) upper value used by the quadratic
↪  interpolator
% (6) t_eval: current time-step the coefficient is to be
↪  evaluated at
% (7) alpha: (scalar) order of the FDE. alpha > 0.
%OUTPUTS:
```

```
% (1) A_0n_Coeff: (scalar) Coefficient calculated using Q_low
% (2) A_1n_Coeff: (scalar) Coefficient calculated using Q_mid
% (3) A_2n_Coeff: (scalar) Coefficient calculated using Q_up
%
%%[1] T. B. Nugyen and B. Jang, ŞA high-order
 ↪  predictor-corrector method for
%    solving nonlinear differential equations of fractional
 ↪  order,Ť
%    Fractional Calculus and Applied Analysis, vol. 20, No. 2,
 ↪  pp. 447Ű476, 2017.

% Call the quadratic Lagrangian interpolation function
[Q_low, Q_mid, Q_up] = Quadratic_Interp(t_low, t_mid, t_up);

%The core of the integral, this portion is the same for A_0,
 ↪  A_1, and A_2.
A_kernel = @(tau) ((t_eval-tau).^(alpha-1));

%These lines combine the kernel with each appropriate
 ↪  interpolation
%function, combining both function handles into one:
A_0n_funct = @(tau) A_kernel(tau).*Q_low(tau);
A_1n_funct = @(tau) A_kernel(tau).*Q_mid(tau);
A_2n_funct = @(tau) A_kernel(tau).*Q_up(tau);

%These lines compute the integral of the overall function, using
 ↪  the bounds
```

```
%input:
A_0n_Coeff = integral(A_0n_funct,integrand_low,integrand_high);
A_1n_Coeff = integral(A_1n_funct,integrand_low,integrand_high);
A_2n_Coeff = integral(A_2n_funct,integrand_low,integrand_high);
end

function [L_j, L_jj] = Linear_Interp(t_j, t_jj)
% Linear_Interp is a supporting function for in FDE solver
 ↪  packages MF_PCL/MF_PCQ.
% This function implements a linear Lagrangian interpolation
 ↪  over the
% time-step from t_j to t_jj.
%INPUTS:
%    (1) t_j = (scalar number)  lower bound of interpolation
%    (2) t_jj = (scalar number) upper bound of interpolation (jj
 ↪  := j + 1)
%OUTPUTS:
%    (1) L_j = (function handle @(tau)... ) lower-end
 ↪  interpolation function
%    (2) L_jj = (function handle @(tau)... ) upper-end
 ↪  interpolation function

L_j  = @(tau) (tau - t_jj) / (t_j - t_jj);
L_jj = @(tau) (tau - t_j) / (t_jj - t_j);
end

function [Q_low, Q_mid, Q_up] =
 ↪  Quadratic_Interp(t_low,t_mid,t_up)
```

```
% Quadratic_Interp is a supporting function in the FDE solver
 ↪  packages MF_PCL/MF_PCQ.
% This function implements a quadratic Lagrangian interpolation
 ↪  over the
% time-step from t_low to t_high.
%INPUTS:
%   (1) t_low  = (scalar number)  lower bound of interpolation
%   (2) t_mid  = (scalar number) middle value of interpolation
%   (3) t_high = (scalar number) upper value of interpolation
%OUTPUTS:
%   (1) Q_low = (function handle @(tau)... ) lower-end
 ↪  interpolation function
%   (2) Q_mid = (function handle @(tau)... ) middle
 ↪  interpolation function
%   (3) Q_up = (function handle @(tau)... ) upper-end
 ↪  interpolation function

 Q_low = @(tau) ((tau - t_mid)/(t_low - t_mid)) .* ((tau -
  ↪  t_up)/(t_low - t_up));

 Q_mid = @(tau) ((tau - t_low)/(t_mid - t_low)) .* ((tau -
  ↪  t_up)/(t_mid - t_up));

 Q_up = @(tau) ((tau - t_low)/(t_up - t_low)) .* ((tau -
  ↪  t_mid)/(t_up - t_mid));

end
```

```
% This is a test script to accompany thr FDE solver packages,
% MF_PCL/MF_PCQ. This script will present an example of how to
 ↪  use the
% packages and present one of the example fde's present in [1].
%
%[1] T. B. Nugyen and B. Jang, ŞA high-order predictor-corrector
 ↪  method for
%    solving nonlinear differential equations of fractional
 ↪  order,Ť
%    Fractional Calculus and Applied Analysis, vol. 20, No. 2,
 ↪  pp. 447Ű476, 2017.
%
clear all;
close all;
clc;
%INPUTS:
alpha = 0.3;
fdefun = @(t,y) (40320/gamma(9-alpha))*t^(8-alpha) -
 ↪  3*(gamma(5+alpha/2)/gamma(5-alpha/2))*t^(4-alpha/2) +
 ↪  9/4*gamma(alpha+1) + (3/2*t^(alpha/2)-t^4)^3-y^(3/2);
t0 = 0;
t_final = 1;
y0 = [0,0];
tfinal = 1;
h = 0.05;

%Call the 2nd order method, MF_PCL:
```

```
[T,Y_PCL] = MF_PCL(alpha,fdefun,t0,tfinal,y0,h);

%Call the 3rd order method, MF_PCQ:
[T,Y_PCQ] = MF_PCQ(alpha,fdefun,t0,tfinal,y0,h);

%Call the original Method, ABM:
[T,Y_ABM] = fde12(alpha,fdefun,t0,tfinal,y0,h);

%Calculate True Solution:
i = 1;
    for t = t0:h:tfinal
        True_Solution(i) = t^8 - 3*t^(4+alpha/2)+9/4*t^alpha;
         ↪  %Change depending on Eq being used!
        i = i +1;
    end
figure(1)
title('Comparison of Competing Methods')
plot(T,Y_PCL);
hold on;
plot(T,Y_PCQ);
hold on;
plot(T,Y_ABM);
hold on;
plot(T,True_Solution);
legend('MF-PCL', 'MF-PCQ', 'ABM', 'Exact Solution')
xlabel('time, (T)')
ylabel('Solution, (Y)')
```

```
function IC = calculate_IC(y0, alpha, t_eval)
% calculate_IC is a supporting function in the FDE solver
 ↪  packages,
% MF_PCL/MF_PCQ. This function calculates the g term relating to
 ↪  the initial
% condition data. It is calculated according to [1].
%
%INPUTS:
%  (1) y0: (vector/ array of scalars) y0 should be of length m,
 ↪  defined as
%           m = ceiling(alpha). The terms of y0 are the values
 ↪  of [y(0),
%           y'(0), y''(0), ..., y^(m-1)] in this order.
%  (2) alpha: (scalar) order of the FDE. alpha > 0.
%  (3) t_eval: (scalar) the time that g is being evaluated at.
%OUTPUTS:
%  (1) IC: (scalar) value of g evaluated at the time specified.
%
% %%[1] T. B. Nugyen and B. Jang, ŞA high-order
 ↪  predictor-corrector method for
%    solving nonlinear differential equations of fractional
 ↪  order,Ť
%    Fractional Calculus and Applied Analysis, vol. 20, No. 2,
 ↪  pp. 447Ű476, 2017.

g = 0;
m = ceil(alpha);
```

```
for k = 0:m-1
    g = g + y0(k+1)*(t_eval^k)/factorial(k);
end

IC = g;
end

function [startup_export] = StartupProcedure(fdefun, scheme, t0,
 ↪  y0, alpha, h)
% StartupProcedure is a supporting function to the FDE solver
 ↪  packages
% MF_PCL/MF_PCQ. This function is called by both schemes to
 ↪  begin the
% method. It uses a combination of constant, linear, and
 ↪  quadratic
% interpolation to provide a high-accuracy early-heredity by
 ↪  computing
% values over a non-constant-sized grid of [0, h/4, h/2, h,
 ↪  2*h]. This
% function follows the definition as described in [1].
%
%INPUTS:
% (1) fdefun: (function handle @(T,Y) ...) The fractional
 ↪  differential equation.
% (2) scheme: (string) 'linear' or 'quadratic'. This determines
 ↪  which
```

```
%             values are returned from the function call,
 ↪  depending on which
%             scheme is being used.
% (3) t0:     (scalar) initial time. t0 must be >= 0.
% (4) y0:     (vector) The initial value of the problem.
% (5) alpha:  (scalar) order of the FDE. alpha > 0.
% (6) h:      (scalar) This is the step-size of the problem. The
 ↪  startup
%             procedure will divide h into the grid described
 ↪  above. h >0.
%OUTPUTS:
% (1) startup_export: (vector) % array that exports the required
 ↪  values
%                     computed in the startup procedure:
%                     Linear scheme: [f_corrector(h),
 ↪  f_corrector(2h),
%                                     y_corrector(h),
 ↪  y_corrector_(2h)]
%                     Quadratic scheme: [f_corrector(h/2),
 ↪  f_corrector(h),
%                                        f_corrector(2h),
 ↪  y_corrector(h/2),
%                                        y_corrector(h),
 ↪  y_corrector_(2h)]
%
% %[1] T. B. Nugyen and B. Jang, ŞA high-order
 ↪  predictor-corrector method for
```

```
%    solving nonlinear differential equations of fractional
 ↪  order,Ť
%    Fractional Calculus and Applied Analysis, vol. 20, No. 2,
 ↪  pp. 447Ű476, 2017.


%% Initial Values
g = zeros(1,10);
%Preallocate Variable arrays
f_predictor = zeros(1, 5);
y_predictor = zeros(1, 5);
f_corrector = zeros(1,5);
y_corrector = zeros(1,5);
y_predictor_P1 = zeros(1,5);
y_predictor_P2 = zeros(1,5);
f_predictor_P1 = zeros(1,5);
f_predictor_P2 = zeros(1,5);



f_corrector(1) = fdefun( t0, y0(1));

%% Approximate y(h/4)
 ↪  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Predict y(h/4):
jj = 1;
g(jj+1) = calculate_IC(y0, alpha, t0+h/4);
```

```
y_predictor(jj+1) = g(jj+1) + 1/gamma(alpha +
 ↪  1)*(h/4)^alpha*f_corrector(jj);

%% Correct y(h/4)
%Intermediate Steps:
% 1. Calculate B coefficients
[B_0n_Coeff, B_1n_Coeff] = B_Coefficients(t0 + 0, t0 + h/4, t0 +
 ↪  h/4, alpha) ;
% 2. Calculate f_pred, using y_pred.
f_predictor(jj+1) = fdefun(t0 + h/4, y_predictor(jj+1)) ;

y_corrector(jj+1) = g(jj+1) +
 ↪  1/gamma(alpha)*(B_0n_Coeff*f_corrector(jj) +
 ↪  B_1n_Coeff*f_predictor(jj+1));

% Calculate f_corrector(h/4)
f_corrector(jj+1) = fdefun( t0 + h/4, y_corrector(jj+1));

jj = jj + 1;

%% Approximate y(h/2)
 ↪  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Next Step Indexing Key:
% jj + 1 = 3, corresponding to h/2.
%     jj = 2, corresponding to h/4.
% jj - 1 = 1, corresponding to 0.
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
 ↪  %%%%%%%%%%%
%Lag term:
%Intermediate Steps:
%1. Calculate B Coefficients
[B_0n_Coeff, B_1n_Coeff] = B_Coefficients(t0 + 0, t0 + h/4, t0 +
 ↪  h/2, alpha) ;

g(jj+1) = calculate_IC(y0, alpha, t0+h/2);

y_lag(jj+1) = 1/gamma(alpha)*(B_0n_Coeff*f_corrector(jj-1) +
 ↪  B_1n_Coeff*f_corrector(jj));
%% Predict y(h/2) Constant Interp:
y_predictor_P1(jj+1) = g(jj+1) + y_lag(jj+1) +
 ↪  1/gamma(alpha+1)*(h/4)^alpha*f_corrector(jj);

%% Predict y(h/2) Linear Interp:
%Intermediate Steps:
[B_0n_Coeff, B_1n_Coeff] = B_Coefficients(t0 + h/4, t0 + h/2, t0
 ↪  + h/2, alpha) ;

f_predictor_P1(jj+1) = fdefun(t0 + h/2, y_predictor_P1(jj+1));

y_predictor_P2(jj+1) = g(jj+1) + y_lag(jj+1) +
 ↪  1/gamma(alpha)*(B_0n_Coeff*f_corrector(jj) +
 ↪  B_1n_Coeff*f_predictor_P1(jj+1));
```

```
%% Correct y(h/2) : Quadratic Interp:
%Intermediate Steps:
[A_0n_Coeff, A_1n_Coeff, A_2n_Coeff] = A_Coefficients(t0 + 0,t0
 ↪  + h/2,t0 + 0,t0 + h/4,t0 + h/2,t0 + h/2,alpha);

f_predictor_P2(jj+1) = fdefun( t0 + h/2, y_predictor_P2(jj+1));

y_corrector(jj+1) = g(jj+1) +
 ↪  1/gamma(alpha)*(A_0n_Coeff*f_corrector(jj-1) +
 ↪  A_1n_Coeff*f_corrector(jj) +
 ↪  A_2n_Coeff*f_predictor_P2(jj+1));

f_corrector(jj+1) = fdefun( t0 + h/2, y_corrector(jj+1));

jj = jj +1;

%% Approximate y(h)
 ↪  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Next Step Indexing Key:
% jj + 1 = 4, corresponding to h.
%     jj = 3, corresponding to h/2.
% jj - 1 = 2, corresponding to h/4.
% jj - 2 = 1, corresponding to 0.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
 ↪  %%%%%%%%%%%
% Lag term:
```

```
[B_0n_Coeff, B_1n_Coeff] = B_Coefficients(t0 + 0, t0 + h/2, t0 +
 ↪  h, alpha) ;

g(jj+1) = calculate_IC(y0, alpha, t0+h);

y_lag(jj+1) = 1/gamma(alpha)*(B_0n_Coeff*f_corrector(jj-2) +
 ↪  B_1n_Coeff*f_corrector(jj));

%% Predict y(h) Constant Interp:
y_predictor_P1(jj+1) = g(jj+1) + y_lag(jj+1) +
 ↪  1/gamma(alpha+1)*(h/2)^alpha*f_corrector(jj);

%% Predict y(h) Linear Interp:
%Intermediate Steps:
[B_0n_Coeff, B_1n_Coeff] = B_Coefficients(t0 + h/2,t0 + h,t0 +
 ↪  h, alpha) ;

f_predictor_P1(jj+1) = fdefun(t0 + h, y_predictor_P1(jj+1));



y_predictor_P2(jj+1) = g(jj+1) + y_lag(jj+1) +
 ↪  1/gamma(alpha)*(B_0n_Coeff*f_corrector(jj) +
 ↪  B_1n_Coeff*f_predictor_P1(jj+1));

%% Correct y(h) : Quadratic Interp:
%Intermediate Steps:
```

```
[A_0n_Coeff, A_1n_Coeff, A_2n_Coeff] = A_Coefficients(t0 + 0,t0
 ↪  + h,t0 + 0,t0 + h/2,t0 + h,t0 + h,alpha);

f_predictor_P2(jj+1) = fdefun( t0 + h, y_predictor_P2(jj+1));


y_corrector(jj+1) = g(jj+1) +
 ↪  1/gamma(alpha)*(A_0n_Coeff*f_corrector(jj-2) +
 ↪  A_1n_Coeff*f_corrector(jj) +
 ↪  A_2n_Coeff*f_predictor_P2(jj+1));

f_corrector(jj+1) = fdefun(t0 + h, y_corrector(jj+1));

jj = jj +1;
%% Approximate y(2h)
 ↪  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Next Step Indexing Key:
% jj + 1 = 5, corresponding to 2h.
%     jj = 4, corresponding to h.
% jj - 1 = 3, corresponding to h/2.
% jj - 2 = 2, corresponding to h/4.
% jj - 3 = 1, corresponding to 0.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%⌋
 ↪  %%%%%%%%%%%
% Lag term:
[A_0n_Coeff, A_1n_Coeff, A_2n_Coeff] = A_Coefficients(t0 +0,t0 +
 ↪  h,t0 + 0, t0 + h/2,t0 + h,t0 + 2*h, alpha) ;
```

```
g(jj+1) = calculate_IC(y0, alpha, t0+2*h);

y_lag(jj+1) = 1/gamma(alpha)*(A_0n_Coeff*f_corrector(jj-3) +
 ↪  A_1n_Coeff*f_corrector(jj-1) + A_2n_Coeff*f_corrector(jj));

%% Predict y(2h) Constant Interp:
y_predictor_P1(jj+1) = g(jj+1) + y_lag(jj+1) +
 ↪  1/gamma(alpha+1)*(h)^alpha*f_corrector(jj);

%% Predict y(2h) Linear Interp:
%Intermediate Steps:
[B_0n_Coeff, B_1n_Coeff] = B_Coefficients(t0 + h, t0 + 2*h, t0 +
 ↪  2*h, alpha) ;

f_predictor_P1(jj+1) = fdefun( t0 + 2*h, y_predictor_P1(jj+1));



y_predictor_P2(jj+1) = g(jj+1) + y_lag(jj+1) +
 ↪  1/gamma(alpha)*(B_0n_Coeff*f_corrector(jj) +
 ↪  B_1n_Coeff*f_predictor_P1(jj+1));

%% Correct y(2h) : Quadratic Interp:
%Intermediate Steps:
[A_0n_Coeff, A_1n_Coeff, A_2n_Coeff] = A_Coefficients(t0 + h, t0
 ↪  + 2*h, t0 + 0, t0 + h,t0 + 2*h, t0 + 2*h,alpha);
```

```
f_predictor_P2(jj+1) = fdefun( t0 + 2*h, y_predictor_P2(jj+1));



y_corrector(jj+1) = g(jj+1) + y_lag(jj+1) +
 ↪  1/gamma(alpha)*(A_0n_Coeff*f_corrector(jj-3) +
 ↪  A_1n_Coeff*f_corrector(jj) +
 ↪  A_2n_Coeff*f_predictor_P2(jj+1));

f_corrector(jj+1) = fdefun( t0 + 2*h, y_corrector(jj+1));

% Linear vs Quadratic Startup:
%Linear scheme requires values at h and 2h, quadratic scheme
 ↪  requires
%values at h/2, h, and 2h.
if strcmp(scheme, 'linear') == true
    startup_export = [f_corrector(jj), f_corrector(jj+1),
     ↪  y_corrector(jj), y_corrector(jj+1)];
elseif strcmp(scheme, 'quadratic') == true
    startup_export = [f_corrector(jj-1), f_corrector(jj),
     ↪  f_corrector(jj+1), y_corrector(jj-1),  y_corrector(jj),
     ↪  y_corrector(jj+1)];
else
    error('Please enter a valid scheme for the startup algorithm
     ↪  to export the required values. The accepted inputs are
     ↪  "linear", "ABM" and "quadratic".')
end
end
```

```
function [T,Y] = ABM(alpha,fdefun,t0,tfinal,y0, h)
%ABM      Solves an initial value problem for a non-linear
 ↪  differential
%         equation of fractional order (FDE). The code
 ↪  implements the
%         Adams-Bashforth-Moulton method.
%
%   [T,Y] = ABM(ALPHA,FDEFUN,T0,TFINAL,Y0,h) integrates the
 ↪  initial value
%   problem for the FDE, or the system of FDEs, of order ALPHA >
 ↪  0
%      D^ALPHA Y(t) = FDEFUN(T,Y(T))
%      Y^(k)(0) = y0, k=0,...,m-1
%   where m is the smallest integer greater than ALPHA and
 ↪  D^ALPHA is the
%   fractional derivative according to the Caputo's definition.
%INPUTS:
%   (1) alpha: The order of the differential equation. Alpha
 ↪  must be >0.
%   (2) fdefun: Fractional Differential Equation. Must be a
 ↪  function handle
%   in the form of fdefun = @(T,Y) ....
%   (3) t0: Initial time. t0 must be a scalar
%   (4) tfinal: Final time. tfinal must be a scalar
%   (5) y0: Initial conditions. See below for info on y0*
%   (6) h: step-size. h must be a scalar value > 0.
%OUTPUTS:
```

```
%   (1) T: Time. T should be a vector ranging from t0:h:tfinal
%   (2) Y: Solution to the FDE. Y should be the same length as
 ↪  T, as
%   defined above.
%
% *The set of initial conditions y0 is a matrix with a number of
 ↪  rows equal to
%   the size of the problem (hence equal to the number of rows
 ↪  of the
%   output of FDEFUN) and a number of columns depending on ALPHA
 ↪  and given
%   by m.
%
%% Error Catching
%Verify that the final time is greater than initial time.
if tfinal <= t0
    error('The initial time, t0, must be less than the final
     ↪  time, tfinal.')
end

%Verify alpha is a positive value.
if alpha <= 0
    error('Alpha must be a positive value')
end

%Check for required number of initial conditions
num_init_conds_req = ceil(alpha);
```

```
if size(y0,2) < num_init_conds_req
    error('y0 does not contain sufficient initial condition
     ↪ information to solve the problem. y0 should be a row
     ↪ vector of length equal to ceil(alpha). (E.G. if alpha =
     ↪ 1.25, y0 = [y(0), y_prime(0)])')
end

% Check that fdefun is a function handle.
if isa(fdefun, 'function_handle') == 0
    error('fdefun input is not a function handle')
end

% Verify that the grid is compatible with the method.
tfinal_on_grid_check = (tfinal-t0)/h;
if mod(tfinal_on_grid_check, 1) ~= 0
    error('grid does not end on tfinal. This method requires a
     ↪ constant step size. Please reconfigure your t0, tfinal,
     ↪ and h values such that (tfinal-t0)/h is an integer
     ↪ value.')
end

%% Internal Variable Initialization
%Preallocate Variable Arrays for Speed
num_terms = (tfinal-t0)/h;
y_corrector = zeros(1, num_terms);
y_predictor = zeros(1, num_terms);
f_predictor = zeros(1, num_terms);
```

```
f_corrector = zeros(1, num_terms);
g = zeros(1, num_terms);
y_lag_final = zeros(1, num_terms);
y_increment = zeros(1, num_terms);
scheme = 'linear';
%% Initialize Counters:
% NOTE: Since the step can be a non-integer value, (i.e. n =
% t0:h:tfinal), we need an integer variable corresponding to
 ↪  this step for
% array indexing purposes. These are defined below:
% 1) jj: This is the indexing variable for lag calculations. It
 ↪  corresponds
%        with the step j, where j is summed from 0 to n-1
% 2) nn: This is the indexing variable for the method. It
 ↪  corresponds with
% the step n, which is the current timestep.
nn = 1;

f_corrector(1) = feval(fdefun, t0, y0(1));
y_corrector(1) = y0(1);

%Calculate Coefficients for Linear Interp method. Will remain a
 ↪  constant
%value over the full method, so can be calculated prior to the
 ↪  loop.
[B_0n_Coeff, B_1n_Coeff] = B_Coefficients(t0,t0+h, t0+h, alpha);
```

```
%% Begin Method for ABM

% n is the current time step. Starts at t0, and increases by h
 ↪  each time.
% This will calculate values up to final time tfinal - h, since
 ↪  each loop
% solves the y_(n+1) term.
for n = t0:h:tfinal-h

    % Calculate initial condition term for the time-step desired
    g(nn+1) = calculate_IC(y0,alpha,n+h);


    %Predictor Formula:

    %lag term incrementor and lag summer reset for each pass of
     ↪  n.
    jj = 1;
    predictor_sum_term = 0;

    for j = 0:h:n
        %Calculate heredity term for the predictor, C_Coeff.
        C_Coeff = h^alpha/alpha*((n+h-j)^alpha - (n-j)^alpha);

        %Summing over all previous time steps
        predictor_sum_term = predictor_sum_term +
         ↪  C_Coeff*f_corrector(jj);
```

```
    jj = jj+1;
end

%We are now able to calculate the predictor at the new step:
y_predictor(nn+1) = g(nn+1) +
 ↪  1/gamma(alpha)*predictor_sum_term;

%Evaluate the fdefun using this y_predicted value, at the
 ↪  current time
%step that we are looking to predict at.
f_predictor(nn+1) = fdefun(n+h, y_predictor(nn+1));

% Corrector Formula (This is the same Same as MF-PCL)
  jj = 1;
  y_lag_sum = 0;

  for j = 0:h:n-h
    %Calculate Coefficients for lag term:
    [B_0j_Coeff, B_1j_Coeff] = B_Coefficients(j,j+h, n+h,
     ↪  alpha);

    % Summation Portion of the lag term.
    y_lag_sum = y_lag_sum + B_0j_Coeff*f_corrector(jj) +
     ↪  B_1j_Coeff*f_corrector(jj+1);

    %Increment the interior counter
    jj = jj + 1;
```

```
    end

    % Lag term for this time step is now complete!
    y_lag_final(nn+1) = 1/gamma(alpha)*y_lag_sum  ;

    %Increment term at the new step
    y_increment(nn+1) =
     ↪  1/gamma(alpha)*(B_0n_Coeff*f_corrector(nn) +
     ↪  B_1n_Coeff*f_predictor(nn+1));

    %Corrector term at the new step
    y_corrector(nn+1) = g(nn+1) + y_lag_final(nn+1) +
     ↪  y_increment(nn+1);

    %Solve using corrector y_value guess instead of predicted
     ↪  y_value. To
    %be used in the next prediction stage.
    f_corrector(nn+1) = feval(fdefun, n+h, y_corrector(nn+1));

    %increment n incexing variable.
    nn = nn + 1;

end
%returns the final output variables
Y = y_corrector;
T = t0:h:tfinal;
%% END of Method
```

```
end
```

B.2 Supplemental Software

```
function [B_0n_Coeff_sym, B_1n_Coeff_sym] =
 ↪  B_Coefficients_Analytical(t_j, t_jj, t_nn, alpha)
%This function is not a part of the core software package and is
 ↪  not
%recommended to be used in the FDE solvers. This function is not
 ↪  called in
%any of the methods, but instead can be substituted with the
%'B_Coefficients()' function call to calculate the analytical
 ↪  answer to
%the convolution integral. This function was used in
 ↪  supplemental error
% analysis only.

syms tau

L_j_sym  =  (tau - t_jj) / (t_j - t_jj);
L_jj_sym =  (tau - t_j) / (t_jj - t_j);

B_0n_funct_sym = (t_nn-tau)^(alpha-1) * L_j_sym;
B_0n_Coeff_sym = double(int(B_0n_funct_sym,t_j,t_jj));

B_1n_funct_sym = (t_nn-tau)^(alpha-1) * L_jj_sym;
B_1n_Coeff_sym = double(int(B_1n_funct_sym,t_j,t_jj));
```

```
end

function [A_0n_Coeff, A_1n_Coeff, A_2n_Coeff] =
 ↪  A_Coefficients_Analytical(integrand_low,integrand_high,t_low
 ↪  ,t_mid,t_up,t_eval,alpha)
%This function is not a part of the core software package and is
 ↪  not
%recommended to be used in the FDE solvers. This function is not
 ↪  called in
%any of the methods, but instead can be substituted with the
%'A_Coefficients()' function call to calculate the analytical
 ↪  answer to
%the convolution integral. This function was used in
 ↪  supplemental error
% analysis only.

syms tau

 Q_low =  ((tau - t_mid)/(t_low - t_mid)) * ((tau - t_up)/(t_low
  ↪  - t_up));

 Q_mid =  ((tau - t_low)/(t_mid - t_low)) * ((tau - t_up)/(t_mid
  ↪  - t_up));

 Q_up =  ((tau - t_low)/(t_up - t_low)) * ((tau - t_mid)/(t_up -
  ↪  t_mid));
```

```
%The core of the integral, this portion is the same for A_0,
 ↪  A_1, and A_2.
A_kernel = ((t_eval-tau)^(alpha-1));

%These lines combine the kernel with each appropriate
 ↪  interpolation
%function, combining both function handles into one:
A_0n_funct =  A_kernel*Q_low;
A_1n_funct =  A_kernel*Q_mid;
A_2n_funct =  A_kernel*Q_up;

%These lines compute the integral of the overall function, using
 ↪  the bounds
%input:
A_0n_Coeff =
 ↪  double(int(A_0n_funct,integrand_low,integrand_high));
A_1n_Coeff =
 ↪  double(int(A_1n_funct,integrand_low,integrand_high));
A_2n_Coeff =
 ↪  double(int(A_2n_funct,integrand_low,integrand_high));
end
```